Java

TRÅDAR OCH GRAFIK

FADIL GALJIĆ

Studentlitteratur

Innehållsförteckning

1 Trådar **5**

Ett program med flera trådar 6

Operationer på en tråd 18

Kontrollera en tråds aktivitet 27

Synkroniserad användning av ett objekt 38

Synkroniserad användning av flera objekt 68

Synkronisering av tillståndsberoende operationer 75

Synkroniseringsobjekt 88

Kommunikation mellan trådar 98

2 Kommunikation mellan program **115**

Kommunikation mellan två program 116

Ett serverprogram 130

3 Fönster **147**

En ram 148

Dialoger 160

4 Grafik **181**

Geometriska figurer 182

Rita i en panel 198

Hantera bilder 210

Olika rittekniker 224

Rörliga figurer 235

Koordinatbyte 249

5 Grafiska användargränssnitt 259
Ett grafiskt användargränssnitt 260
Grafiska komponenter 268
Ordna komponenter i en behållare 277
Hantera händelser 299

6 Användargränssnittets funktioner 321
Texthantering 322
Val mellan olika alternativ 334
Strukturerade val 357
Dialoger 373
Datahantering via tabeller 377

Sakregister 386

© FADIL GALJIĆ

Kapitel 1
Trådar

Ett program med flera trådar
Definiera, skapa och starta en tråd • Flera trådar av samma typ
En subklass till klassen Thread • Definiera en tråd lokalt

Operationer på en tråd
Identifiera en tråd • Uppehåll i en tråd
En demontråd • En tråds prioritet

Kontrollera en tråds aktivitet
Avbryta en tråd • Signal till en tråd • Signal till en grupp trådar

Synkroniserad användning av ett objekt
Osynkroniserad användning av ett objekt • Helt synkroniserade objekt
Sammansatta synkroniserade operationer • Optimera synkroniseringen
Synkronisering på klientsidan • Synkroniserade behållare

Synkroniserad användning av flera objekt
Exklusivt ägande av flera objekt • Ett dödläge

Synkronisering av tillståndsberoende operationer
Tillståndsberoende operationer • Synkronisera tillståndsberoende operationer
Synkronisering på klientsidan

Synkroniseringsobjekt
Synkroniseringslås • Läslås och skrivlås • Villkorsobjekt

Kommunikation mellan trådar
Objektöverföring via en kanal • Respons från mottagaren

© FADIL GALJIĆ

Ett program med flera trådar

Definiera, skapa och starta en tråd

Definiera en tråd

I ett program kan flera aktiviteter pågå samtidigt. Uppgifter kan matas in från standardinmatningsenheten samtidigt som en databearbetning pågår. Uppgifter kan matas ut till en fil, medan en beräkning utförs. Det kan finnas flera figurer på skärmen, som rör sig och existerar oberoende av varandra. En bakgrundsaktivitet kan pågå, som avslutas så snart huvudaktiviteten avslutas. Man kan behålla full kontroll över programmet (möjlighet att avbryta en aktivitet, att gå framåt eller bakåt, att välja en viss aktivitet) medan olika aktiviteter pågår. Dessa parallella aktiviteter i ett program kallas för *trådar*. Man har ett program bestående av flera trådar, där varje tråd utför en specifik uppgift.

För att kunna skapa ett program med flera trådar, krävs det ett sätt att definiera en aktivitet som kan exekveras parallellt med andra aktiviteter. Det måste finnas möjlighet att definiera en tråd. Ett antal instruktioner som utgör en viss aktivitet, måste kunna anges och avgränsas. Man måste på något sätt förpacka den kod som ska utföras som en av flera parallella aktiviteter.

En tråd kan definieras genom att en klass som implementerar gränssnittet `java.lang.Runnable` skapas. Detta gränssnitt innehåller bara en metod, som heter `run`. Metoden definieras så här:

```
void run ()
```

Den kod som ska utföras som en särskild aktivitet anges i metoden `run` i en klass som implementerar gränssnittet `Runnable`. En klass som implementerar gränssnittet `Runnable` kan innehålla olika variabler, konstruktorer och metoder. Men det är bara det som anges i klassens `run`-metod som kan utföras som en parallell aktivitet. Naturligtvis kan `run`-metoden använda olika variabler (instansvariabler och statiska variabler) i klassen och anropa andra metoder i klassen.

Säg att man vill skapa ett program med två trådar, där en tråd skriver ut ett meddelande, och en annan tråd skriver ut ett annat meddelande. Man kan då skapa klassen `Skrivare1` som definierar den första tråden, och klassen `Skrivare2` som definierar den andra tråden. Klassen `Skrivare1` kan implementeras så här:

© FADIL GALJIĆ

```
class Skrivare1 implements Runnable
{
    public void run ()
    {
        for (int i = 0; i < 5; i++)
            System.out.println ("gyllene");
    }
}
```

Klassen `Skrivare1` implementerar gränssnittet `Runnable`, och definierar den aktivitet som ska utföras parallellt med andra aktiviteter i metoden `run`. Meddelandet `gyllene` skrivs ut fem gånger.

Klassen `Skrivare2` kan definieras på samma sätt. Här kan ett annat meddelande väljas, till exempel meddelandet `gryningen`:

```
class Skrivare2 implements Runnable
{
    public void run ()
    {
        for (int i = 0; i < 5; i++)
            System.out.println ("gryningen");
    }
}
```

Klassen `Skrivare1` definierar en aktivitet och klassen `Skrivare2` definierar en annan aktivitet. Det går att skapa ett program som utför dessa två aktiviteter parallellt.

Skapa och starta en tråd

Klassen `Skrivare1` definierar en aktivitet och klassen `Skrivare2` definierar en annan aktivitet. Två trådar kan skapas utifrån dessa två klasser. För att kunna skapa en tråd utifrån klassen `Skrivare1`, skapar man först ett objekt av denna klass. Utifrån detta objekt skapas ett objekt av klassen `java.lang.Thread`. Det nya objektet representerar en tråd. Man kan också säga att det nya objektet är en tråd. Så här skapas tråden:

```
Skrivare1    s1 = new Skrivare1 ();
Thread       t1 = new Thread (s1);
```

På samma sätt kan en tråd skapas som utför (är) aktiviteten som anges i klassen `Skrivare2`:

```
Skrivare2    s2 = new Skrivare2 ();
Thread       t2 = new Thread (s2);
```

Nu har två trådar skapats, men de är ännu inte körbara. En tråd måste startas, för att exekveringen av den kod som anges i `run`-metoden ska

© FADIL GALJIĆ

kunna påbörjas. En tråd startas med metoden `start` i klassen `Thread`. Trådarna `t1` och `t2` kan startas så här:

```
t1.start ();
t2.start ();
```

Ett `Runnable`-objekt (ett objekt av en klass som implementerar gränssnittet `Runnable`) och en tråd till detta objekt kan skapas i en enda sats:

```
Thread    t1 = new Thread (new Skrivare1 ());
```

Man gör på detta sätt om `Runnable`-objektet inte ska användas på något annat sätt. Objektet används i detta fall endast för att skapa en tråd.

En tråd kan skapas och startas i en enda sats:

```
new Thread (new Skrivare1 ()).start ();
```

Den namnlösa referensen som returneras av operatorn `new` (som refererar till ett objekt av typen `Thread`) används för att aktivera metoden `start` (i samband med detta objekt).

Ett program som skapar och startar två trådar kan se ut så här:

```
class FleraTradar
{
    public static void main (String[] args)
    {
        Thread    t1 = new Thread (new Skrivare1 ());
        Thread    t2 = new Thread (new Skrivare2 ());

        t1.start ();
        t2.start ();
    }
}
```

Metoden `main` skapar och startar två trådar, och avslutas därefter. Detta innebär dock inte att hela programmet avslutas. Programmet fortlever i trådarna `t1` och `t2`.

När trådarna `t1` och `t2` startas, utför de sina aktiviteter. Tråden `t1` skriver ut meddelandet `gyllene` och tråden `t2` skriver ut meddelandet `gryningen`. Utskriften kan se ut så här:

```
gyllene
gryningen
gyllene
gryningen
gyllene
gryningen
gyllene
gryningen
```


© FADIL GALJIĆ

```
gyllene
gryningen
```

Här skapas en tråd i två steg: först skapas ett `Runnable`-objekt, och utifrån detta objekt skapas en tråd. Ett objekt av typen `Thread` är en tråd. Denna tråd exekverar (via motsvarande `Runnable`-objektet) den kod som finns i den motsvarande `run`-metoden. Man säger därför att tråden `t1` skriver ut meddelandet `gyllene`. Detta är ett objektorienterat sätt att tänka, där ett objekt utför olika aktiviteter. Men en tråd kan även uppfattas som en aktivitet, som representeras via ett objekt av typen `Thread`. I detta fall kan man säga: i tråden (aktiviteten som representeras via) `t1` skrivs meddelandet `gyllene` ut. Man kan även säga så här: tråden (aktiviteten som representeras via) `t1` består av utskriften av meddelandet `gyllene` till standardutmatningsenheten. När man säger att en klass definierar en tråd, menar man att klassen definierar en aktivitet som kan exekveras parallellt med andra aktiviteter.

Schemaläggning av trådar

En dator kan ha flera processorer. I detta fall kan varje tråd exekveras i en särskild processor. Flera aktiviteter kan utföras samtidigt.

Ett program med flera trådar kan även exekveras i en dator med en processor. Programmet växlar då mellan de olika trådarna. Varje tråd tilldelas ett tidsintervall och exekverar (koden i den motsvarande `run`-metoden) i detta intervall. Därefter kan en annan tråd få chans att exekvera. När den tråden avbryts, kan en tidigare avbruten tråd få chans att fortsätta exekvera. Den tråden fortsätter från den punkt där den avbröts.

Ett program kan innehålla de två trådarna `t1` och `t2`, där tråden `t1` skriver ut meddelandet `gyllene` fem gånger, och tråden `t2` skriver ut meddelandet `gryningen` fem gånger. När programmet exekveras kan följande utskrift skapas:

```
gyllene
gryningen
gyllene
gryningen
gyllene
gryningen
gyllene
gryningen
gyllene
gryningen
```

Först exekverar tråden `t1`. Denna tråd hinner skriva ut sitt meddelande en gång, och därefter avbryts tråden. Sedan exekverar tråden `t2`, och den hinner skriva ut sitt meddelande en gång. Tråden `t2` avbryts tillfälligt, och exekveringen av tråden `t1` fortsätter. Växlingen mellan trådarna fortsätter på samma sätt. Varje tråd får ett tidsintervall att exekvera i. På detta vis kan en parallellitet simuleras. Det förefaller som om aktiviteterna verkligen utförs parallellt.

Det går inte att förutsäga hur trådarna i ett program kommer att schemaläggas (eng. schedule) när programmet exekveras. Detta kan variera från en exekvering till en annan. Om programmet med trådarna `t1` och `t2` exekveras på nytt, kan utskriften bli den följande:

```
gryningen
gryningen
gryningen
gryningen
gyllene
gryningen
gyllene
gyllene
gyllene
gyllene
```

Varje tråd utför sin uppgift, och skriver ut sitt meddelande exakt fem gånger. Men i vilken ordning detta sker går inte att förutsäga.

Javas virtuella maskin växlar mellan olika trådar i ett och samma program. Men den virtuella maskinen stöder sig helt på det underliggande operativsystemet. Det är egentligen operativsystemet som organiserar och sköter de olika trådarna. Operativsystemet tilldelar varje tråd (varje trådobjekt, som är ett gränssnitt mellan ett Javaprogram och operativsystemet) de nödvändiga resurserna (register, minne, …), och ser till att varje tråd får tillgång till processorn under en viss tid så att den kan exekveras. Operativsystemet skapar och organiserar sina egna aktiviteter, sina egna trådar, för att kunna utföra de aktiviteter (trådar) som definieras och skapas i ett program. Högnivåtrådarna i ett Javaprogram avbildas till dessa lågnivåtrådar i operativsystemet. När en tråd skapas i Java, skapas även dess motsvarighet i operativsystemet (operativsystemet tar ansvar för handhavandet av tråden).

När en tråd startas (med metoden `start`) blir tråden *körbar*. Tråden kan när som helst få chans att exekveras. De olika trådarna körs växelvis, och varje tråd får sin chans. En tråds exekvering avslutas när den motsvarande `run`-metoden avslutas. Tråden kan därefter inte exekveras på nytt. En tråd som har avslutat sin exekvering kan inte startas på nytt.

© FADIL GALJIĆ

Ett operativsystem har ett särskilt program som växlar mellan olika trådar. En algoritm definieras, som bestämmer vilken av de körbara trådarna som ska exekveras och när exekveringen ska avbrytas. Olika operativsystem använder olika algoritmer. Vanligtvis används så kallad *time slicing*, där ett tidsintervall definieras, och varje tråd kan exekveras under den angivna tiden. På så sätt växlar man mellan olika trådar. Men det finns även andra lösningar. Ett operativsystem kan tillåta att en tråd exekveras tills den avslutas. Först därefter kan en annan tråd exekveras. I detta fall behöver varje tråd avbryta sig själv (med metoden `Thread.sleep`), för att andra trådar ska kunna exekveras.

Flera trådar av samma typ

En klass som definierar en typ av trådar

Klassen `Skrivare1`, som definierar en tråd som skriver ut meddelandet `gyllene` ett antal gånger, kan skapas. Man kan även skapa klassen `Skriva-re2`, som definierar en tråd som skriver ut meddelandet `gryningen` ett antal gånger. En bättre idé är dock att skapa klassen `Skrivare`, som definierar en tråd som skriver ut ett givet meddelande ett antal gånger. Detta meddelande kan anges när ett objekt av klassen skapas. På så sätt kan en och samma klass utnyttjas för att skapa trådar som skriver ut olika meddelanden.

Klassen `Skrivare`, som definierar en tråd som skriver ut ett givet meddelande exakt fyra gånger, kan skapas så här:

```
class Skrivare implements Runnable
{
    private String    meddelande;

    public Skrivare (String meddelande)
    {
        this.meddelande = meddelande;
    }

    public void run ()
    {
        for (int i = 0; i < 4; i++)
            System.out.println (meddelande);
    }
}
```

Denna klass har en viss flexibilitet, eftersom den kan ta emot information om meddelandet utifrån. På ett liknande sätt kan man åstadkomma att

klassen även får information om hur många gånger meddelandet ska skrivas ut (istället för att detta antal fixeras till fyra gånger). I detta fall ska en konstruktor med två parametrar användas.

Två trådar av typen `Skrivare` kan skapas och startas så här:

```
Thread       t1 = new Thread (new Skrivare ("gyllene"));
Thread       t2 = new Thread (new Skrivare ("gryningen"));
t1.start ();
t2.start ();
```

Tråden `t1` skriver ut meddelandet `gyllene`, och tråden `t2` skriver ut meddelandet `gryningen`. Två trådar skapas utifrån en och samma klass. Varje tråd utför den kod som finns i `run`-metoden i klassen `Skrivare`. Varje tråd har dock sitt eget meddelande (det som anges när motsvarande objekt av klassen `Skrivare` skapas) och dessa meddelanden kan inte förväxlas. Ett meddelande hör till ett visst objekt. Varje tråd har även sin egen variabel `i`, som används som räknare i loopen. Alla variabler som skapas i metoden `run` är lokala variabler, och varje tråd skapar dessa variabler för sig. Operativsystemet säkerställer att varje tråd lagrar sina variabler på ett särskilt ställe, så att de inte kan förväxlas.

En vektor av trådar

Klassen `Skrivare` definierar egentligen en typ av trådar. Utifrån denna klass går det att skapa hur många trådar (objekt av typen `Thread`) som helst. Alla dessa trådar utför samma kod, men med olika uppgifter (meddelanden). Det går också att skapa en vektor av trådar. Detta kan illustreras med följande program:

```
class FleraTradarAvSammaTyp
{
    public static void main (String[] args)
    {
        Thread[]    t = new Thread[4];
        t[0] = new Thread (new Skrivare ("öst"));
        t[1] = new Thread (new Skrivare ("väst"));
        t[2] = new Thread (new Skrivare ("syd"));
        t[3] = new Thread (new Skrivare ("nord"));

        for (int i = 0; i < t.length; i++)
            t[i].start ();

        for (int i = 0; i < 4; i++)
            System.out.println ("i mitten");
    }
}
```


© FADIL GALJIĆ

Tråden `t[0]` skriver ut meddelandet `öst` exakt fyra gånger, tråden `t[1]` skriver ut meddelandet `väst` exakt fyra gånger, tråden `t[2]` skriver ut meddelandet `syd` exakt fyra gånger och tråden `t[3]` skriver ut meddelandet `nord` exakt fyra gånger. Förutom dessa fyra trådar finns även `main`-tråden. Denna tråd skapar vektorn och startar trådarna, och sedan skriver meddelandet `i mitten` ut exakt fyra gånger. Den tråd som exekverar den kod som anges i `main`-metoden skapas automatiskt när programmet startas. Men när de fyra andra trådarna har skapats och startats, måste `main`-tråden kämpa för tid att kunna skriva ut sitt meddelande. Denna tråd blir en av flera körbara trådar i programmet.

Följande utskrift kan skapas när programmet `FleraTradarAvSammaTyp` exekveras:

```
i mitten
i mitten
väst
syd
nord
i mitten
öst
väst
syd
nord
i mitten
öst
väst
syd
nord
öst
väst
syd
nord
öst
```

I det här fallet avslutas `main`-tråden först. Men programmet lever vidare även efter detta, eftersom det finns andra trådar som fortsätter exekvera. I ett program med flera trådar betyder inte ett avslutande av `main`-metoden att hela programmet också avslutas.

En subklass till klassen Thread

Definiera och använda en subklass till klassen Thread

För att skapa en tråd i ett program, definierar man först en klass som implementerar gränssnittet `java.lang.Runnable`. Sedan skapas ett objekt av

denna klass, ett `Runnable`-objekt. Med detta objekt som argument skapas en tråd (ett objekt av klassen `java.lang.Thread`). Men en tråd kan även skapas på ett annat sätt. Man kan skapa en subklass till klassen `java.lang.Thread` och omdefiniera metoden `run` (som har samma prototyp i klassen `Thread` som metoden `run` i gränssnittet `Runnable`). Det objekt som sedan skapas av klassen är en tråd. När objektet startas (med metoden `start` som ärvs från klassen `Thread`) blir tråden körbar.

Klassen `Skrivare` definierar en tråd som skriver ut ett givet meddelande exakt fem gånger. Denna klass kan skapas som en subklass till klassen `Thread` (istället för att implementera gränssnittet `java.lang.Runnable`). I klassen ska metoden `run` från klassen `Thread` (som är en tom metod i klassen `Thread`) omdefinieras. Så här kan detta göras:

```
class Skrivare extends Thread
{
    private String    meddelande;

    public Skrivare (String meddelande)
    {
        this.meddelande = meddelande;
    }

    public void run ()
    {
        for (int i = 0; i < 5; i++)
            System.out.println (meddelande);
    }
}
```

Ett objekt av klassen `Skrivare` är en tråd (ett objekt av superklassen `Thread`). Den ärvda metoden `start` kan användas i samband med detta objekt och göra tråden till en körbar tråd. När tråden exekveras, exekveras den koden som finns i metoden `run` i klassen `Skrivare`.

Klassen `Skrivare` definierar en typ av trådar. Utifrån denna klass kan ett valfritt antal trådar skapas. Så här kan man göra:

```
Skrivare    s1 = new Skrivare ("gyllene");
Skrivare    s2 = new Skrivare ("gryningen");
s1.start ();
s2.start ();
```

Tråden `s1` skriver ut meddelandet `gyllene` exakt fem gånger och tråden `s2` skriver ut meddelandet `gryningen` exakt fem gånger. Detta kan ge följande utskrift:

```
gryningen
gryningen
```

 © FADIL GALJIĆ

```
gryningen
gryningen
gyllene
gryningen
gyllene
gyllene
gyllene
gyllene
```

En Runnable-klass eller en subklass till klassen Thread

När man använder en klass som implementerar gränssnittet `Runnable`, måste man först skapa ett objekt av denna klass, och utifrån detta objekt en tråd (ett objekt av typen `Thread`). När en subklass till klassen `Thread` används, skapas en tråd direkt, eftersom ett objekt av klassen är en tråd. Dessutom kan alla metoder som finns i klassen `Thread` appliceras i samband med ett objekt av denna subklass (alla metoderna ärvs). Metoden `start`, till exempel, kan appliceras direkt på detta objekt. Både de metoder som finns i klassen `Thread` och metoderna i subklassen kan appliceras på objektet. Metoderna i klassen `Thread` kan inte appliceras direkt på ett objekt av en klass som implementerar gränssnittet `Runnable`. Först måste ett objekt av typen `Thread` skapas. Men de metoder som definieras i den klassen som implementerar gränssnittet `Runnable` kan inte användas i samband med detta `Thread`-objekt. Det är därför lättare att skapa och hantera trådar från en subklass till klassen `Thread`, än att skapa och hantera trådar från en klass som implementerar gränssnittet `Runnable`.

Nya klasser skapas ofta genom att härledas från andra klasser. En klass placeras på ett lämpligt ställe i en klasshierarki. Om en klass ärver från en annan klass, kan den inte ärva från klassen `Thread`, eftersom multipelt arv inte är tillåtet i Java. Om man vill kunna skapa trådar från den klassen, måste den implementera gränssnittet `Runnable`. Arvsrelationen *är en tråd* vid arv från klassen `Thread` kan också förefalla tveksam. Kan man till exempel säga att en boll som rör sig på skärmen "är en tråd". En boll är en figur eller något liknande, men inte en tråd. En tråd kan skapas utifrån en boll. Därför kan man vilja definiera sina trådar genom att skapa klasser som implementerar gränssnittet `Runnable`. En klass som implementerar gränssnittet `Runnable` kan å ena sidan ärva från en annan klass, och å andra sidan bevaras en konceptuell renhet när det gäller relationen är-en.

Definiera en tråd lokalt

En klass som implementerar gränssnittet `Runnable` kan skapas, och utifrån ett objekt av denna klass kan en tråd skapas. En och samma definitionsklass kan användas för att skapa flera trådar i ett program. Man kan skapa trådar av en och samma klass i flera olika program.

I vissa situationer skapas en klass för att definiera en enda tråd i ett enda program. Denna klass kan i så fall skapas som en lokal, anonym (namnlös) klass. Man kan skapa klassen där den behövs, och samtidigt skapa ett objekt av klassen. Detta objekt kan användas via en referens av typen `Runnable`. En sådan referens kan referera till ett godtyckligt objekt av en klass som implementerar gränssnittet `Runnable`. Så här kan man göra:

```
Runnable    r = new Runnable ()
    {
        public void run ()
        {
            for (int i = 0; i < 5; i++)
                System.out.println ("gyllene");
        }
    };
```

Klassens variabler och metoder definieras mellan två klamrar. Man kan inte definiera konstruktorer när en klass skapas på detta vis. En konstruktor har samma namn som motsvarande klass, men denna klass saknar namn. I detta fall definieras endast metoden `run` i klassen. Klassen definieras och ett objekt av den skapas i en enda sats (därför finns det ett semikolon på slutet). Man refererar till objektet via referensen `r` av typen `Runnable`. Operatorn `new` används för att skapa ett objekt av denna klass. Efter operatorn `new` följer klassens definition. Man anger först att klassen implementerar gränssnittet `Runnable`. På denna plats kan namnet på en klass anges i stället för namnet på ett gränssnitt. Den definierade klassen blir i så fall en subklass till den angivna klassen, och argument till superklassens konstruktor kan anges mellan parenteserna. När ett gränssnitt anges har de två parenteserna som följer efter gränssnittets namn ingen funktion, men måste ändå anges (enligt Javas syntax).

Man definierar en klass lokalt, på den platsen där den behövs. Man skapar samtidigt ett objekt av den klassen. Detta objekt refereras av referensen `r`. Eftersom objektet är ett `Runnable`-objekt kan det användas för att skapa en tråd:

```
Thread    t1 = new Thread (r);
```


© FADIL GALJIĆ

Om ett `Runnable`-objekt endast används för att skapa en tråd, kan detta objekt skapas samtidigt som tråden skapas. Man kan i så fall definiera en klass, skapa ett objekt av klassen och skapa en tråd utifrån detta objekt i en enda sats:

```
Thread    t2 = new Thread (new Runnable ()
    {
        public void run ()
        {
            for (int i = 0; i < 5; i++)
                System.out.println ("gryningen");
        }
    } );
```

När trådarna `t1` och `t2` skapats, kan de startas:

```
t1.start ();
t2.start ();
```

En lokal, anonym klass som implementerar gränssnittet `Runnable` kan definieras. På samma sätt kan en klass som ärver från klassen `Thread` definieras:

```
Thread    t = new Thread ()
    {
        public void run ()
        {
            for (int i = 0; i < 16; i++)
                System.out.println ("gyllene");
        }
    };
```

En tråd kan definieras, skapas och startas i en enda sats:

```
(new Thread ()
    {
        public void run ()
        {
            for (int i = 0; i < 16; i++)
                System.out.println ("gryningen");
        }
    }).start ();
```

Det går alltså att definiera och implementera en klass när ett objekt av klassen skapas. Eftersom denna klass inte ska användas vid ett senare tillfälle, behöver inte dess namn anges. Det skapas en lokal, anonym klass. Dessa klasser kan användas när man vill definiera en enda tråd. En tråd knyts enkelt och elegant till en konkret `run`-metod.

© FADIL GALJIĆ

Operationer på en tråd

Identifiera en tråd

Man kan skapa flera trådar av samma typ i ett och samma program. Alla dessa trådar exekverar kod i en och samma `run`-metod. De "passerar genom" metoden, och kan tillfälligt avbrytas på ett eller flera ställen i metoden. När en tråd utför koden i en `run`-metod, skapar den alla de variabler som skapas i denna metod. En tråd har en egen uppsättning lokala variabler, och dessa variabler kan inte förväxlas med en annan tråds variabler. Varje tråd tilldelas en särskild plats där dess variabler lagras.

Man kan vid vissa tillfällen vilja ha en referens till den tråd som exekverar koden i `run`-metoden. Det kan gälla ett anrop till en metod i samband med den exekverande tråden. Metoden `currentThread` i klassen `java.lang.Thread` returnerar en referens till den exekverande tråden. Det är en statisk metod som returnerar en referens av typen `Thread`. Metoden kan användas så här:

```
Thread    t = Thread.currentThread ();
```

Referensen `t` refererar till den exekverande tråden (det objekt av typen `Thread` som vid detta tillfälle exekverar den angivna satsen i `run`-metoden). Denna referens kan användas till att anropa en instansmetod i klassen `Thread` i samband med den refererade tråden. Man kan till exempel anropa metoden `getName`, som returnerar namnet på den aktuella tråden:

```
String    namn = t.getName ();
```

Olika trådar kan passera genom en och samma `run`-metod. Metoden `currentThread` gör det möjligt att få kontakt med en tråd som passerar genom `run`-metoden.

Klassen `Identifierare`, som definierar en tråd som skriver ut sitt namn till standardutmatningsenheten, kan skapas på följande vis:

```
class Identifierare implements Runnable
{
    public void run ()
    {
        Thread    t = Thread.currentThread ();
        String    namn = t.getName ();
        System.out.println (namn);
    }
}
```


© FADIL GALJIĆ

Den exekverande tråden får en referens till sig själv, och bestämmer sitt namn via denna referens. Tråden skriver sedan ut namnet. Variablerna `t` och `namn` är lokala variabler i metoden `run`. Varje tråd som passerar genom denna metod skapar sina egna variabler. Dessa variabler för de olika trådarna kan inte förväxlas.

En tråd kan namnges när den skapas:

```
Thread    t1 = new Thread (new Identifierare (), "tråd t1");
```

Här skapas en tråd av typen `Identifierare` som får namnet `tråd t1`. Det är även möjligt att först skapa en tråd och sedan ange trådens namn:

```
Thread    t2 = new Thread (new Identifierare ());
t2.setName ("tråd t2");
```

Här skapas en tråd av typen `Identifierare`, och dess namn sätts sedan till `tråd t2`.

Om trådarna `t1` och `t2` startas, utför de den kod som finns i `run`-metoden i klassen `Identifierare`. Tråden `t1` bestämmer och skriver ut sitt namn (`tråd t1`) och tråden `t2` bestämmer och skriver ut sitt namn (`tråd t2`). Så här kan utskriften se ut:

```
tråd t2
tråd t1
```

Uppehåll i en tråd

Vänta en angiven tid

En tråd kan göra ett eller flera uppehåll under sin exekvering. Dessa uppehåll planeras när tråden definieras. Uppehållen byggs in i en tråds definitionsklass och blir en del av trådens normala beteende. De införs inte utifrån, från en annan tråd.

Ett sätt att införa ett uppehåll i en tråd är att använda metoden `sleep` i klassen `Thread`. Metoden är en statisk metod, som inför uppehåll i den anropande tråden. Uppehållets längd anges i ett eller två argument till metoden `sleep`. Uppehållet kan anges i antalet millisekunder, eller i antalet millisekunder och nanosekunder. Metoden `sleep` kan anropas så här:

```
Thread.sleep (1000);
```

Tråden som utför det här anropet gör ett uppehåll under en sekund (`1000` millisekunder). Tråden är *blockerad* under detta tidsintervall, och kan inte exekvera koden i den motsvarande `run`-metoden. När denna tid har gått

blir tråden *körbar* igen. När tråden åter är körbar, kan den när som helst få chans att fortsätta exekvera.

Klassen `Skrivare` definierar en tråd som skriver ut ett givet meddelande exakt fyra gånger. I denna klass kan ett uppehåll definieras efter varje utskrift. På så sätt kan trådens hastighet minskas. Så här kan detta göras:

```
class Skrivare implements Runnable
{
    private String    meddelande;

    public Skrivare (String meddelande)
    {
        this.meddelande = meddelande;
    }

    public void run ()
    {
        for (int i = 0; i < 4; i++)
        {
            System.out.println (meddelande);

            try
            {
                Thread.sleep (500);
            }
            catch (InterruptedException e)
            {}
        }
    }
}
```

När en tråd exekverar metoden `run` i klassen `Skrivare`, kommer den så småningom till anropet till metoden `sleep`. Tråden blockeras under `500` millisekunder, och kan inte exekvera under denna tid. Den blir körbar (kan få chans att exekvera) igen när denna tid har gått. När tråden åter får chans att exekvera, fortsätter den direkt efter anropet till metoden `sleep`. På så sätt införs ett uppehåll mellan två successiva utskrifter.

En tråds uppehåll kan avbrytas utifrån, från en annan tråd. Detta sker med ett anrop till instansmetoden `interrupt` (i klassen `Thread`) i samband med denna tråd. Detta medför att metoden `sleep` avslutas omedelbart och kastar ett undantag av typen `java.lang.InterruptedException`.

Ett undantag av typen `InterruptedException` är ett kontrollerat undantag, som antingen måste fångas och hanteras eller deklareras. Om metoden `sleep` anropas i metoden `run` (den kan anropas även i andra metoder), måste undantaget fångas och hanteras. Detta undantag ska inte deklareras, eftersom det inte är deklarerat i gränssnittet `Runnable` (ett

© FADIL GALJIĆ

kontrollerat undantag kan inte anges i en metods huvud, om inte undantaget finns med i gränssnittet). Om metoden `interrupt` inte används, kan inte ett undantag av typen `InterruptedException` kastas. I detta fall kan undantaget ignoreras (`catch`-blocket lämnas tomt).

Metoden `run` (i en klass som definierar en tråd) innehåller ofta en loop. Ett anrop till metoden `sleep` kan i så fall placeras i denna loop. Genom att välja ett lämpligt värde som argument till metoden `sleep` kan trådens hastighet anpassas. Men det finns en anledning till att ett anrop till metoden `sleep` placeras i en loop. På så sätt kan en tråd blockeras varje gång den passerar genom loopen, och därmed ge andra trådar möjlighet att exekvera. Processortiden fördelas jämnare mellan olika trådar. På så vis blir man mindre beroende av hur ett konkret operativsystem schemalägger trådar i ett program.

En körbar tråd och en blockerad tråd

En tråd som är startad och som ännu inte avslutat sin exekvering, kan finnas i två olika tillstånd. Tråden kan vara körbar, och kan i så fall få möjlighet att exekvera koden i den motsvarande `run`-metoden.. Men den kan även vara blockerad. I så fall kan tråden inte exekvera förrän orsaken till blockeringen upphör. En orsak till att tråden är blockerad kan vara att tråden anropat metoden `sleep`. Tråden förblir blockerad under den angivna tiden, eller tills metoden `interrupt` anropas i samband med tråden (om metoden `interrupt` anropas innan den angivna tiden har gått, avslutas metoden `sleep`). En tråd blockeras vid anrop till metoden `sleep`, men det finns även andra situationer då en tråd blockeras. Det kan inträffa vid anrop till metoden `wait` i klassen `java.lang.Object` (när en tråd väntar på att situationen för exekvering blir gynsammare), eller vid inmatning och utmatning.

När en tråd har startats börjar dess existens. Tråden existerar tills den avslutar metoden `run` (normalt eller genom ett undantag). En tråd kan existera på två olika sätt: antingen som en körbar tråd (som antingen exekverar eller väntar på att exekvera) eller som en blockerad tråd. Med metoden `isAlive` (en instansmetod) i klassen `Thread` kan man kontrollera om en given tråd fortfarande existerar. Metoden returnerar `true` om den aktuella tråden existerar när metoden anropas.

© FADIL GALJIĆ

Avbryta exekveringen av en tråd

Klassen `Thread` innehåller, förutom metoden `sleep`, en metod till som tillfälligt avbryter exekveringen av en tråd. Denna metod heter `yield`. Det är en statisk metod, som avbryter den exekverande tråden. Metoden `yield` kan användas för att införa ett uppehåll i metoden `run` i klassen `Skrivare`:

```
public void run ()
{
    for (int i = 0; i < 4; i++)
    {
        System.out.println (meddelande);
        Thread.yield ();
    }
}
```

Metoden `yield` avbryter tillfälligt exekveringen av en tråd, men blockerar inte tråden. Den avbrutna tråden förblir körbar, och det kan hända att den väljs att exekvera på nytt. Det finns inga garantier för att en annan tråd får chansen att exekvera. Det är därför bättre att använda metoden `sleep`. Om man inte vill blockera en tråd under en längre tid, kan man välja korta väntetider i metoden `sleep` (ett antal nanosekunder).

Vänta på en tråd

En tråd kan vänta på en annan tråd. Ett sätt att göra detta är att använda metoden `join` i klassen `Thread`. Metoden anropas i samband med den tråd som inväntas. Metoden `join` är en instansmetod, och därför krävs det en referens till den tråd som inväntas för att kunna anropa metoden. Om `t` representerar en sådan referens, kan metoden anropas så här:

```
t.join ();
```

Tråden som exekverar satsen blockeras, och förblir blockerad så länge tråden `t` lever. När tråden `t` avslutar sin aktivitet (den motsvarande `run`-metoden avslutas), blir den blockerade tråden körbar igen (kan få chansen att exekvera). På så sätt kan aktiviteter synkroniseras mellan två trådar: en aktivitet ska endast fortsätta när en annan aktivitet avslutas. Olika aktiviteter kan ordnas i tiden. En aktivitet kan ha sin naturliga plats efter en annan aktivitet. Det kan även hända att en aktivitet är beroende av resultatet av en annan aktivitet.

Om metoden `interrupt` anropas på den tråd som väntar i metoden `join`, avslutas metoden `join` (väntandet avslutas) omedelbart. Metoden kastar

© FADIL GALJIĆ

ett undantag av typen `InterruptedException`. Detta undantag måste antingen fångas och hanteras, eller deklareras. Om inte metoden `interrupt` anropas, kan undantaget ignoreras (det motsvarande `catch`-blocket lämnas tomt). Detta kan göras så här:

```
try
{
    t.join ();
}
catch (InterruptedException e)
{}
```

Metoden `join` kan anropas i `main`-metoden i ett program:

```
System.out.println ("start");

Thread    t = new Thread (new Skrivare ("lycka"));
t.start ();

try
{
    t.join ();
}
catch (InterruptedException e)
{}

System.out.println ("slut")
```

Meddelandet `start` skrivs ut. Sedan skapas och startas tråden `t`. Därefter inväntas att tråden `t` ska avsluta sin aktivitet. När tråden `t` avslutar sin aktivitet blir `main`-tråden körbar igen, och fortsätter att exekvera. Den skriver ut meddelandet `slut`, och avslutas sedan.

I detta fall skapar man en tråd och väntar på denna tråd i en och samma tråd (i `main`-tråden). Det kan dock hända att en tråd skapas på ett ställe och inväntas på ett annat ställe. I detta fall måste det finnas en referens till den tråd som inväntas även på den plats där man väntar (i en annan tråd). En referens (som refererar till en tråd) kan tillföras till en tråd när tråden skapas. Denna referens anges som argument till motsvarande konstruktor. En referens kan även tillföras via en lämplig metod, som definieras i trådens definitionsklass.

En demontråd

En tråd kan göras till en *demontråd*. Detta gör man med metoden `setDaemon` i klassen `Thread`. Metoden (som är en instansmetod) appliceras i samband med en tråd, och argumentet `true` anges till metoden:

```
t.setDaemon (true);
```

Tråden `t` (`t` är en referens till ett objekt av typen `Thread`) görs här till en demontråd. Metoden `setDaemon` måste appliceras på en tråd innan tråden startas (annars genereras ett undantag av typen `java.lang.IllegalThreadStateException`). En tråd som inte är en demontråd kallas vanligtvis för en *användartråd*. Ett program kan innehålla flera användartrådar och demontrådar.

Det som skiljer en demontråd från andra trådar (användartrådar) är att en demontråd kan avslutas även innan den motsvarande `run`-metoden avslutas. En demontråd avslutas automatiskt när alla användartrådar avslutas. En kort stund efter det att den sista användartråden avslutats, avslutas även alla demontrådar.

En demontråds enda syfte är att betjäna andra trådar i programmet. Det kan vara en bakgrundstråd, eller en tråd som utför en speciell tjänst åt en eller flera andra trådar. När dessa trådar försvinner, finns det inte längre någon anledning för demontråden att fortleva. Den avslutas, och hela programmet avslutas.

Klassen `Understrykare`, som definierar en tråd som skriver ut en understrykningslinje i en oändlig loop, kan skapas så här:

```
class Understrykare implements Runnable
{
    public void run ()
    {
        while (true)
        {
            System.out.println ("------------------");

            try
            {
                Thread.sleep (500);
            }
            catch (InterruptedException e)
            {}
        }
    }
}
```


© FADIL GALJIĆ

Förutom klassen `Understrykare`, kan man skapa en klass som definierar en tråd som skriver ut ett givet meddelande exakt fem gånger. Denna klass kan kallas för `Skrivare`. Ett program kan skapa en tråd av typen `Skrivare` och en tråd av typen `Understrykare`. Detta kan göras så här:

```
Thread    t1 = new Thread (new Skrivare ("gyllene gryningen"));
Thread    t2 = new Thread (new Understrykare ());
t2.setDaemon (true);

t1.start ();
t2.start ();
```

Tråden `t1` skriver ut sitt meddelande ett antal gånger, och tråden `t2` stryker under det som skrivs ut. Tråden `t2` utför en tjänst åt tråden `t1`, och dess aktivitet är meningsfull så länge `t1` existerar. När tråden `t1` avslutar sin aktivitet, har tråden `t2` inte längre något att stryka under (inga meddelanden skrivs ut). Därför görs tråden `t2` till en demontråd. Denna tråd avslutas automatiskt när tråden `t1` avslutas, trots att dess `run`-metod innehåller en oändlig loop. Som resultat av trådarnas samtidiga aktiviteter kan följande utskrift skapas:

```
gyllene gryningen
-----------------
gyllene gryningen
-----------------
gyllene gryningen
-----------------
gyllene gryningen
-----------------
gyllene gryningen
-----------------
-----------------
```

En tråds prioritet

Ordningen i vilken olika trådar exekverar kan påverkas genom att olika prioriteter anges för trådarna. I så fall exekverar den tråd som har högst prioritet först. Om det finns flera trådar med samma prioritet, kan var och en av dessa trådar få chans att exekvera. En tråd som har en lägre prioritet än den högsta, måste vänta tills alla trådar med högre prioritet avslutar sin exekvering. En tråd med en lägre prioritet får endast möjlighet att exekvera om alla trådar med högre prioritet är blockerade. En tråd kan till exempel blockera sig genom att anropa metoden `sleep`. Detta innebär att en tråd med lägre prioritet kan få chans att exekvera (om det inte finns någon annan körbar tråd med högre prioritet).

© FADIL GALJIĆ

En tråds prioritet anges med metoden `setPriority` i klassen `Thread`. Denna metod tar emot ett heltal mellan `1` och `10` som argument (om ett heltalsvärde utanför detta intervall anges, kastar metoden ett undantag av typen `java.lang.IllegalArgumentException`). Heltalet anger den aktuella trådens prioritet. En tråds prioritet kan avläsas med metoden `getPriority`. Metoderna `setPriority` och `getPriority` kan användas så här:

```
Thread    t = new Thread ();
t.setPriority (7);
int    p = t.getPriority ();  // p blir 7
```

Det förvalda värdet för en prioritet är `5`. Huvudtråden (`main`-tråden) har denna prioritet så länge inte detta värde ändras. Om en tråds prioritet inte anges, får tråden samma prioritet som trådens skapare (som också är en tråd). En tråds prioritet kan ändras under exekveringen. På så sätt kan en kodsekvens köras med en prioritet, och en annan kodsekvens med en annan prioritet. Man kan ändra prioriteten för den exekverande tråden inuti `run`-metoden. Detta kan göras så här:

```
Thread.currentThread ().setPriority (10);
```

Ett operativsystem behöver inte stödja tio olika prioriteter. I så fall avbildas de i källkoden angivna prioriteterna till de prioriteter som finns på den aktuella plattformen. På så vis kan två trådar som tilldelas olika prioriteter i källkoden ha samma prioritet på en konkret plattform.

Istället för att prioriteterna fastställs till konkreta heltalsvärden, kan motsvarande konstanter som finns i klassen `Thread` användas. Dessa konstanter är `MIN_PRIORITY` (den lägsta prioriteten), `NORM_PRIORITY` (den förvalda prioriteten) och `MAX_PRIORITY` (den högsta prioriteten). Så här kan en sådan konstant användas:

```
Thread    t = new Thread ();
t.setPriority (Thread.NORM_PRIORITY - 1);
```

Vissa operativsystem stödjer inte alls prioritet för olika trådar. Därför behöver man känna till på vilka plattformar ett program ska exekveras, när man skapar ett program som baseras på olika prioriteter. I princip ska man inte stödja sig på olika prioriteter när man utformar sina program.

© FADIL GALJIĆ

Kontrollera en tråds aktivitet

Avbryta en tråd

Avbryta en tråd från en annan tråd

En tråd kan skapa och starta en annan tråd. En tråd kan även avbryta en annan tråd, och eventuellt avsluta exekveringen av denna tråd.

En tråd avslutas normalt när den utför den kod som finns i den motsvarande `run`-metoden. Denna metod representerar en meningsfull aktivitet, och det kan därför vara olämpligt att avbryta en tråd mitt i denna aktivitet. Men det går att identifiera de punkter i `run`-metoden där tråden kan avbrytas, och tillåta avbrott endast vid dessa punkter. Det betyder att en tråd som ska avbrytas måste acceptera avbrottet, och aktivt delta i detta avbrott. En tråd ska inte avbrytas utifrån utan trådens tillåtelse.

En tråd kan bearbeta uppgifter om olika personer i en loop. Man kan till exempel ändra adresser för olika personer. Först kan personens gata och nummer ändras, och sedan personens ort. En sådan ändring representerar en komplett cykel, och man kan inte tillåta att den tråd som utför denna ändring avbryts mitt i cykeln. Tråden kan avbrytas antingen innan cykeln påbörjas, eller när cykeln avslutas (och innan nästa cykel påbörjas). Man ska bara avbryta den tråd som känner till dessa punkter vid vilka den kan avbrytas.

Metoden `interrupt` i klassen `java.lang.Thread` kan användas för att begära ett avbrott för en tråd. Denna metod appliceras i en tråd i samband med en annan tråd. Om referensen `t` refererar till ett objekt av typen `Thread`, kan metoden anropas så här:

```
t.interrupt ();
```

Ett anrop till metoden `interrupt` avslutar inte måltråden (den tråd som metoden `interrupt` anropas i samband med). Det som händer är att måltråden märker att en annan tråd skickat en `interrupt`-signal till den. Måltråden väljer sedan själv hur den ska reagera. Den kan tolka signalen som en begäran om ett avbrott och bestämma sig för att avsluta exekveringen. Måltråden väljer i så fall en lämplig punkt i exekveringen och avslutas vid denna punkt.

Varje objekt av typen `Thread` har en inbyggd boolesk variabel, en flagga. Det är denna flagga som metoden `interrupt` använder för att signalera sitt krav. När metoden `interrupt` anropas i samband med en tråd, sätts

© FADIL GALJIĆ

trådens flagga till true. Målträden kan vid lämpliga punkter granska flaggan, och på så sätt kontrollera om en interrupt-signal inkommit. Om flaggan har värdet true, har en interrupt-signal inkommit och tråden måste reagera på ett lämpligt sätt.

Den flaggan som sätts med metoden interrupt kan kontrolleras på en eller flera lämpliga platser i run-metoden. Statusen för den exekverande trådens interrupt-flagga kontrolleras med metoden interrupted i klassen Thread. Detta är en statisk metod som returnerar flaggans värde i den exekverande tråden. Anropet till denna metod har också en bieffekt: flaggans värde återställs till false, så att en eventuell nästkommande interrupt-signal kan registreras. Metoden interrupted kan användas på följande sätt i run- metoden:

```
if (Thread.interrupted ())
    // reagera på interrupt-signalen
```

Det finns dock en sak som komplicerar användandet av interrupt-signalen. Det finns inga garantier för att anropet till interrupt-metoden resulterar i att värdet för den motsvarande flaggan i målträden sätts till true. Detta sker om tråden är körbar när interrupt-metoden anropas. Men målträden kan vara blockerad när interrupt-metoden anropas. Tråden kan vara blockerad i metoden sleep, metoden wait eller metoden join. I detta fall sätts inte den motsvarande flaggan till true, utan blockeringsmetoden avslutas med att kasta ett undantag av typen java.lang.InterruptedException. Samtidigt rensas avbrottsflaggan (sätts till false). Ett undantag av typen InterruptedException signalerar att metoden interrupt anropats på målträden medan den var blockerad. Även i detta fall måste därför lämpliga åtgärder vidtas. Man kan göra så här:

```
try
{
    Thread.sleep (10);
}
catch (InterruptedException e)
{
    // reagera på interrupt-signalen
}
```

Målträden kan alltså informeras om interrupt-signalen på två olika sätt: antingen genom motsvarande flagga eller genom ett undantag av typen InterruptedException. Man bör då och då kontrollera flaggans status, och reagera på ett lämpligt sätt om en interrupt-signal mottagits. Man bör också reagera på ett lämpligt sätt om ett undantag av typen Interrup-

© FADIL GALJIĆ

tedException kastas (eftersom detta också är ett tecken på att interrupt-signalen kommit).

En räknare och en kontrollör

Ett program kan innehålla en räknare, som räknar antalet enheter av en viss typ. Det kan även finnas en kontrollör som aktiverar räknaren och avbryter räknarens aktivitet vid en lämplig tidpunkt. Klassen `Raknare`, som definierar en tråd som simulerar en räknare, kan skapas. Klassen `Kontrollor`, som simulerar en kontrollörs aktivitet, kan också skapas.

Kontrolltråden måste avbryta räknartrådens aktivitet vid ett visst tillfälle. Kontrolltråden behöver anropa metoden `interrupt` i samband med räknartråden. Räknartråden ska då och då titta på sin `interrupt`-flagga, och om den är satt ska tråden på ett lämpligt sätt avsluta sin aktivitet. Räknartråden måste också avsluta sin aktivitet på ett lämpligt sätt om ett undantag av typen `InterruptedException` kastas.

Klassen `Kontrollor` kan definieras och implementeras så här:

```
class Kontrollor implements Runnable
{
    private Thread raknare;

    public void run ()
    {
        raknare = new Thread (new Raknare ());
        raknare.start ();

        try
        {
            int    p = (int) (20 * Math.random () + 1);
            Thread.sleep (p * 1000);
        }
        catch (InterruptedException e)
        {}

        raknare.interrupt ();
    }
}
```

En räknartråd skapas och startas. Metoden väntar medan räkningen pågår (detta simuleras med en slumpmässig tid), och därefter avbryts räkningen (`interrupt`-signalen skickas till räknartråden). Sedan avslutar kontrolltråden sin aktivitet.

En räknare måste vara förberedd på `interrupt`-signalen och reagera på den på ett lämpligt sätt. Klassen `Raknare` kan definieras och implementeras så här:

```
class Raknare implements Runnable
{
    private int    antal = 0;

    public void run ()
    {
        int    v;

        while (true)
        {
            try
            {
                v = (int) (4 * Math.random () + 1);
                Thread.sleep (v * 500);
            }
            catch (InterruptedException e)
            {
                System.out.println ();
                break;
            }

            antal++;
            System.out.print (antal + " ");

            if (Thread.interrupted ())
            {
                System.out.println ():
                break;
            }
        }
    }
}
```

Här inväntas en enhet en slumpmässigt vald tid, och när denna enhet kommer (när väntetiden har gått ut) räknas räknaren upp (variabeln `antal` ökas med `1`). Sedan visas räknarens värde (variabeln `antal` skrivs ut). Därefter inväntas nästa enhet, och allt upprepas på samma sätt. Aktiviteten avslutas när räknartråden tar emot en `interrupt`-signal.

I slutet av varje cykel kontrolleras (via metoden `interrupted`) om en `interrupt`-signal kommit. I fall att en `interrupt`-signal kommit påbörjas en ny rad, och loopen avslutas. På så sätt avslutas räknarens `run`-metod (räknaren avslutas). Tråden avslutas även om en `interrupt`-signal inkommer medan metoden `sleep` exekveras (medan räknartråden är blockerad). I detta fall avslutas metoden `sleep` omedelbart genom att ett undantag av

© FADIL GALJIĆ

typen InterruptedException kastas (interrupt-flaggan sätts inte i detta fall). Undantaget fångas och hanteras på så sätt att en ny rad påbörjas och loopen avbryts (varvid run-metoden avslutas).

Signalen interrupt kan avbryta räknartråden eftersom tråden accepterar detta. Räknartråden undersöker om interrupt-signalen kommit, och går i så fall till en ny rad och avslutar därefter sin aktivitet. Efter mottagandet av interrupt-signalen kan måltråden utföra vissa nödvändiga operationer (gå till ny rad) innan den avslutas.

Räknartråden bestämmer även den plats i run-metoden där den ska avbrytas. En tråd bör inte avbrytas mitt i en odelbar aktivitet. Räknartråden, till exempel, ska inte avbrytas efter det att räknaren räknats upp och innan det att räknarens värde visats. Om man gör det, visas en felaktig information om antalet räknade enheter. interrupt-flaggans tillstånd ska kontrolleras på en lämplig plats i run-metoden.

Om catch-blocket för undantag av typen InterruptedException reagerar på interrupt-signalen, behöver inte tråden avslutas på denna plats. Det räcker med att man signalerar att interrupt-signalen kommit genom att sätta en boolesk variabel. Denna variabel kan sedan analyseras, och situationen hanteras på ett lämpligt sätt på ett annat ställe i metoden run. Det är inte nödvändigt att skapa en ny variabel för detta ändamål, istället kan den flagga som redan finns i varje objekt av typen Thread, och som används i samband med metoden interrupt, användas. En tråd måste i detta fall sätta sin egen flagga. Tråden måste först erhålla en referens till sig själv med metoden currentThread, och därefter anropa metoden interrupt med denna referens. Detta kan göras så här:

```
catch (InterruptedException e)
{
    Thread.currentThread.interrupt ();
}
```

Med den här tekniken sätts interrupt-flaggan i måltråden när en interrupt-signal kommer, oavsett om tråden är körbar eller blockerad i detta ögonblick.

För att testa klasserna Raknare och Kontrollor kan man skapa ett program, och en kontrolltråd i programmet. Man startar sedan kontrolltråden (som sedan skapar och startar sin räknare) och på så sätt påbörjar aktiviteten. När programmet exekveras, kan följande utskrift skapas:

```
1 2 3 4 5 6 7 8 9 10
```

Räknaren räknar antalet enheter, och visar sitt tillstånd efter varje ändring. Detta upprepas ett slumpmässigt antal gånger (i detta fall tio gång-

er). Räknaren avslutar sin aktivitet när den får `interrupt`-signalen från kontrolltråden.

Signaler till en tråd

Skicka en signal till en tråd

Metoden `interrupt` kan användas för att avbryta en tråd. Men denna metod kan även användas för att kontrollera en tråds aktivitet. Metoden `interrupt` anropas i samband med en tråd, och tråden kan se att `interrupt`-signalen kommit. Tråden bestämmer sedan hur den ska reagera på signalen. Den kan avsluta sin aktivitet, men den kan även reagera på ett annat sätt. Den kan till exempel avsluta en typ av aktivitet och påbörja en annan typ av aktivitet. Den kan även ignorera `interrupt`-signalen.

Man kan införa en statusvariebel i en klass som definierar en tråd, och en metod som sätter den variabelns värde. Man kan kalla den statusvariabeln för `status`, och den metoden för `setStatus`. Statusvariabeln kan vara en heltalsvariabel, som kan anta olika heltalsvärden: `-1, 0, 1, 2` och så vidare. Variabelns värde kan kontrolleras på vissa ställen i `run`-metoden, och beroende på detta värde kan olika operationer utföras. Variabelns värde kan anges från en annan tråd via metoden `setStatus`. Tråden bestämmer först variabeln, och sedan anropar metoden `interrupt` i samband med den tråd som innehåller statusvariabeln. Mottagartråden kan tolka `interrupt`-signalen på olika sätt beroende på statusvariabelns värde. På så vis kan måltråden kontrolleras. Först sätts trådens statusvariabel, och sedan anropas metoden `interrupt`. Genom att kontrollera statusvariabelns värde vet måltråden vad sändartråden vill meddela. Den tar emot och tolkar signalen, och följer eventuellt de anvisningar som ges av sändartråden.

En räknare och en kontrollör

Ett program kan innehålla en räknare, som räknar antalet enheter av en viss typ. Denna räknare kan när som helst återställas till `0`, så att den kan påbörja en ny räkning. Räknarens aktivitet kan också avbrytas när som helst. En kontrollör kan skapas, som kontrollerar räknarens aktivitet. Denna kontrollör kan skapa och starta en räknare, den kan återställa räknaren och avbryta räknarens aktivitet. Även kontrollören kan avbrytas utifrån.

 © FADIL GALJIĆ

Klassen `Raknare`, som definierar en tråd som simulerar en räknare, kan skapas. Klassen `Kontrollor`, som simulerar en kontrollörs aktivitet, kan också skapas. I klassen `Raknare` kan statusvariabeln `status` och metoden `setStatus` (som sätter statusvariabelns värde till ett givet heltalsvärde) införas. När `interrupt`-signalen kommer ska en räknartråd först läsa av statusvariabelns värde. Beroende på detta värde ska räknartråden antingen återställa sitt tillstånd till `0`, eller avsluta sin aktivitet. En kontrolltråd ska skapa och starta en räknartråd. Efter en tid ska kontrolltråden återställa räknaren, och låta den börja räkna på nytt. Denna aktivitet ska upprepas tills kontrolltråden får en `interrupt`-signal. Då ska kontrolltråden avsluta räknartrådens aktivitet, och sedan avsluta sin egen aktivitet. Kontrolltråden ska använda räknarens statusvariabel och `interrupt`-signaler för att kontrollera räknartråden.

Klassen `Raknare` kan definieras och implementeras så här:

```
class Raknare implements Runnable
{
    private int    antal = 0;
    private int    status = 0;

    public void run ()
    {
        int    v;

        while (true)
        {
            if (Thread.interrupted ())
            {
                System.out.println ();

                if (status == -1)
                    break;
                else
                    antal = status;
            }

            try
            {
                v = (int) (4 * Math.random () + 1);
                Thread.sleep (v * 500);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread ().interrupt ();
                continue;
            }

            antal++;
```

© FADIL GALJIĆ

```
            System.out.print (antal + " ");
        }
    }

    public void setStatus (int status)
    {
        this.status = status;
    }
}
```

I början av varje pass genom loopen kontrolleras om en `interrupt`-signal kommit. Om en sådan signal har kommit, kontrolleras statusvariabelns värde. Beroende på detta värde återställs räknaren eller så avslutas aktiviteten. `interrupt`-signalen kan inte missas om tråden är blockerad i `sleep`-metoden. I detta fall fångas och hanteras det undantag som kastas. Motsvarande flagga sätts, och denna analyseras sedan med metoden `interrupted`.

Klassen `Kontrollor` kan definieras och implementeras så här:

```
class Kontrollor implements Runnable
{
    private Raknare    raknare;
    private Thread     raknartrad;

    public void run ()
    {
        raknare = new Raknare ();
        raknartrad = new Thread (raknare);
        raknartrad.start ();

        while (true)
        {
            try
            {
                int    p = (int) (20 * Math.random () + 1);
                Thread.sleep (p * 1000);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread ().interrupt ();
            }

            if (Thread.interrupted ())
            {
                raknare.setStatus (-1);
                raknartrad.interrupt ();
                break;
            }
```


© FADIL GALJIĆ

```
                raknare.setStatus (0);
                raknartrad.interrupt ();
            }
        }
}
```

För att kunna kontrollera räknartråden, innehåller kontrolltråden både en referens till ett objekt av typen `Raknare` och en referens till motsvarande räknartråd (ett objekt av typen `Thread`). Metoden `setStatus` är en metod i klassen `Raknare`, inte i klassen `Thread`. Denna metod kan endast användas i samband med ett objekt av typen `Raknare`.

I slutet av varje pass genom loopen återställs räknaren. En `interrupt`-signal skickas till räknartråden, men först sätts räknarens statusvariabel till `0`. Räknartråden ska tolka detta värde som en begäran om återställning. När kontrolltråden själv tar emot en `interrupt`-signal, avbryter den först räknartråden och avslutar sedan sin aktivitet. För att avbryta räknartråden skickas en `interrupt`-signal till denna tråd. Men innan denna signal skickas, sätts räknarens statusvariabel till `-1`. Räknartråden ska tolka detta värde som en begäran om att avsluta sin aktivitet.

Flera `interrupt`-signaler kan skickas till räknartråden. Räknartråden kan tolka dessa signaler på olika sätt, beroende på statusvariabelns aktuella värde. Denna variabel gör det möjligt att utanför denna tråd välja en viss aktivitet inuti tråden.

För att testa klasserna `Raknare` och `Kontrollor` kan man skapa ett program, och en kontrolltråd i programmet. Man startar sedan kontrolltråden (som sedan skapar och startar sin räknare) och påbörjar på så sätt aktiviteten. Efter en viss tid skickas en `interrupt`-signal till tråden, och på så sätt avslutas den (före än kontrolltråden avslutas, avslutar den sin räknare). När programmet exekveras, kan följande utskrift skapas:

```
1 2 3 4 5 6
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2
```

I det här fallet återställs räknaren tre gånger innan aktiviteten avbryts.

Signaler till en grupp trådar

För att ett stort antal trådar lättare ska kunna manipuleras i ett program, kan dessa trådar grupperas i ett antal grupper. Det finns vissa operationer som kan utföras i samband med en grupp trådar. En `interrupt`-signal, till exempel, kan skickas till en sådan grupp. I detta fall får varje tråd i grup-

© FADIL GALJIĆ

pen denna signal. Istället för att en signal skickas till varje enskild tråd, behöver den bara skickas till en trådgrupp.

En grupp trådar representeras med ett objekt av klassen `java.lang.ThreadGroup`. Ett objekt av denna klass kan skapas så här:

```
ThreadGroup    tg = new ThreadGroup ("riktningar");
```

Här skapas trådgruppen `tg` med namnet `riktningar`. Enskilda trådar kan skapas och läggas till i denna grupp. För att placera en tråd i en grupp, anger man gruppens namn (som första argument till motsvarande konstruktor) när tråden skapas.

Klassen `Skrivare` definierar en tråd som skriver ut ett meddelande exakt fyra gånger. Detta meddelande anges när ett objekt av typen `Skrivare` skapas. Klassen kan definieras och implementeras så här:

```
class Skrivare implements Runnable
{
    private String    meddelande;

    public Skrivare (String meddelande)
    {
        this.meddelande = meddelande;
    }

    public void run ()
    {
        for (int i = 0; i < 4; i++)
        {
            System.out.println (meddelande);

            if (Thread.interrupted ())
                break;
        }
    }
}
```

Flera trådar av typen `Skrivare` kan skapas och placeras i trådgruppen `tg`:

```
Thread    t1 = new Thread (tg, new Skrivare ("öst"));
Thread    t2 = new Thread (tg, new Skrivare ("väst"));
Thread    t3 = new Thread (tg, new Skrivare ("syd"));
Thread    t4 = new Thread (tg, new Skrivare ("nord"));
```

Varje tråd skriver ut sitt meddelande exakt fyra gånger. Alla trådar tillhör trådgruppen `tg`.

De enskilda trådarna kan startas, och därefter kan en `interrupt`-signal skickas till trådgruppen `tg`:

© FADIL GALJIĆ

```
tg.interrupt ();
```

Denna signal vidarebefordras automatiskt till alla trådar i gruppen. På så vis kan samtliga trådar i gruppen avbrytas. Det kan hända att trådarna inte har hunnit skriva ut sina meddelanden när de avbryts. Utskriften kan i så fall se ut på följande sätt:

```
öst
väst
öst
väst
syd
nord
öst
```

© FADIL GALJIĆ

Synkroniserad användning av ett objekt

Osynkroniserad användning av ett objekt

En klass som representerar ett ensiffrigt heltal

Klassen `EnsiffrigtHeltal`, som representerar ett ensiffrigt heltal, kan skapas. Denna klass hanterar ett heltals numeriska värde och heltalets namn. Så här kan detta göras:

```
class EnsiffrigtHeltal
{
    private int       varde;
    private String    namn;

    public EnsiffrigtHeltal ()
    {
        varde = 0;
        namn = "noll";
    }

    public String toString ()
    {
        String    s = null;

        s = "[" + varde + ", " + namn + "]";
        return s;
    }

    public void set (int v)
    {
        this.varde = v;

        String    n = null;
        int    absv = Math.abs (v);
        switch (absv)
        {
        case 0:
            n = "noll";
            break;
        case 1:
            n = "ett";
            break;
        case 2:
            n = "två";
```


© FADIL GALJIĆ

```
                break;
            case 3:
                n = "tre";
                break;
            case 4:
                n = "fyra";
                break;
            case 5:
                n = "fem";
                break;
            case 6:
                n = "sex";
                break;
            case 7:
                n = "sju";
                break;
            case 8:
                n = "åtta";
                break;
            case 9:
                n = "nio";
                break;
            default:
                n = "inte ensiffrigt heltal";
                break;
            }

            if (v < 0)
               n = "minus " + n;

            this.namn = n;
    }
}
```

Klassen `EnsiffrigtHeltal` har två instansvariabler. Variabeln `varde` representerar heltalets numeriska värde och variabeln `namn` representerar heltalets namn. Klassens konstruktor initierar heltalets värde till `0` och heltalets namn till `noll`. Metoden `toString` skapar och returnerar en sträng som representerar heltalet. Strängen innehåller heltalets numeriska värde och motsvarande namn. Metoden `set` får ett heltalsvärde som argument, och heltalets numeriska värde och namn bestäms utifrån detta argument. Metoden analyserar heltalets numeriska värde, och bestämmer och anger heltalets namn utifrån detta.

Klassen `EnsiffrigtHeltal` kan användas på följande sätt i ett program:

```
EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();
n.set (5);
System.out.println (n);  // metoden toString anropas
n.set (-7);
```

© FADIL GALJIĆ

```
System.out.println (n);  // metoden toString anropas
```

Här skapas ett objekt som representerar ett ensiffrigt heltal, och detta objekt ändras två gånger. Först justeras objektet så att det representerar heltalet `5`. Sedan justeras objektet så att det representerar heltalet `-7`. Heltalet visas efter varje ändring. Följande utskrift skapas:

```
[5, fem]
[-7, minus sju]
```

Osynkroniserade ändringar av ett objekt

Ett objekt av typen `EnsiffrigtHeltal` kan användas i ett program med flera trådar. Ett objekt av denna typ kan skapas, och två (eller flera) trådar kan sedan använda objektet. Dessa trådar kan definieras så här:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 10; i++)
        {
            int    v = (int) (10 * Math.random ());
            heltal.set (v);

            String    s = heltal.toString ();
            System.out.println (s);
        }
    }
}

class Anvandare2 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare2 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
```


© FADIL GALJIĆ

```
        for (int i = 0; i < 10; i++)
        {
            int    v = (int) (-9 * Math.random () - 1);
            heltal.set (v);

            String    s = heltal.toString ();
            System.out.println (s);
        }
    }
}
```

Klassen `Anvandare1` definierar en tråd som har tillgång till ett objekt av typen `EnsiffrigtHeltal`. I metoden `run` ändras detta objekt ett antal gånger, men objektet representerar alltid ett heltal mellan `0` (inklusive) och `9` (inklusive). Objektet visas efter varje ändring. Klassen `Anvandare2` definierar en tråd på ett liknande sätt. En tråd av denna klass har ett objekt av typen `EnsiffrigtHeltal`, som representerar ett heltal mellan `-9` (inklusive) och `-1` (inklusive).

Ett objekt av typen `EnsiffrigtHeltal` och två trådar som använder objektet kan skapas på följande sätt:

```
class OsynkroniseradeAndringarAvEttObjekt
{
    public static void main (String[] args)
    {
        EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();

        Thread    t1 = new Thread (new Anvandare1 (n));
        Thread    t2 = new Thread (new Anvandare2 (n));

        t1.start ();
        t2.start ();
    }
}
```

Tråden `t1` har tillgång till objektet `n` (av typen `EnsiffrigtHeltal`), och exekverar den kod som finns i `run`-metoden i klassen `Anvandare1`. Den modifierar objektet `n` flera gånger, men så att objektet representerar ickenegativa ensiffriga heltal. Tråden `t2` har tillgång till samma objekt, och exekverar den koden som finns i `run`-metoden i klassen `Anvandare2`. Även denna tråd modifierar objektet `n`, men på så sätt att det representerar negativa ensiffriga heltal. Två trådar använder och modifierar ett och samma objekt samtidigt.

Om programmet `OsynkroniseradeAndringarAvEttObjekt` exekveras, kan utskriften se ut på följande vis:

```
[6, sex]
```

```
[9, nio]
[0, noll]
[7, sju]
[0, noll]
[-4, minus fyra]
[3, tre]
[-9, minus nio]
[9, nio]
[-8, minus åtta]
[7, sju]
[-3, minus tre]
[0, noll]
[-2, minus två]
[4, fyra]
[-8, minus åtta]
[-1, minus ett]
[-2, minus två]
[-3, minus tre]
[-4, minus fyra]
```

Varje tråd ändrar objektet `n` exakt tio gånger, och visar objektet efter varje ändring. Tråden `t1` modifierar heltalet så att det representerar olika icke-negativa värden, och tråden `t2` modifierar heltalet så att det representerar olika negativa värden. Varje tråd får tillfälle att exekvera och använda objektet `n`. Trådarna exekveras i en ordning som inte går att förutsäga.

En utskrift på formen `[6, sex]` är en passande utskrift. Ett heltals värde (`6`) överensstämmer med heltalets namn (`sex`). Det skapas en konsistent information. Men vid vissa exekveringar av programmet `OsynkroniseradeAndringarAvEttObjekt` kan helt opassande utskrifter skapas. Man kan till exempel få följande utskrift:

```
[-9, sex]
[1, ett]
[9, nio]
[2, två]
[5, fem]
[5, minus nio]
[5, fem]
[-2, minus två]
[9, nio]
[-1, minus ett]
[2, två]
[-1, minus ett]
[8, åtta]
[-3, minus tre]
[2, två]
[-6, minus sex]
[-5, minus fem]
[-1, minus ett]
```


© FADIL GALJIĆ

```
[-4, minus fyra]
[-9, minus nio]
```

När objektet `n` skrivs ut första gången blir utskriften `[-9, sex]`. I detta fall stämmer inte heltalets värde (`-9`) överens med heltalets namn (`sex`). Objektet `n` har hamnat i ett inkonsistent tillstånd. Detta är en typisk situation som kan uppstå när flera trådar samtidigt använder ett och samma objekt.

Metoden `set` innehåller flera typiska steg. Först sätts objektets numeriska värde utifrån det värde som anges som argument. Därefter bestäms argumentvärdets absolutbelopp. Sedan bestäms heltalets grundnamn utifrån detta absolutbelopp. Detta grundnamn kompletteras med ordet `minus` i fall av ett negativt heltal. Slutligen sätts objektets namn.

Objektets numeriska värde fastställs i första satsen i metoden `set`, och objektets namn i den sista satsen i denna metod. Mellan dessa två satser finns kod som bestämmer objektets namn. Det som kan inträffa är att en tråd som ändrar objektet avbryts någonstans mellan den första och den sista satsen. Den andra tråden kan då få tillfälle att exekvera. Denna tråd fortsätter från den punkt där den senast avbröts. Detta kan vara på någon plats mellan den första och den sista satsen. Tråden kan nå fram till denna punkt i metoden `set` där objektets namn bestäms. I detta fall sätter en tråd objektets numeriska värde, och den andra tråden objektets namn. Objektet hamnar på så sätt i ett inkonsistent tillstånd, där objektets namn inte överensstämmer med objektets numeriska värde.

Tråden `t1` kan sätta objektets numeriska värde till `6`, och tillfälligt avbrytas efter det. Tråden `t2` kan få tillfälle att exekvera, och sätta objektets numeriska värde till `-9` (värdet ändras från `6` till `-9`). Därefter kan tråden `t2` tillfälligt avbrytas, och tråden `t1` får tillfälle att exekvera. Tråden `t1` fortsätter, och sätter objektets namn till `sex` (eftersom metoden `set` anropas med argumentet `6` i denna tråd – variabeln `v` i tråden är `6`). Objektets numeriska värde blir därmed `-9`, och objektets namn blir `sex`. Objektet `n` är i ett inkonsistent tillstånd. Tråden `t1` kan sedan skriva ut objektet `n`, och fortsätta exekvera. Den kan ändra och skriva ut heltalet flera gånger. När den ändrar heltalet till `5`, kan tråden `t2` få tillfälle att exekvera. Denna tråd fortsätter med att bestämma objektets namn (den ändrar inte heltalets numeriska värde). Namnet bestäms utifrån det värdet som tråden senast skickade till metoden `set`, och detta värde är `-9` (variabeln `v` i denna tråd är `-9`). Objektets numeriska värde förblir därmed `5`, och dess namn blir `minus nio`. Tråden `t2` skriver ut objektet, vilket ger utskriften `[5, minus nio]`. Objektet `n` är åter i ett inkonsistent tillstånd.

Man ska observera att varje tråd har sin egen (lokala) variabel `v`, som skapas före metoden `set` anropas. Under exekveringen av metoden `set` skapas även variabeln `n` och variabeln `absv`. Dessa variabler är lokala variabler i metoden `set`, som skapas och används av den tråd som exekverar metoden. Varje tråd har sina egna variabler `n` och `absv`. Dessa variabler försvinner när en tråd avslutar exekveringen av metoden `set`, och skapas åter när tråden på nytt kommer in i denna metod. Varje tråd har sina egna lokala variabler, och dessa kan inte förväxlas. Den ena tråden har inte tillgång till den andra trådens lokala variabler. Varje tråd använder sina egna variabler `v`, `n` och `absv` för att bestämma objektets numeriska värde och namn. En tråds lokala variabler kan inte användas av en annan tråd som får tillfälle att exekvera.

Å ena sidan har trådarna `t1` och `t2` sina egna variabler `v`, `n` och `absv`. Å andra sidan använder de båda trådarna ett och samma objekt. De använder objektet `n` och dess variabler `varde` och `namn` (variablerna används via metoderna `set` och `toString`). Som ett resultat av en osynkroniserad användning av objektet kan detta hamna i ett inkonsistent tillstånd.

Två trådar kan använda två helt skilda objekt. Detta inträffar till exempel om trådarna skapas så här:

```
EnsiffrigtHeltal    n1 = new EnsiffrigtHeltal ();
EnsiffrigtHeltal    n2 = new EnsiffrigtHeltal ();

Thread    t1 = new Thread (new Anvandare1 (n1));
Thread    t2 = new Thread (new Anvandare2 (n2));
```

Trådarna `t1` och `t2` har i detta fall ingen kontaktpunkt. Varje tråd har sitt eget objekt, och dessa objekt kan inte hamna i ett inkonsistent tillstånd. Det händer dock ofta att flera trådar kämpar om en och samma resurs (en fysisk resurs, en variabel eller ett objekt), eller att flera trådar samarbetar på något sätt i samband med en viss resurs. I sådana fall måste användningen av resursen synkroniseras.

Osynkroniserade ändringar och avläsningar

Om flera trådar ändrar ett och samma objekt på ett osynkroniserat sätt, kan detta objekt hamna i ett otillåtet tillstånd. Ett liknande problem kan uppstå även när en tråd ändrar ett objekt och en annan tråd avläser objektets tillstånd. Lästråden kan få tillfälle att exekvera när det gemensamma objektet endast är delvis ändrat. Objektet kan tillfälligt vara i ett inkonsistent tillstånd, och lästråden kan se detta tillstånd. En tråd kan avlä-

© FADIL GALJIĆ

sa ett objekts tillstånd när objektet är mitt i ändringen. På så sätt kan man få en motsägelsefull information om objektet.

I programmet `OsynkroniseradeAndringarAvEttObjekt` finns det två trådar, som båda ändrar ett objekt av typen `EnsiffrigtHeltal` och avläser tillståndet hos detta objekt. Problem kan i detta fall uppstå både på grund av osynkroniserade ändringar, och på grund av att avläsningarna inte är synkroniserade med motsvarande ändringar. För att kunna undersöka de problem som uppstår när inte avläsningarna och ändringarna är synkroniserade, kan man skapa en tråd som bara ändrar ett objekt, och en annan tråd som bara avläser objektets tillstånd. Dessa trådar kan definieras så här:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 5; i++)
        {
            int    v = (int) (10 * Math.random ());
            heltal.set (v);
        }
    }
}

class Anvandare2 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare2 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 5; i++)
        {
            String    s = heltal.toString ();
            System.out.println (s);
        }
    }
}
```

Ett objekt av typen `EnsiffrigtHeltal` kan skapas, och två trådar (en tråd av typen `Anvandare1` och en tråd av typen `Anvandare2`) som använder detta objekt:

```
class OsynkroniseradeAndringarOchAvlasningar
{
    public static void main (String[] args)
    {
        EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();

        Thread    t1 = new Thread (new Anvandare1 (n));
        Thread    t2 = new Thread (new Anvandare2 (n));

        t1.start ();
        t2.start ();
    }
}
```

Tråden `t1` är en skrivtråd (skriver till objektet), som ändrar objektet `n`. Tråden `t2` är en lästråd, som avläser (via metoden `toString`) objektets tillstånd. Beroende av när dessa två trådar får tillfälle att exekvera, kan olika utskrifter skapas. Man kan få till exempel följande utskrift:

```
[0, noll]
[0, noll]
[0, noll]
[0, noll]
[0, noll]
[5, fem]
[5, fem]
[5, fem]
[5, fem]
[5, fem]
```

Lästråden får tillfälle att exekveras när objektet `n` innehåller värdet `0`, och sedan när objektet innehåller värdet `5`. I båda fallen är objektet i ett konsistent tillstånd (objektets namn överensstämmer med dess numeriska värde). Men utskriften kan också vara motsägelsefull, som i följande fall:

```
[3, ett]
[3, ett]
[3, ett]
[3, ett]
[3, ett]
[0, noll]
[0, noll]
[0, noll]
[0, noll]
[0, noll]
```

 © FADIL GALJIĆ

I detta fall får lästråden tillfälle att exekvera när skrivtråden ändrat objektets numeriska värde till `3`, men ännu inte hunnit anpassa objektets namn, som fortfarande är `ett`. Lästråden fick se objektet när den inte borde ha det. Därför uppkommer en motsägelsefull information.

Helt synkroniserade objekt

Samtidiga användningar av en och samma resurs

När flera trådar använder ett objekt samtidigt, kan detta objekt hamna i ett inkonsistent tillstånd. Om flera trådar samtidigt ändrar ett objekt, kan det inträffa att dessa ändringar inte passar ihop med varandra. Kanske har en tråd endast hunnit ändra en del av objektets variabler, när en annan tråd får tillfälle att fortsätta exekvera. Den andra tråden kan ändra vissa av objektets variabler och sedan avsluta sin exekvering. Dessa kombinerade ändringar kan vara motsägelsefulla. Det kan hända att vissa variablers värden inte stämmer överens med andra variablers värden i objektet. Samma situation kan uppstå när flera trådar samtidigt använder en fysisk resurs, till exempel en fil. När flera trådar samtidigt skriver till en och samma fil, kan filens innehåll till slut bli meningslöst.

Förutom dessa problem med samtidiga ändringar av ett och samma objekt, kan olika problem även uppstå när en tråd ändrar ett objekt och en annan tråd läser olika uppgifter från objektet. Det kan hända att en tråd bara hinner ändra en del av ett objekts variabler, när en annan tråd får tillfälle att exekvera. Tråden får tillgång till objektet när det är i ett inkonsistent tillstånd, och hämtar därför en motsägelsefull information. Användningen av denna information kan leda till oönskade konsekvenser. Samma situation kan uppstå när en tråd ändrar en fysisk resurs, och en annan tråd samtidigt läser olika informationer från denna resurs. Det kan hända till exempel att en tråd endast hinner skriva en del av en information till en fil, när en annan tråd får tillfälle att exekvera. Den andra tråden kan avläsa filens innehåll, och få en ofullständig (kanske motsägelsefull) information.

En tråd kan avbrytas när som helst. Det kan även hända mitt i en källkodsinstruktion. En källkodsinstruktion översätts till flera lågnivåinstruktioner, och exekveringen kan avbrytas efter att en av dessa lågnivåinstruktioner har utförts. En tråd kan till exempel exekvera följande källkodsinstruktion:

```
m = n + 2;
```

Den exekverande tråden kan avbrytas efter det att den hämtat värdet på variabeln `n`. Tråden hinner inte utföra additionen och ändra variabeln `m`. En annan tråd kan få tillfälle att exekvera, och denna tråd kan ändra variabeln `n`. Den första tråden kan sedan fortsätta, och slutföra ändringen av variabeln `m` (utifrån det värde på variabeln `n` som tråden hämtat tidigare). Men denna ändring avspeglar inte den situation som gäller när ändringen utförs. Värdet på variabeln `m` motsvarar inte det aktuella värdet på variabeln `n`.

Denna situation, där flera trådar samtidigt använder en och samma resurs, kan normalt uppstå i olika program. Det kan hända att flera trådar kämpar om en och samma resurs, eftersom det inte finns flera sådana resurser. Det kan också inträffa att flera trådar samarbetar för att modifiera en komplicerad resurs. Det kan även finnas situationer då en tråd modifierar en resurs och en annan tråd läser olika informationer från denna resurs. I alla dessa situationer måste de olika trådarnas aktiviteter på något sätt samordnas. De olika trådarna som använder en och samma resurs måste synkroniseras.

En synkroniserad metod

En ny typ av objekt definieras i en särskild klass, en så kallad definitionsklass. Redan när man skapar en sådan klass kan man fundera över hur objekt av denna typ kan användas i samband med flera trådar. En definitionsklass kan konstrueras på ett sådant sätt att ett objekt av klassen inte kan hamna i ett inkonsistent tillstånd. Ett typiskt element i en sådan klass är en *synkroniserad metod*.

En metod blir en synkroniserad metod genom att den deklareras som `synchronized`. Metoden `set` i klassen `EnsiffrigtHeltal`, till exempel, kan göras till en synkroniserad metod:

```
public synchronized void set (int v)
{
    // kod som tidigare
}
```

Om en metod deklareras som `synchronized`, kan bara en tråd exekvera metoden i samband med ett objekt vid ett visst tillfälle. Endast efter det att tråden har avslutat sin exekvering av metoden (antingen normalt eller genom att kasta ett undantag), kan någon annan tråd få tillfälle att exekvera metoden i samband med det aktuella objektet. På så vis förhindras att flera trådar samtidigt kommer åt ett och samma objekt via denna metod. Om metoden på något sätt ändrar ett objekt, förhindras att flera

© FADIL GALJIĆ

trådar samtidigt ändrar objektet. Först när en tråd har slutfört ändringen, kan en annan tråd få möjlighet att ändra objektet. På så vis förhindras att ett objekt hamnar i ett inkonsistent tillstånd.

Om metoden `set` i klassen `EnsiffrigtHeltal` deklareras som `synchronized`, kan bara en tråd modifiera ett objekt av denna klass vid ett visst tillfälle. Bara när tråden anger både objektets numeriska värde och dess namn (när metoden `set` avslutas), kan en annan tråd få chans att exekvera metoden `set` i samband med objektet. Det kan inte inträffa att en tråd bestämmer ett objekts numeriska värde och en annan tråd dess namn. Ett objekt av typen `EnsiffrigtHeltal` kan inte hamna i ett tillstånd där dess namn inte överensstämmer med dess numeriska värde.

Flera trådar kan inte samtidigt exekvera en synkroniserad metod i samband med ett och samma objekt. Det finns dock inget som hindrar att en sådan metod samtidigt exekveras i samband med olika objekt. En tråd kan exekvera metoden `set` i samband med ett objekt av typen `EnsiffrigtHeltal`, och en annan tråd kan samtidigt exekvera samma metod i samband med ett annat objekt av den typen.

När en tråd exekverar en synkroniserad metod i samband med ett objekt, kan inte någon annan tråd samtidigt exekvera denna metod i samband med samma objekt. Det finns även en strängare restriktion: flera synkroniserade metoder i en klass kan inte samtidigt exekveras i samband med ett och samma objekt av denna klass. Om till exempel både metoden `toString` och metoden `set` i klassen `EnsiffrigtHeltal` deklareras som `synchronized`, kan inte dessa två metoder exekveras samtidigt i samband med ett objekt av typen `EnsiffrigtHeltal`. Det går bara att exekvera en synkroniserad metod åt gången i samband med ett objekt. En synkroniserad metod och en metod som inte är synkroniserad kan dock exekveras samtidigt i samband med ett och samma objekt.

Endast en tråd kan exekvera en synkroniserad metod i samband med ett objekt vid ett visst tillfälle. När denna metod avslutas, kan en annan tråd få tillfälle att exekvera metoden (eller någon annan synkroniserad metod i samma klass) i samband med detta objekt. Detta innebär dock inte att den tråd som exekverar en synkroniserad metod behöver utföra exekveringen utan avbrott. Tråden kan avbrytas när som helst, och andra trådar kan då få möjlighet att exekvera. Dessa trådar kan utföra olika aktiviteter, men de kan inte exekvera en synkroniserad metod i samband med samma objekt. Om de vill göra det, måste de vänta. De kan få möjlighet att fortsätta denna exekvering först när exekveringen av den synkroniserade metoden har avslutats.

© FADIL GALJIĆ

Varje objekt i Java har ett inbyggt *lås*. Detta lås är en variabel, vars värde avgör om motsvarande objekt är låst eller inte. En tråd använder detta lås för att erhålla exklusiv rätt att exekvera en synkroniserad metod i samband med ett objekt.

När en tråd anropar en synkroniserad metod i samband med ett objekt, kontrolleras automatiskt om objektet är låst. Om objektet inte är låst, får tråden objektets lås (den motsvarande variabeln ändras så att objektet låses) och kan exekvera den anropade metoden. Låset behålls även om tråden avbryts tillfälligt. När tråden åter får möjlighet att exekvera, fortsätter exekveringen från den punkt där den avbröts. Först när tråden avslutat exekveringen av metoden, släpper den objektets lås (den motsvarande variabeln antar det värde som indikerar att objektet inte är låst). Objektet låses upp, och en annan tråd har möjlighet att exekvera en synkroniserad metod i samband med objektet.

En tråd kan exekvera en synkroniserad metod som anropar en annan synkroniserad metod i samma klass. I så fall går den tråd som exekverar metoden in i den metod som anropas, exekverar denna metod, och återgår utan att förlora motsvarande lås. Tråden släpper låset när den avslutar exekveringen av den anropande metoden. Om tråden åter anropar en synkroniserad metod i samband med samma objekt, måste den på nytt få objektets lås för att kunna exekvera den anropade metoden. Om en tråd exekverar en synkroniserad metod som anropar en synkroniserad metod i samband med ett annat objekt (av samma typ eller av en annan typ), måste tråden även få det andra objektets lås för att kunna fortsätta exekveringen. En och samma tråd kan hålla flera lås vid ett visst tillfälle.

En konstruktor i en klass kan inte deklareras som `synchronized`. Det är användningen av redan existerande (konstruerade) objekt som synkroniseras. Flera trådar kan inte använda ett objekt som skapas. En tråd måste först skapa ett objekt och returnera en referens till det, och sedan kan andra trådar använda objektet. Om en konstruktor använder ett redan konstruerat objekt, kan användningen av objektet synkroniseras inuti konstruktorn (det finns tekniker som gör det möjligt att synkronisera exekveringen av ett kodavsnitt).

Även en statisk metod kan deklareras som `synchronized`. Det kan inträffa att flera statiska metoder manipulerar statiska variabler i en klass. För att förhindra motsägelsefulla operationer ska dessa metoder deklareras som `synchronized`. En statisk metod i en klass kan bara exekveras av en tråd vid ett visst tillfälle. En statisk metod anropas inte i samband med något objekt, och därför kan inte det aktiverande objektets lås användas (det finns inget aktiverande objekt). Ett annat objekt måste utnyttjas för att

 © FADIL GALJIĆ

åstadkomma synkronisering av statiska metoder. I ett program som använder en klass, skapas automatiskt ett objekt som representerar klassen. Om man till exempel använder klassen `X`, skapas automatiskt ett objekt som representerar denna klass. Detta objekt kan nås via referensen `X.class`. Man använder objektets lås för att synkronisera exekveringen av statiska metoder i klassen `X`.

Helt synkroniserade objekt

När en tråd exekverar en synkroniserad metod i samband med ett visst objekt, kan samtidigt en annan tråd exekvera en metod som inte är synkroniserad i samband med detta objekt. Denna samtidiga användning av ett och samma objekt kan ge oönskade resultat. Den ena tråden kan utföra vissa ändringar i det gemensamma objektet, och den andra tråden kan utföra andra ändringar i samma objekt. Som en konsekvens av dessa osynkroniserade ändringar kan ett objekt hamna i ett inkonsistent tillstånd. En annan risk uppstår när den ena tråden läser olika informationer från det gemensamma objektet, medan den andra tråden ändrar det objektet. I så fall kan lästråden få tillgång till det gemensamma objektet mitt i ändringen, när objektet befinner sig i ett inkonsistent tillstånd. Den erhållna informationen kan på detta vis bli inkorrekt.

För att förhindra osynkroniserad användning av ett objekt, kan objektet låsas helt när en tråd exekverar en metod i samband med objektet. Detta kan åstadkommas genom att samtliga metoder i objektets klass deklareras som `synchronized`. Då kan endast en metod (av de metoder som finns i definitionsklassen) åt gången exekveras i samband med ett objekt av klassen. När denna metod utfört hela sin uppgift, kan andra metoder få tillfälle att exekveras i samband med objektet. På så vis förhindras oönskade interferenser mellan olika trådar via olika metoder.

En konstruktor ska initiera ett objekt så att det blir konsistent. En synkroniserad metod som ändrar objektet ska inte lämna detta i ett inkonsistent tillstånd. Oavsett om metoden avslutas normalt eller genom ett undantag, måste objektet lämnas i ett konsistent tillstånd. Ett objekt kan vara tillfälligt inkonsistent inuti metoden, men det måste vara konsistent när metoden avslutas.

En klass som innehåller synkroniserade metoder ska inte ha publika instansvariabler. Om en klass har en publik variabel, kan variabelns värde påverkas direkt, oavsett vilken metod i klassen som exekveras vid ett visst tillfälle. En osynkroniserad användning av ett objekts variabler via olika metoder och via direkt åtkomst kan leda till oönskade konsekvenser.

© FADIL GALJIĆ

En synkroniserad metod måste avslutas efter en viss tid, så att andra trådar får möjlighet att använda det aktuella objektet. En sådan metod kan inte innehålla en oändlig loop. En tråd ska inte kunna hålla ett objekts lås för alltid. Andra trådar måste få möjlighet att använda samma objekt.

Man ska alltså skapa konsistenta objekt, och utesluta möjligheten av samtidig användning av ett objekt genom att deklarera samtliga metoder som `synchronized` och genom att deklarera instansvariabler som `private`. En metod ska vara ändlig i tiden, och den ska inte lämna ett objekt i ett inkonsistent tillstånd. Om dessa regler respekteras, skapas en klass som definierar *helt synkroniserade objekt*. Ett objekt av en sådan klass kan inte hamna i ett inkonsistent tillstånd när det används av flera trådar. Objektet kan endast tillfälligt hamna i ett inkonsistent tillstånd, inuti en metod som modifierar det (objektet kan dock inte kommas åt så länge modifieringsprocessen pågår). När metoden avslutas är dock objektet åter konsistent (och kan användas).

Samtliga metoder i klassen `EnsiffrigtHeltal` ska deklareras som `synchronized`. På så vis blir objekt av denna klass säkra i ett program med flera trådar. Klassens variabler är privata, och klassens konstruktor initierar ett objekt på så sätt att dess namn stämmer överens med dess numeriska värden. Klassens metoder är synkroniserade, de är ändliga i tiden och de lämnar det aktuella objektet i ett konsistent tillstånd. Detta innebär att objekt av klassen `EnsiffrigtHeltal` är helt synkroniserade. Det finns ingen risk för en utskrift som har formen `[-9, sex]`, där objektets namn inte överensstämmer med dess numeriska värde. När en tråd använder ett objekt av typen `EnsiffrigHeltal`, kan inte någon annan tråd använda samma objekt. Den andra tråden måste vänta tills objektets lås släpps.

Sammansatta synkroniserade operationer

Osynkroniserade operationer på ett objekt

Det går att skapa en klass som definierar helt synkroniserade objekt. Varje operation (metod) i en sådan klass är en synkroniserad operation (metod). Medan en tråd utför en operation i samband med ett objekt av klassen, kan inte andra trådar använda objektet. Objektet blir tillgängligt först när operationen avslutats. Operationen lämnar objektet i ett konsistent tillstånd, och en annan operation kan utföras i samband med objektet.

I vissa fall kan det finnas behov av att utföra två eller flera operationer i samband med ett objekt, innan objektet släpps. En enskild operation kan

© FADIL GALJIĆ

vara endast ett steg i en mer komplicerad hantering av ett visst objekt. För att kunna skapa sammansatta synkroniserade operationer räcker det inte med helt synkroniserade objekt. Det behövs ytterligare synkroniseringstekniker.

Klassen `EnsiffrigtHeltal` definierar helt synkroniserade objekt. Ett objekt av denna klass kan inte hamna i ett inkonsistent tillstånd. Ett objekt kan inte ha ett namn som inte stämmer överens med objektets numeriska värde. Det kan inte hända att man får ett heltal av formen `[-9, sex]`, eller liknande. Trots detta kan vissa problem uppstå när ett objekt av klassen `EnsiffrigtHeltal` används i samband med flera trådar. Man kan vilja utföra flera operationer (anropa flera metoder) i samband med ett objekt av klassen, innan objektet släpps. Men ett helt synkroniserat objekt tillhandahåller inte möjligheten att utsträcka synkroniseringen över flera operationer. Ett sådant objekt synkroniserar bara enskilda operationer, inte uppsättningar av operationer.

Man kan skapa en tråd som ändrar ett objekt av typen `EnsiffrigtHeltal` tio gånger, och visar objektet efter varje ändring. Tråden ändrar objektet så att dess numeriska värde blir mellan `0` och `9`. En tråd till kan skapas, som använder samma objekt, på samma sätt. Den enda skillnaden är att den andra tråden ger objektet numeriska värden mellan `-9` och `-1`. När de två trådarna startas, kan utskriften bli så här:

```
[-7, minus sju]
[-3, minus tre]
[-3, minus tre]
[0, noll]
[-4, minus fyra]
[7, sju]
[-5, minus fem]
[4, fyra]
[-2, minus två]
[0, noll]
[-3, minus tre]
[0, noll]
[-9, minus nio]
[0, noll]
[-3, minus tre]
[9, nio]
[-3, minus tre]
[5, fem]
[7, sju]
[1, ett]
```

Varje tråd får möjlighet att ändra och visa objektet exakt tio gånger. Utskriften kan dock även få en annan form, till exempel den följande:

```
[7, sju]
[4, fyra]
[4, fyra]
[2, två]
[2, två]
[4, fyra]
[4, fyra]
[1, ett]
[1, ett]
[6, sex]
[6, sex]
[7, sju]
[7, sju]
[2, två]
[2, två]
[9, nio]
[9, nio]
[6, sex]
[6, sex]
[-4, minus fyra]
```

Objektet som ändras och visas är hela tiden i ett konsistent tillstånd, eftersom objektet är helt synkroniserat. Men det verkar som om huvudsakligen endast en tråd exekveras, nämligen den tråd som sätter objektets numeriska värde till icke-negativa heltal. Den andra tråden förefaller på något sätt vara undertryckt. Detta är dock inte fallet.

Varje tråd ändrar och visar det aktuella objektet exakt tio gånger. Men det finns inga garantier för att en tråd kommer att visa objektet omedelbart efter att den har ändrat objektet. Till exempel kan den andra tråden hinna ändra (men inte visa) objektet först. I denna stund kan den första tråden få tillfälle att exekvera. Denna tråd kan ändra och visa objektet. Därefter kan den andra tråden få tillfälle att exekvera. Tråden fortsätter sin aktivitet, visar först objektet (efter att den första tråden ändrat det) och ändrar det igen. Trådarna kan fortsätta att exekvera växelvis på detta sätt. I detta fall visar båda trådarna objektet efter det att den första tråden ändrat det. Den andra tråden visar inte sin egen ändring, utan den första trådens ändring. Det är bara på slutet som den andra tråden visar sin egen ändring.

Båda trådarna använder metoderna `set` och `toString` i klassen `EnsiffrigtHeltal`. Dessa två metoder är synkroniserade, och objektet kan inte hamna i ett inkonsistent tillstånd. Detta är dock inte tillräckligt i det här fallet. Ändringarna av ett objekt ska synkroniseras med utskrifterna av objektet. En tråd ska kunna ändra ett objekt och skriva ut det, innan en annan tråd får möjlighet att använda samma objekt. Objektets lås ska

© FADIL GALJIĆ

hållas längre, inte bara under en enskild operation, utan under flera operationer. Man vill kunna skapa en sammansatt synkroniserad operation.

En sammansatt synkroniserad operation

Man kan skapa en sammansatt synkroniserad operation utifrån flera enskilda (synkroniserade eller icke-synkroniserade) operationer. Ett block med dessa operationer bildas, och blocket definieras som en synkroniserad enhet. Det objekt vars lås ska hållas under operationerna specificeras också.

För att säkerställa att en tråd både ändrar ett objekt av typen `EnsiffrigtHeltal` och visar den utförda ändringen, ska ändringen och utskriften förenas i en sammansatt synkroniserad operation. Detta kan inte ske i klassen `EnsiffrigtHeltal`, utan görs istället på klientsidan (på den plats där ett objekt av klassen används), när tråden definieras. Man kan göra detta så här:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 10; i++)
        {
            synchronized (heltal)
            {
                int    v = (int) (10 * Math.random ());
                heltal.set (v);

                String    s = heltal.toString ();
                System.out.println (s);
            }
        }
    }
}
```

Klassen `Anvandare1` definierar en tråd som använder ett ensiffrigt heltal (av typen `EnsiffrigtHeltal`). Tråden ändrar detta heltal och visar det exakt tio gånger. Både ändringen av heltalet och dess utskrift placeras i ett *synkroniserat block*. På så vis bildas en sammansatt synkroniserad opera-

tion. Det använda heltalet ska låsas under operationen. Vilket objekt som helst kan användas efter nyckelordet `synchronized`, men normalt låses det använda objektet. Då kan inte någon annan tråd få tillgång till objektets lås medan operationerna pågår. När tråden utfört hela det synkroniserade blocket låses objektet upp, och först då kan en annan tråd få tillgång till objektets lås.

Det går att definiera en tråd till på ett liknande sätt. I så fall visar varje tråd sin egen ändring. En ändring synkroniseras med motsvarande utskrift, vilket innebär att en tråd inte kan visa en ändring som utförts av en annan tråd. I trådarnas definitionsklasser bestämmer man vilka operationer som ska utföras som en odelbar sammansatt operation.

Optimera synkroniseringen

Negativa följder av synkroniseringen

När en tråd exekverar en synkroniserad metod eller ett synkroniserat block, håller tråden objektets lås. Ingen annan tråd kan anropa objektets synkroniserade metoder under denna tid. Det går inte heller att komma in i ett synkroniserat block som använder objektets lås. Det är först när den synkroniserade metoden eller det synkroniserade blocket avslutats som en annan tråd kan få tillgång till objektets lås, och anropa en synkroniserad metod eller utföra ett synkroniserat block i samband med objektet. På så vis förhindras oönskade interferenser mellan olika trådar, och objektet bevaras i ett konsistent tillstånd.

Synkroniseringen har även vissa negativa följder. Att utföra en synkroniserad metod eller ett synkroniserat block tar lite mer tid och resurser, än det tar att utföra en osynkroniserad metod eller ett osynkroniserat block. Det måste ske en kontroll av om ett objekt är låst eller inte, ett objekt måste låsas, och senare låsas upp. Allt detta tar tid och resurser. Medan en tråd utför en synkroniserad metod eller ett synkroniserat block i samband med ett objekt, kan inte andra trådar använda objektets synkroniserade metoder eller exekvera de block som är synkroniserade på objektet. De måste vänta, och detta minskar programmets hastighet. Nyckelordet `synchronized` bör därför användas på ett försiktigt och genomtänkt sätt. Synkroniseringen måste optimeras på något sätt.

 © FADIL GALJIĆ

Optimera synkroniseringen i en klass som definierar en tråd

Klassen `Anvandare1` definierar en tråd som använder ett objekt av typen `EnsiffrigtHeltal`. I denna klass kan ett synkroniserat block skapas, som ändrar objektet och därefter visar det. Ändringen och utskriften kan på detta sätt utföras som en odelbar operation:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 10; i++)
        {
            synchronized (heltal)
            {
                int    v = (int) (10 * Math.random ());
                heltal.set (v);

                String    s = heltal.toString ();
                System.out.println (s);
            }
        }
    }
}
```

Ett heltal ändras och skrivs ut i ett synkroniserat block. Man förenar två relaterade operationer, men inte alla ändringar och utskrifter. Hela loopen placeras inte i det synkroniserade blocket. Man gör inte så här:

```
public void run ()
{
    synchronized (heltal)    // inte så här
    {
        for (int i = 0; i < 10; i++)
        {
            int    v = (int) (10 * Math.random ());
            heltal.set (v);

            String    s = heltal.toString ();
            System.out.println (s);
        }
    }
}
```

© FADIL GALJIĆ

Om hela loopen placeras i ett synkroniserat block, kommer en tråd att hålla låset till objektet `heltal` så länge loopen exekveras. Detta innebär att andra trådar måste vänta på låset under en relativt lång tid. Det är först när en tråd utfört alla sina ändringar och utskrifter, som en annan tråd kan få tillgång till objektets lås. Om det inte krävs att alla ändringar och utskrifter utförs som en odelbar (atomär) operation, ska det synkroniserade blocket placeras inuti loopen. Detta låser det motsvarande objektet under en betydligt kortare tid. Endast en ändring och motsvarande utskrift utförs som en odelbar operation.

Synkroniseringen av klassen `Anvandare1` kan optimeras ytterligare. Endast de operationer som använder ett gemensamt objekt behöver placeras i det synkroniserade blocket. Alla andra operationer kan lämnas utanför blocket, så här:

```
public void run ()
{
    for (int i = 0; i < 10; i++)
    {
        int    v = (int) (10 * Math.random ());
        String   s = null;

        synchronized (heltal)
        {
            heltal.set (v);
            s = heltal.toString ();
        }

        System.out.println (s);
    }
}
```

I det här fallet placeras endast två metodanrop i det synkroniserade blocket. De motsvarande satserna bildar en *kritisk sektion*. Det är bara där som det motsvarande objektet används, och det är då som objektet behöver skyddas. Variablerna `v` och `s` är lokala variabler i `run`-metoden, och kan inte förväxlas mellan olika trådar. Varje tråd har sina egna lokala variabler, och dessa kan hanteras utanför det synkroniserade blocket. Genom att minimera antalet operationer som utförs i det synkroniserade blocket, minimeras den tid under vilken det motsvarande objektet är låst.

Optimera synkroniseringen i en definitionsklass

Klassen `EnsiffrigtHeltal` är en definitionsklass, som hanterar ett ensiffrigt heltal. Klassen hanterar ett heltals numeriska värde och namn. För att

© FADIL GALJIĆ

bevara ett objekt av klassen i ett konsistent tillstånd, kan samtliga metoder i klassen definieras som `synchronized`:

```
class EnsiffrigtHeltal
{
    private int       varde;
    private String    namn;

    public EnsiffrigtHeltal ()
    {
        varde = 0;
        namn = "noll";
    }

    public synchronized String toString ()
    {
        String    s = null;

        s = "[" + varde + ", " + namn + "]";

        return s;
    }

    public synchronized void set (int v)
    {
        this.varde = v;

        String    n = null;
        int    absv = Math.abs (v);
        switch (absv)
        {
        case 0:
            n = "noll";
            break;
        case 1:
            n = "ett";
            break;
        case 2:
            n = "två";
            break;
        case 3:
            n = "tre";
            break;
        case 4:
            n = "fyra";
            break;
        case 5:
            n = "fem";
            break;
        case 6:
            n = "sex";
```

© FADIL GALJIĆ

```
            break;
        case 7:
            n = "sju";
            break;
        case 8:
            n = "åtta";
            break;
        case 9:
            n = "nio";
            break;
        default:
            n = "inte ensiffrigt heltal";
            break;
        }

        if (v < 0)
            n = "minus " + n;

        this.namn = n;
    }
}
```

När man anropar en synkroniserad metod låses det aktiverande objektet, och förblir låst under exekveringen av metoden. Men man behöver inte låsa objektet under hela den tiden. Synkroniseringen kan optimeras genom att kritiska sektioner i metoden separeras, och genom att endast dessa sektioner synkroniseras. Det kritiska i metoden `set` är de instruktioner som bestämmer objektets värde och namn. Resten av metoden består av en beräkning med lokala variabler, och denna kod är inte kritisk på något vis. De instruktioner som anger objektets värde och namn kan placeras i ett synkroniserat block, och resten av metoden kan placeras utanför blocket:

```
public void set (int v)
{
    String    n = null;
    int    absv = Math.abs (v);
    switch (absv)
    {
    case 0:
        n = "noll";
        break;
    case 1:
        n = "ett";
        break;
    case 2:
        n = "två";
        break;
    case 3:
```

 © FADIL GALJIĆ

```
            n = "tre";
            break;
        case 4:
            n = "fyra";
            break;
        case 5:
            n = "fem";
            break;
        case 6:
            n = "sex";
            break;
        case 7:
            n = "sju";
            break;
        case 8:
            n = "åtta";
            break;
        case 9:
            n = "nio";
            break;
        default:
            n = "inte ensiffrigt heltal";
            break;
        }

        if (v < 0)
            n = "minus " + n;

        synchronized (this)
        {
            this.varde = v;
            this.namn = n;
        }
}
```

Man deklarerar inte hela metoden som `synchronized`. En kritisk sektion avskiljs och placeras i slutet av metoden, och det är endast denna del som deklareras som `synchronized`. På så sätt kan låsningstiden för det aktuella objektet minskas. Eftersom man låser det objekt som aktiverar metoden `set`, används referensen `this` efter nyckelordet `synchronized`. Det är `this`-objektet som låses.

Metoden `toString` är relativt kort, och den kan behållas som `synchronized`. För att optimera synkroniseringen även i denna metod, kan man göra så här:

```
public String toString ()
{
    int       v = 0;
    String    n = null;
```

© FADIL GALJIĆ

```
    synchronized (this)
    {
        v = this.varde;
        n = this.namn;
    }

    String    s = ;

    s = "[" + v + ", " + n + "]";

    return s;
}
```

Det aktiverande objektet är bara låst medan uppgifterna om objektets numeriska värde och namn hämtas. Dessa uppgifter lagras i två lokala variabler i metoden. Därefter låses det aktuella objektet upp. De lokala variablerna kan sedan användas i en hur komplicerad procedur som helst. Användningen av dessa variabler kan inte orsaka någon interferens mellan olika trådar, eftersom varje tråd har sina egna lokala variabler.

De klasser och program som är avsedda att användas i samband med flera trådar, måste alltså utformas på ett speciellt sätt. I ett program med flera trådar uppstår särskilda problem, och för att hantera dessa problem måste motsvarande strategier användas.

Synkronisering på klientsidan

Sorterbara ensiffriga heltal

Klassen `EnsiffrigtHeltal` representerar ett ensiffrigt heltal. I denna klass har man en konstruktor, metoden `toString` som returnerar ett ensiffrigt heltals strängrepresentation, och metoden `set` som anger ett ensiffrigt heltals numeriska värde och namn utifrån ett givet heltalsvärde.

För att kunna sortera heltal av typen `EnsiffrigtHeltal`, behöver man definiera ett sätt att jämföra två heltal av denna typ. Detta kan man göra genom att implementera gränssnittet `java.lang.Comparable` i klassen `EnsiffrigtHeltal`. Man behöver implementera metoden `compareTo` (som anges i gränssnittet), och definiera där en jämförelsestrategin. Metoden kan implementeras så här:

```
public int compareTo (EnsiffrigtHeltal heltal)
{
    int    v1 = 0;
    int    v2 = 0;
    synchronized (this)
```

 © FADIL GALJIĆ

```
    {
        v1 = this.varde;
        synchronized (heltal)
        {
            v2 = heltal.varde;
        }
    }

    int    p = 0;
    if (v1 < v2)
        p = -1;
    else if (v1 > v2)
        p = 1;

    return p;
}
```

Heltalets (`this`-objektets) numeriska värde jämförs med argumentets numeriska värde. Om de två värdena är lika returneras `0`, om `this`-heltalets numeriska värde är mindre returneras `-1`, och om `this`-heltalets numeriska värde är större returneras `1`. Med hjälp av metoden `compareTo` kan två heltal av typen `EnsiffrigtHeltal` ordnas i en följd.

Två numeriska värden som gäller vid ett visst tillfälle jämförs. Därför låses både `this`-heltalet och det heltal som anges som argument. För att optimera synkroniseringen är de två objekten endast låsta medan deras numeriska värden hämtas. Objekten låses därefter upp, och de hämtade värdena jämförs. I det här fallet räcker det inte med att metoden `compareTo` deklareras som `synchronized`. Även argumentobjektet måste låsas medan dess numeriska värde hämtas.

Synkroniserad användning av en vektor

Man kan skapa en klass som definierar en ny typ av objekt. Denna klass kan implementeras på så sätt att den definierar helt synkroniserade objekt. Då kan ett objekt av klassen inte hamna i ett inkonsistent tillstånd när flera trådar använder objektet. Synkroniseringsmekanismer byggs in i definitionsklassen, och dessa mekanismer skyddar objekt av klassen.

I vissa fall används objekt som inte har några inbyggda synkroniseringsmekanismer. I dessa fall måste användningen av objekten synkroniseras i den klass där de används. En klass som använder objekt av en annan klass kallas för en klientklass. Man säger därför att synkroniseringen utförs på klientsidan (istället för i definitionsklassen).

En vektor av den inbyggda typen är ett objekt i Java. Men en sådan vektor har inga inbyggda synkroniseringsmekanismer. Om flera trådar samtidigt använder en vektor, kan vektorn hamna i ett inkonsistent tillstånd. Användningen av en vektor måste därför synkroniseras. Vektorn måste låsas (varje vektor har ett eget lås) medan den används från en tråd.

I klassen `Sorterare` definieras en tråd som sorterar en vektor med ensiffriga heltal:

```
class Sorterare implements Runnable
{
    private EnsiffrigtHeltal[]    heltal;

    public Sorterare (EnsiffrigtHeltal[] heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        synchronized (heltal)
        {
            java.util.Arrays.sort (heltal);
        }
    }
}
```

Heltalen sorteras med metoden `sort` i klassen `java.util.Arrays`. Denna metod kan sortera en vektor med ensiffriga heltal, eftersom klassen `EnsiffrigtHeltal` implementerar gränssnittet `java.lang.Comparable`. Vektorns lås hålls medan vektorn sorteras. Det kan därför inte inträffa att någon annan tråd använder vektorn under sorteringen. Vektorn kan användas före och efter sorteringen, men inte när sorteringen pågår.

Man kan skapa en vektor med ensiffriga heltal, och sortera och visa denna. Detta kan göras så här:

```
class SynkroniseringPaKlientSidan
{
    public static void main (String[] args)
    {
        EnsiffrigtHeltal[]    v = new EnsiffrigtHeltal[10];
        for (int i = 0; i < 10; i++)
            v[i] = new EnsiffrigtHeltal ();
        for (int i = 0; i < 10; i++)
            v[i].set (9 - i);

        Thread    t = new Thread (new Sorterare (v));
        t.start ();
```


© FADIL GALJIĆ

```
        synchronized (v)
        {
            for (int i = 0; i < 10; i++)
                System.out.println (v[i]);
        }
    }
}
```

Här skapas en vektor som innehåller de ensiffriga heltalen `9`, `8`, `7`, `6`, `5`, `4`, `3`, `2`, `1` och `0`. Sedan skapas en tråd av typen `Sorterare` som sorterar heltalen. Slutligen visas vektorn.

Vektorn `v` används både i tråden `t` och i `main`-tråden. Därför låser man vektorn medan dess element skrivs ut. Vektorn låses både vid sorteringen och vid utskriften. På så vis förhindras att vektorn växelvis sorteras och visas. Vektorn visas i sin helhet, antingen före eller efter sorteringen (beroende på i vilken ordning trådarna exekverar). Om inte användandet av vektorn synkroniseras, kan vektorn återges på ett felaktigt sätt. Utskriften kan då få följande form:

```
[9, nio]
[8, åtta]
[7, sju]
[6, sex]
[5, fem]
[5, fem]
[6, sex]
[7, sju]
[8, åtta]
```

I det här fallet visas en del av vektorn före sorteringen, och resten efter sorteringen. Detta ger en felaktig information om vektorns element.

Synkronisering på klientsidan används vid användning av objekt som inte har inbyggda synkroniseringsmekanismer. Men synkronisering på klientsidan kan även användas i de fall där helt synkroniserade objekt används. Detta sker då flera operationer kombineras i en sammansatt synkroniserad operation. I detta fall behålls objektets lås medan operationerna på objektet utförs.

Synkroniserade behållare

Synkroniseringen tar tid, och därför är de flesta klasser i Javas standardbibbliotek inte synkroniserade. Ofta används olika resurser av bara en tråd i taget, och i så fall behövs ingen synkronisering. I vissa fall definieras två klasser som beter sig likadant, men den ena klassen är synkroniserad och

© FADIL GALJIĆ

den andra klassen inte. Det är fallet med exempelvis klasserna java.lang.StringBuilder och java.lang.StringBuffer. De båda klasserna representerar en teckenbehållare, där olika tecken kan lagras och bearbetas. Klassen StringBuilder är en snabb klass, tänkt att användas i samband med en tråd i taget. Om flera trådar samtidigt ändrar en och samma sträng (eller en tråd ändrar strängen, och andra trådar använder den), så ska strängen representeras via ett objekt av klassen StringBuffer.

I paketet java.util finns ett antal klasser, som definierar olika objektbehållare (vektorer, länkade listor, häshtabeller, mängder och så vidare). Metoder i de flesta av dessa klasser är inte synkroniserade. Det gäller klasserna ArrayList, LinkedList, HashMap, HashSet och andra. Genom att inte synkronisera klasserna, definieras snabba behållare, som passar bra vid olika tillfällen. Men dessa behållare kan vara olämpliga om de används av flera trådar samtidigt. Om (minst) en tråd ändrar en behållare (lägger till eller tar bort element) medan andra trådar använder den, kan olika problem uppstå.

När flera trådar samtidigt använder en osynkroniserad behållare, behöver man synkronisera denna användning. Man ska låsa behållaren när den används (man kan skapa ett synkroniserat block, som använder behållarens lås), så att en tråd i taget kommer åt den. Men det finns även en annan strategi. Utifrån en osynkroniserad behållare kan man skapa motsvarande synkroniserad behållare, och använda den istället. Man kan göra det med en lämplig metod från klassen java.util.Collections.

Man kan skapa en osynkroniserad vektor av typen java.util.ArrayList så här:

```
List<Object>    v = new ArrayList<Object> ();
```

Utifrån den här vektorn kan en synkroniserad vektor skapas. Man använder den statiska metoden synchronizedList från klassen Collections:

```
List<Object>    w = Collections.synchronizedList (v);
```

Vektorn w är ett synkroniserat objekt, och kan samtidigt användas av flera trådar. Metoderna add, remove, size, toString och andra är synkroniserade, och kan inte exekveras samtidigt i samband med vektorn.

Förutom metoden synchronizedList, finns flera andra metoder i klassen Collections som utför samma funktion (skapar en synkroniserad behållare utifrån en osynkroniserad behållare). Metoden synchronizedSet skapar en synkroniserad mängd, metoden synchronizedMap skapar en synkroniserad hashtabell och så vidare. Alla dessa metoder returnerar en referens av motsvarande gränssnitt (Collection, List, Set, Map och så vidare), som

© FADIL GALJIĆ

refererar till den nyskapade behållaren. Även parametrar i dessa metoder är referenser av motsvarande gränssnit, som gör möjligt att man använder en och samma metod i samband med olika typer behållare. Metoded `synchronizedList`, till exempel, har en parameter av typen `java.util.List`. Tack vare det kan metoden användas i samband med vektorer av typen `java.util.ArrayList`, och med länkade listor av typen `java.util.LinkedList`.

Visa problem kan uppstå även i samband med en synkroniserad behållare. Det är bara enskilda operationer (metoder) som är synkroniserade. Men det kan hända att man vill utföra flera operationer på en behållare utan att släppa dess lås. Man vill till exempel stega igenom behållaren, och utföra olika operationer i samband med dess element. I så fall måste behållaren låsas under hela operationen. Om `v` är en synkroniserad vektor, kan man göra så här:

```
synchronized (v)
{
    for (int i = 0; i < v.size (); i += 2)
        v.remove (i);
}
```

Man låser vektorn `v` och stegar igenom den. Varannat element tas bort från vektorn.

Det bör också nämnas att det i paketet `java.util.concurrent` finns flera behållarklasser, som redan är synkroniserade. Klassen `java.util.concurrent.ConcurrentHashMap` representerar en synkroniserad hashtabell. Klassen implementerar gränssnittet `java.util.concurrent.ConcurrentMap`, som är ett subgränssnitt till gränssnittet `java.util.Map`. Klassen `java.util.concurrent.ConcurrentLinkedQueue` representerar en synkroniserad obegränsad kö. Klassen implementerar gränssnittet `java.util.Queue`. Ett antal klasser representerar en synkroniserad *blockeringskö*, som kan användas i olika sammanhang i samband med flera trådar. Alla dessa klasser implementerar gränssnittet `java.util.concurrent.BlockingQueue`, som är ett subgränssnitt till gränssnittet `java.util.Queue`. Bland dessa klasser kan nämnas klasserna `ArrayBlockingQueue`, `LinkedBlockingQueue` och `PriorityBlockingQueue` (alla i paketet `java.util.concurrent`).

© FADIL GALJIĆ

Synkroniserad användning av flera objekt

Exklusivt ägande av flera objekt

En tråd kan behöva flera objekt samtidigt för att kunna fullgöra sin uppgift. En tråd kan till exempel behöva två objekt samtidigt för att kunna jämföra objekten. Eller det kan behövas två objekt för att ett tredje objekt ska kunna bestämmas utifrån dessa två objekt. När en tråd använder flera objekt samtidigt, kan tråden behöva låsa alla dessa objekt. Tråden kan vilja ha exklusiv rätt att använda dessa objekt, medan den utför en viss operation i samband med objekten. När en tråd exempelvis jämför två objekt, ska ingen annan tråd få chans att ändra något av objekten (annars kan resultatet av jämförelsen bli meningslöst).

Rätt till exklusivt ägande av flera objekt kan implementeras genom ett nästlat synkroniserat block. Ett sådant block kan föreställas så här:

```
synchronized (objekt1)
{
    // eventuell kod

    synchronized (objekt2)
    {
        // använd de båda två objekten
    }

    // eventuell kod
}
```

När en tråd exekverar denna kodsektion, låser tråden först objektet `objekt1`. Tråden utför eventuellt vissa operationer, och sedan låser den även objektet `objekt2`. Tråden använder två angivna objekt under en viss tid, och släpper sedan objektet `objekt2`. Ytterligare operationer utförs eventuellt, och slutligen släpps även objektet `objekt1`.

Ett nästlat synkroniserat block kan användas både i en definitionsklass (där ett objekts beteende definieras) och på klientsidan (där objekt av klassen används, till exempel i en klass som definierar en tråd). Ett nästlat synkroniserat block med fler nivåer än två kan skapas, och på så sätt går det att implementera rätt till exklusivt ägande av flera objekt.

 © FADIL GALJIĆ

För att åstadkomma rätt till exklusivt ägande av flera objekt, kan man kombinera synkroniserade metoder och synkroniserade block. Ett anrop till en synkroniserad metod kan placeras i ett synkroniserat block, till exempel. Detta kan presenteras så här:

```
synchronized (objekt1)
{
    // kod

    objekt2.metod ();

    // kod
}
```

En tråd som utför denna kodsektion låser objektet `objekt1` när den kommer in i det synkroniserade blocket. Tråden anropar sedan en synkroniserad metod i samband med objektet `objekt2`, och på så sätt låses även detta objekt. En liknande situation uppkommer när man i en synkroniserad metod i en definitionsklass anropar en annan synkroniserad metod i samband med ett objekt av en annan klass. Denna situation kan föeställas så här:

```
class X
{
    public synchronized void metod1 (Y y)
    {
        // kod

        y.metod2 ();

        // kod
    }

    // kod
}
```

När en tråd anropar metoden `metod1` i samband med ett objekt av typen `X`, får tråden tillgång till objektets lås. Tråden exekverar den kod som finns i metoden, och kommer fram till den sats där metoden `metod2` anropas i samband med ett objekt av typen `Y`. Tråden låser då även objektet `y`, och håller dess lås under exekveringen av metoden `metod2`. Objektet `y` som används i metoden `metod1` kan skapas inuti denna metod, istället för att det anges som argument. Referensen `y` kan även vara en instansvariabel i klassen `X`, och motsvarande objekt kan skapas i någon av klassens konstruktorer eller metoder.

En definitionsklass kan ha en synkroniserad metod, som innehåller ett synkroniserat block. Denna struktur kan presenteras så här:

© FADIL GALJIĆ

```
class X
{
    public synchronized void metod (Y y)
    {
        // kod

        synchronized (y)
        {
            // kod
        }

        // kod
    }

    // kod
}
```

Den här metoden anropas i samband med ett objekt av typen `X`, och på så sätt låses detta objekt. Inuti metoden låses även objektet `y`, och på så sätt åstadkoms exklusivt ägande av två objekt.

Oavsett hur en tråd åstadkommer exklusivt ägande av flera objekt, låser tråden dessa objekt i en följd. Först låses ett objekt, därefter ett annat objekt och så vidare. Tråden släpper sedan objekten i omvänd ordning: det sist låsta objektet släpps först.

Ett dödläge

Ett dödläge i ett program

Det händer att flera trådar samtidigt använder flera gemensamma objekt. För att alla dessa objekt ska bevaras i ett konsistent tillstånd, synkroniseras deras användning. När en tråd använder ett objekt, låses objektet så att andra trådar inte kan använda det. Om en tråd samtidigt använder flera objekt, låses alla dessa objekt. Men denna teknik, som används för att bevara objekten i ett konsistent tillstånd, kan orsaka vissa oönskade konsekvenser. Flera trådar kan låsa flera objekt och vilja låsa ytterligare objekt, men dessa nya objekt kan redan vara låsta. I detta fall kan det hända att vissa (kanske alla) av dessa trådar låser sig, och därmed inte kan fortsätta exekvera.

Ett program kan skapa två objekt `n1` och `n2`, och två trådar `t1` och `t2` som samtidigt använder dessa objekt. Det kan hända att tråden `t1` låser objektet `n1` och försöker låsa objektet `n2`. Men tråden `t2` kan redan ha hunnit låsa objektet `n2`, så att tråden `t1` måste invänta objektets lås. Tråden `t2`

© FADIL GALJIĆ

kan samtidigt försöka låsa objektet `n1`, men eftersom detta objekt redan är låst, måste den vänta på det. Tråden `t1` håller objektet `n1` och väntar på objektet `n2`. Tråden `t2` håller objektet `n2` och väntar på objektet `n1`. Ingen av dessa trådar kan fortsätta exekvera. Ett *dödläge* har uppstått. Andra trådar i programmet kan eventuellt fortsätta exekvera, men det kan hända att även dessa trådar snabbt låser sig. I så fall hänger sig hela programmet.

Klassen `EnsiffrigtHeltal` representerar ett ensiffrigt heltal. Man kan skapa två objekt av denna klass, och två trådar som samtidigt använder dessa objekt. Objekten kan kallas för `n1` och `n2`, och trådarna för `t1` och `t2`. Tråden `t1` kan definieras i en klass som heter `Anvandare1`:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal1;
    private EnsiffrigtHeltal    heltal2;

    public Anvandare1 (EnsiffrigtHeltal heltal1,
                       EnsiffrigtHeltal heltal2)
    {
        this.heltal1 = heltal1;
        this.heltal2 = heltal2;
    }

    public void run ()
    {
        for (int i = 0; i < 10; i++)
        {
            int    v1 = (int) (10 * Math.random ());
            int    v2 = (int) (10 * Math.random ());

            synchronized (heltal1)
            {
                heltal1.set (v1);

                synchronized (heltal2)
                {
                    heltal2.set (v2);
                    System.out.println (heltal1 + " " + heltal2);
                }
            }
        }
    }
}
```

Klassen `Anvandare1` definierar en tråd som har tillgång till två objekt av typen `EnsiffrigtHeltal`. En tråd av denna typ låser ett av två givna objekt, och sätter (via metoden `set`) dess numeriska värde och namn utifrån ett slumpmässigt heltalsvärde mellan `0` och `9`. Därefter låses även det

andra objektet, dess numeriska värde och namn sätts, och båda objekten skrivs ut. För att utföra en synkroniserad ändring och utskrift av två objekt, håller tråden de båda objekten låsta under en kort tid.

En liknande klass, `Anvandare2`, som använder två objekt av typen `EnsiffrigtHeltal` och sätter deras numeriska värden och namn, kan skapas. I denna klass ska slumpmässiga värden vara mellan `-9` och `-1`. När klasserna `EnsiffrigtHeltal`, `Anvandare1` och `Anvandare2` har definierats, kan två ensiffriga heltal skapas och två trådar som använder dessa heltal. Detta kan göras så här:

```
class Dodlage
{
    public static void main (String[] args)
    {
        EnsiffrigtHeltal    n1 = new EnsiffrigtHeltal ();
        EnsiffrigtHeltal    n2 = new EnsiffrigtHeltal ();

        Thread    t1 = new Thread (new Anvandare1 (n1, n2));
        Thread    t2 = new Thread (new Anvandare2 (n2, n1));

        t1.start ();
        t2.start ();
    }
}
```

När programmet startas kan tråden `t1` hinna låsa objektet `n1`, och tråden `t2` kan hinna låsa objektet `n2`. I så fall väntar tråden `t1` på objektet `n2`, och tråden `t2` på objektet `n1`. Ingenting kan ändra denna situation (avbryta väntandet) och trådarna kan inte fortsätta exekvera. Ett dödläge uppstår och programmet hänger sig.

Ett dödläge kan uppstå innan någonting hinner skrivas ut. Det kan också hända att flera operationer hinner utföras innan ett dödläge uppstår. En utskrift som har följande form kan skapas:

```
[0, noll] [9, nio]
[5, fem] [9, nio]
[8, åtta] [6, sex]
[9, nio] [7, sju]
[1, ett] [0, noll]
```

Tråden `t1` passerar i detta fall fem gånger genom loopen, och startar sitt sjätte pass genom att låsa objektet `n1`. Då startar den andra tråden sitt första pass och hinner låsa objektet `n2`, eftersom denna tråd har objektet `n2` som `heltal1` (`n2` anges som första argument när tråden `t2` skapas – referensen `heltal1` i tråden refererar till `n2`). Efter det kan ingen av trådarna fortsätta exekvera.

 © FADIL GALJIĆ

Det kan också hända att ett dödläge inte alls uppstår, och att trådarna hinner utföra sina uppgifter. En tråd kan till exempel få möjlighet att exekvera först, och den andra tråden sedan. Eller så kan en tråd utföra ett eller flera hela pass genom loopen innan den andra tråden får möjlighet att exekvera. Den andra tråden kan utföra ett eller flera hela pass genom loopen när den första tråden får tillfälle att fortsätta exekvera. Aktiviteten kan fortsätta på samma sätt utan att ett dödläge uppstår. I detta fall får en tråd möjlighet att exekvera just när den andra tråden släpper de båda objekten. Olika trådar använder två gemensamma objekt under olika tidpunkter, och på så vis ett dödläge undviks.

Förhindra dödlägen

Tråden `t1` låser först objektet `n1` och därefter objektet `n2`. Tråden `t2` låser först objektet `n2` och därefter objektet `n1`. Denna ordning är den avgörande faktor som möjliggör ett dödläge. Om ordningen ändras så att även tråden `t2` först låser objektet `n1`, elimineras risken för ett dödläge. I detta fall kan tråden `t1` låsa objektet `n1`, men då kan inte tråden `t2` låsa objektet `n2`, eftersom även denna tråd först måste låsa objektet `n1`. Men eftersom objektet `n1` redan är låst, måste tråden `t2` vänta. Tråden `t1` kan fortsätta exekvera, och kan utan problem låsa även objektet `n2`. Den utför sedan de nödvändiga operationerna, och släpper därefter objekten. Samma situation uppstår om tråden `t2` först hinner låsa objektet `n1`. I detta fall måste tråden `t1` vänta.

Ordningen i vilken tråden `t2` låser enskilda objekt kan ändras genom att objektet `n1` anges som första argument när tråden skapas. Detta görs så här:

```
Thread    t2 = new Thread (new Anvandare2 (n1, n2));
```

I det här fallet refererar referensen `heltal1` i tråden `t2` till objektet `n1`, och det är detta objekt som låses först.

Samma strategi kan användas för att förhindra ett dödläge när flera trådar använder flera gemensamma objekt. Det som egentligen orsakar ett dödläge är en cirkulär kedja (en ring) av trådar som väntar på varandra. Tråden `t1`, till exempel, kan äga objektet `n1` och vänta på objektet `n2`. Tråden `t2` kan äga objektet `n2` och vänta på objektet `n3`. Tråden `t3` kan äga objektet `n3` och vänta på objektet `n1`. Detta är en cirkulär kedja av trådar som väntar på varandra. Ett dödläge uppstår, och ingenting kan avbryta detta dödläge. Ett dödläge kan inte uppstå om alla tre trådarna låser objekten i en förbestämd ordning, till exempel först `n1`, sedan `n2` och till sist `n3`. I

detta fall kan det inte inträffa att en tråd äger objektet `n3` och begär objektet `n1`. Denna tråd ska först begära objektet `n1`, och kan bara fortsätta vidare och begära objektet `n3` om den får det förstnämnda objektet. En tråd som begär flera objekt måste först begära det objekt som kommer först i den fastställda ordningen.

För att ett dödläge ska kunna uppstå i ett program, måste det finnas en cirkulär kedja av väntande trådar. I denna kedja äger varje tråd vissa resurser, och väntar på andra resurser som ägs av andra trådar. Om programmen utformas så att de inte innehåller dessa cirkulära kedjor av väntande trådar, förhindras ett dödläge. För att underlätta användningen av denna strategi, kan man numrera de objekt som samtidigt används av flera trådar. I detta fall ska objekten tilldelas till en tråd i stigande ordning. Om en tråd behöver objekt 2, objekt 5 och objekt 7, ska tråden kräva dessa objekt i just denna (stigande) ordning. Då kan ett dödläge inte uppstå. Det kan inte inträffa att en tråd äger objekt 7 och begär objekt 2. På så sätt kan man förhindra att en ring av väntande trådar uppstår (man kan inte stänga kedjan och bilda en ring).

Det finns även en annan strategi som förhindrar att en cirkulär kedja av väntande trådar bildas, och att ett dödläge uppstår. En tråd som äger ett objekt, och försöker få tillgång till ett annat objekt som vid tillfället är låst, behöver släppa det objekt som den äger. Tråden ska vänta tills en mer gynnsam situation uppstår. Under denna tid ska den inte förhindra att andra trådar utför sina uppgifter.

© FADIL GALJIĆ

Synkronisering av tillståndsberoende operationer

Tillståndsberoende operationer

Operationer som beror på ett objekts tillstånd

En klass som definierar en objekttyp definierar olika operationer som kan utföras i samband med ett objekt av klassen. Man definierar en sådan operation genom att definiera en instansmetod i definitionsklassen. Ibland kan en viss operation bara utföras i samband med ett objekt, om objektet är i ett lämpligt tillstånd. Det kan exempelvis bara gå att ta ut pengar från ett konto i en bank, om det finns tillräckligt med pengar på kontot. När uttagningsoperationen utförs i samband med ett kontoobjekt, måste kontots tillstånd först kontrolleras. Uttagningsoperationen är en tillståndsberoende operation.

Om ett objekts tillstånd inte tillåter att en viss operation utförs, kan ett undantag av en lämplig typ kastas och därmed avslutas operationen. Om det endast är en tråd som använder objektet, finns det inte någon annan utväg. Men om flera trådar använder ett och samma objekt, kan en annan teknik användas. Man kan tillfälligt lämna operationen, och vänta. Andra trådar kan så småningom modifiera objektet på ett gynnsamt sätt, så att det blir möjligt att genomföra även den avbrutna operationen. Om det till exempel inte går att ta ut pengar på ett konto, kan man vänta tills en annan användare av kontot (en annan tråd i programmet) sätter in tillräckligt med pengar på kontot. Efter en eller flera insättningar kan eventuellt den önskade uttagningsoperationen utföras. En insättningsoperation ändrar ett konto på så sätt att en uttagningsoperation blir möjlig.

En tillståndsberoende operation

Klassen `EnsiffrigtHeltal` representerar ett ensiffrigt heltal. Denna klass hanterar ett ensiffrigt heltals numeriska värde och namn. Ett heltals numeriska värde representeras med instansvariabeln `varde`, och heltalets namn med instansvariabeln `namn`. Klassen innehåller metoden `set`, som bestämmer ett heltals numeriska värde och namn utifrån ett givet heltalsvärde. Klassen innehåller även metoden `toString`, som returnerar ett heltals strängrepresentation. Denna klass kan användas så här:

© FADIL GALJIĆ

```
EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();
n.set (5);
System.out.println (n.toString ());
```

Detta ger följande utskrift:

```
[5, fem]
```

I klassen `EnsiffrigtHeltal` kan metoden `add` definieras, som ökar eller minskar ett heltal med ett givet heltalsvärde. Metoden kan definieras så här:

```
public synchronized void add (int p)
{
    int    v = this.varde + p;
    this.set (v);
}
```

Heltalets numeriska värde beräknas, och metoden `set` (från samma klass) anropas för att sätta objektets numeriska värde och namn. Metoden `add` kan användas både för att öka ett heltal (om ett positivt argument anges), och för att minska ett heltal (om ett negativt argument anges). Metoden kan användas så här:

```
EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();
n.set (5);
n.add (4);
n.add (-1);
System.out.println (n.toString ());
```

Här skapas objektet `n`, som representerar det ensiffriga heltalet `5`. Detta heltal ökas sedan med `4`, och minskas efter det med `1` (ökas med `-1`). Därefter representerar objektet heltalet `8`, vilket ger följande utskrift:

```
[8, åtta]
```

Operationen `add` är en tillståndsberoende operation. Denna operation kan endast utföras om det nya värdet för det ensiffriga heltalet inte blir större än `9` eller mindre än `-9`. Ett ensiffrigt heltal kan ändras, men det måste förbli inom vissa gränser. Heltalets numeriska värde kan variera mellan `-9` (inklusive) och `9` (inklusive). Ett ensiffrigt heltal måste förbli ett ensiffrigt heltal även efter operationen `add`.

Metoden `add` utför inte någon kontroll av heltalets aktuella värde. Ett heltal kan därför hamna i ett otillåtet tillstånd, när metoden appliceras i samband med det, som i följande exempel:

```
EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();
n.set (5);
n.add (7);
System.out.println (n);
```


© FADIL GALJIĆ

Objektet `n` representerar det ensiffriga heltalet `5`. Till detta heltal adderas heltalsvärdet `7`. Objektet `n` hamnar därmed i ett otillåtet tillstånd (objektets numeriska värde blir `12`, men ett ensiffrigt heltal kan inte anta detta värde). Detta kan ge följande utskrift:

```
[12, inte ensiffrigt heltal]
```

Ett heltal av typen `EnsiffrigtHeltal` kan skapas, och flera trådar som använder detta heltal. En tråd kan öka heltalet, en annan tråd kan minska heltalet, en tredje tråd kan både öka och minska heltalet, och så vidare. Metoden `add` är en synkroniserad metod, och det kan inte inträffa att ett objekt av typen `EnsiffrigtHeltal` hamnar i ett inkonsistent tillstånd som ett resultat av samtidiga användningar av metoden i samband med detta objekt. Om även andra metoder i klassen `EnsiffrigtHeltal` synkroniseras, kan bara en tråd åt gången använda ett objekt av denna klass. Men ett objekt av klassen kan hamna i ett otillåtet tillstånd på ett annat sätt. Det händer om metoden `add` appliceras i samband med objektet med ett argument som inte passar med objektets aktuella tillstånd. Om objektets numeriska värde är till exempel `9`, och metoden appliceras med argumentet `2`, hamnar objektet i ett otillåtet tillstånd.

När flera trådar använder ett och samma objekt av typen `EnsiffrigtHeltal`, kan dessa trådar eventuellt samordnas så att heltalet hela tiden förblir i ett tillåtet tillstånd. En tråd kan vänta med ökningen, om denna ökning försätter heltalet i ett otillåtet tillstånd. Andra trådar kan eventuellt minska heltalet, så att även denna ökning blir möjlig. På samma sätt kan man vänta med en minskning tills andra trådar ökar det gemensamma heltalet till en lämplig nivå. Det måste finnas en teknik som gör det möjligt för en tråd att vänta tills andra trådar har modifierat det gemensamma objektet till ett passande tillstånd.

Synkronisera tillståndsberoende operationer

Synkroniseringsmetoder i klassen Object

Klassen `java.lang.Object` innehåller metoderna `wait`, `notify` och `notifyAll`, som används till synkronisering av tillståndsberoende operationer. Eftersom klassen `Object` ligger i roten av Javas klasshierarki, ärver alla klasser dessa metoder. Metoderna deklareras som `final` i klassen `Object`, och kan därför inte omdefinieras.

© FADIL GALJIĆ

En tråd kan bara använda metoderna `wait`, `notify` och `notifyAll` i samband med ett objekt, om tråden håller objektets lås. Om en tråd som inte håller objektets lås anropar någon av dessa metoder i samband med objektet, kastar metoden ett undantag av typen `java.lang.IllegalMonitorStateException`. Detta innebär att de här metoderna bara kan anropas inuti en synkroniserad metod eller inuti ett synkroniserat block. Om någon av metoderna anropas i en metod som inte är synkroniserad, måste metoden (som inte är synkroniserad) användas i ett synkroniserat block, eller inuti en annan metod som är synkroniserad.

En tråd kan försöka utföra en tillståndsberoende operation i samband med ett objekt. Tråden måste vid en lämplig tidpunkt kontrollera objektets tillstånd, för att ta reda på om den kan fortsätta med operationen eller inte. Objektet kan vara i ett sådant tillstånd som tillåter att operationen går vidare. Om objektets aktuella tillstånd inte är lämpligt, anropas metoden `wait` i samband med objektet för att vänta tills ett bättre tillfälle för operationen uppstår. Den tråd som anropar metoden `wait` i samband med objektet släpper objektets lås och blockeras. Tråden placeras bland en mängd trådar som väntar på ändringen av objektets tillstånd. Förutom att varje objekt i Java har ett lås, har varje objekt även en *`wait`-mängd* (där olika trådar kan placera sig och vänta på ändringen av objektets tillstånd).

Medan en mängd trådar väntar på en gynnsam ändring av ett objekt, kan en annan tråd utföra en operation som ändrar objektet. Det kan vara vilken operation som helst som modifierar det aktuella objektet. Om ändringen av objektets tillstånd medför att en eller flera trådar i `wait`-mängden eventuellt kan fortsätta med sina operationer, anropas metoden `notifyAll` i samband med objektet. Detta anrop aktiverar alla trådar i objektets `wait`-mängd, och de kan åter få möjlighet att exekvera. Den tråd som anropar metoden `notifyAll` fortsätter med sin operation, och släpper det aktuella objektets lås när den avslutar den (synkroniserade) operationen. En av de återaktiverade trådarna kan få möjlighet att exekvera, och kan få tillgång till objektets lås (om låset har släppts och ingen annan tråd har fått tillgång till det). Tråden fortsätter från den punkt där den avbröts. Den kontrollerar åter objektets tillstånd, och fortsätter med sin operation om så är möjligt. Om tillståndet fortfarande inte passar, ska tråden åter anropa metoden `wait` i samband med objektet.

Metoden `wait` anropas i samband med ett objekt enligt följande mönster:

```
while (!okTillstånd)
    objekt.wait ();

// kod som utförs om tillståndet är ok
```

 © FADIL GALJIĆ

Det aktuella tillståndet kontrolleras, och om tillståndet inte är lämpligt blockeras tråden genom anrop till metoden `wait`. När tråden sedan får möjlighet att fortsätta, kontrollerar den det aktuella tillståndet igen. Ändringar som andra trådar har utfört i objektet kan vara otillräckliga, och i så fall anropar tråden metoden `wait` ånyo. Därför placeras anropet till metoden `wait` i en loop, inte i en `if`-sats. Om en tråd blockeras inuti en loop, fortsätter tråden med loopen när den återaktiveras och får tillfälle att exekvera.

Så här ska man inte göra:

```
if (!okTillstånd)    // fel
    objekt.wait ();
```

Om metoden `wait` anropas i en `if`-sats, fortsätter den exekverande tråden direkt efter denna sats när den återaktiveras. Den kontrollerar inte det senaste tillståndet, trots att detta tillstånd fortfarande kanske inte passar för den operation som ska utföras. Anropet till metoden `wait` måste placeras i en loop, och villkoret i loopen knytas till det aktuella tillståndet (inte till ett tillstånd som gällde tidigare).

Metoden `notifyAll` anropas i samband med ett objekt för att aktivera de trådar som väntar i objektets `wait`-mängd. Metoden anropas efter det att objektet har ändrats, så att de avbrutna operationerna eventuellt kan fortsätta. Detta anrop kan finnas i samma metod eller block där anropet till metoden `wait` finns, eller i någon annan metod eller i ett annat block. Metoden `notifyAll` kan anropas direkt efter ändringen eller på något annat ställe efter ändringen (till exempel i slutet av metoden). Detta spelar ingen roll, eftersom tråden som anropar metoden `notifyAll` håller det aktuella objektets lås även efter anropet till metoden `notifyAll`. Tråden släpper låset när den avslutar metoden eller blocket som den exekverar. Metoden `notifyAll` anropas så här:

```
// instruktioner som ändrar objektet
// eventuella andra instruktioner

objekt.notifyAll ();

// eventuella andra instruktioner i samma synkroniserad metod
// eller i samma synkroniserat block
```

När en tråd anropar metoden `wait` i samband med ett objekt, släpper den objektets lås. Detta kan inträffa mitt i en synkroniserad metod (eller ett synkroniserat block), efter det att objektet delvis ändrats. Andra trådar kan få möjlighet att exekvera i samband med objektet, och även dessa kan ändra objektet. Dessa ändringar kan vara motsägelsefulla, och objektet kan hamna i ett inkonsistent tillstånd. För att förhindra detta måste man

© FADIL GALJIĆ

säkerställa att ett objekt lämnas i ett konsistent tillstånd vid anropet till `wait`-metoden. Därför brukar anropet till metoden `wait` placeras i början av en synkroniserad metod eller ett synkroniserat block, innan någon ändring av det aktuella objektet utförs.

Medan en tråd väntar i `wait`-anropet, kan en annan tråd anropa metoden `interrupt` i samband med denna tråd. I detta fall fortsätter tråden att vänta i `wait`-anropet, tills den med metoden `notifyAll` notifieras om en ändring. I så fall börjar den aktiverade tråden kämpa för det aktuella objektets lås, och när den får tillfälle att exekvera kastar den ett undantag av typen `java.lang.InterruptedException` (motsvarande boolesk variabel i tråden sätts till `false`). Detta undantag är ett kontrollerat undantag, som måste antingen deklareras eller fångas och hanteras. Om metoden `wait` anropas i en allmän klass som definierar en typ av objekt, brukar man deklarera detta undantag. Hanteringen av undantaget överlämnas till klassens klienter.

Metoden `notify` kan användas istället för metoden `notifyAll`. I detta fall aktiveras bara en, slumpmässigt vald tråd i `wait`-mängden. Det kan hända att det aktuella tillståndet gör det omöjligt för tråden att fortsätta, och att vissa andra trådar i `wait`-mängden skulle ha bättre möjligheter. Därför används normalt `notifyAll` istället för `notify`. Metoden `notify` kan användas i vissa speciella situationer. Man kan till exempel öka programmets hastighet genom att inte aktivera de trådar som säkert kommer att blockeras igen (eftersom de inte kan fortsätta med sina operationer). Varje blockering och aktivering belastar systemets resurser.

Klassen `Object` innehåller en variant av metoden `wait` som tar ett argument av typen `long`, som anger den längsta tid (i millisekunder) som den blockerade tråden ska vänta. Om denna metod används, aktiveras den väntande tråden när den angivna tiden går ut, även om inte någon av metoderna `notify` eller `notifyAll` har anropats (metoden aktiveras naturligtvis även vid ett anrop till `notify` eller `notifyAll`).

Synkronisera en tillståndsberoende operation

Klassen `EnsiffrigtHeltal` representerar ett ensiffrigt heltal. Metoden `add` i denna klass är en tillståndsberoende operation. Denna operation kan bara utföras om heltalets numeriska värde förblir mellan `-9` (inklusive) och `9` (inklusive) när operationen slutförs. Ett ensiffrig heltal måste förbli ett ensiffrigt heltal även efter `add`-operationen.

© FADIL GALJIĆ

Metoden wait kan användas för att förhindra att operationen add utförs, när det aktuella objektets tillstånd inte passar för operationen. Klassen EnsiffrigtHeltal ärver denna metod från klassen Object, och den kan användas i samband med ett objekt av typen EnsiffrigtHeltal. Metoden ska anropas i en while-loop. I loopen ska det ske en kontroll av om det aktuella objektets gränser överskrids när operationen add utförs. Detta kan göras på följande sätt:

```
public synchronized void add (int p) throws InterruptedException
{
    while (this.varde + p < - 9  ||  this.varde + p > 9)
        this.wait ();

    this.set (this.varde + p);
    this.notifyAll ();
}
```

För att kontrollera om det aktuella objektets gränser överskrids, beräknas det numeriska värdet som skulle gälla om operationen utfördes. Detta värde är this.varde + p. Sedan kontrolleras om värdet hamnar utanför intervallet [-9, 9], och om så är fallet anropas metoden wait. På så sätt förhindras att operationen utförs. Den tråd som anropar metoden wait släpper det aktuella (this) objektets lås och blockerar sig. Tråden lämnar objektet i ett konsistent tillstånd, eftersom ingen ändring av objektet utförs före anropet till wait.

Om det aktuella objektet är i ett tillstånd som tillåter att operationen add utförs (objektets gränser överskrids inte), anropas inte metoden wait. Exekveringen av koden fortsätter efter while-loopen. Objektets nya numeriska värde beräknas (this.varde + p), och metoden set i samma klass anropas för att sätta objektets nya värde och namn. Efter denna ändring hamnar det aktuella objektet (det objekt som metoden add anropas i samband med) i ett nytt tillstånd. Det nya tillståndet kan vara lämpligt för de operationer som väntar på att fortsätta. Därför anropas metoden notifyAll i samband med det aktuella objektet. Alla trådarna i objektets wait-mängd aktiveras. Därefter avslutar den exekverande tråden metoden add och släpper det aktuella objektets lås (som den fått när metoden add påbörjats – metoden add är en synkroniserad metod).

De återaktiverade trådarna får så småningom tillfälle att exekvera. Om en sådan tråd lyckas få tillgång till det aktuella objektets lås, fortsätter tråden från den punkt där den avbröts. I klassen EnsiffrigtHeltal kan detta bara vara i while-loopen i metoden add, eftersom det bara är i den metoden som metoden wait anropas (anropet till notifyAll i en metod kan orsaka att exekveringen av en annan metod fortsätter, men i samband

© FADIL GALJIĆ

med samma objekt). Objektets tillstånd kontrolleras på nytt, och beroende på detta tillstånd blockeras den exekverande tråden igen eller fortsätter med operationen.

När en tråd kontrollerar objektets tillstånd, måste den kontrollera det tillstånd som gäller när kontrollen utförs. Nya beräkningar, som avspeglar objektets aktuella tillstånd, behöver utföras. Tråden kan inte stödja sig på de beräkningar som utfördes när den tidigare var i metoden. Metoden `add`, till exempel, kan inte implementeras så här:

```
public synchronized void add (int p) throws InterruptedException
{
    int    v = this.varde + p;
    while (v < -9  ||  v > 9)    // fel
        this.wait ();

    this.set (this.varde + p);
    this.notifyAll ();
}
```

I det här fallet beräknas det värde som ska sättas i det aktuella objektet, och detta värde lagras i variabeln `v`. Denna variabel är en lokal variabel i metoden `add`, och den tillhör den exekverande tråden (varje tråd skapar sin egen `v`). Om den exekverande tråden blockeras i metoden `wait`, behåller den sin variabel `v`. När den sedan får tillfälle att fortsätta med operationen, fortsätter den i `while`-loopen. Den kontrollerar värdet på variabeln `v`, men detta värde avspeglar inte objektets aktuella (ändrade) tillstånd. Variabeln `v` beräknas inte på nytt, utan dess gamla värde används. Men detta värde ligger inte inom de tillåtna gränserna (annars skulle inte tråden blockeras) och tråden blockeras igen. Tråden kan inte fortsätta exekvera, och som en konsekvens kan hela programmet hamna i ett dödläge.

Det kan hända att en tråd aldrig får möjlighet att slutföra en påbörjad operation. Den ska slutföra operationen endast om det aktuella objektets tillstånd passar för operationen. Detta tillstånd kan passa redan från början, eller efter det att det ändrats av andra trådar. Om det inte finns passande villkor för en operation, ska den inte utföras. På så sätt kan en eller flera trådar vara blockerade hela tiden.

Använda en tillståndsberoende operation

Ett heltal av typen `EnsiffrigtHeltal`, och två trådar som använder detta heltal, kan skapas. En tråd kan öka heltalet och den andra tråden kan minska heltalet. På så sätt skapar den ena tråden gynnsamma villkor för den andra tråden. Ett ensiffrigt heltal kan bara ökas när det inte är alltför

© FADIL GALJIĆ

stort. Om heltalet till exempel är `9`, kan det inte ökas alls. Tråden måste invänta den tråd som minskar heltalet. På samma sätt kan ett ensiffrigt heltal inte minskas om heltalet redan är för lågt. Tråden måste i detta fall vänta på den tråd som ökar heltalet. De två trådarna kan på så vis hjälpa varandra, och det kan hända att båda trådarna lyckas utföra alla sina operationer.

På följande vis kan en tråd som har tillgång till ett ensiffrigt heltal, och som ökar detta heltals värde flera gånger, definieras:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 4; i++)
        {
            int    p = (int) (3 * Math.random ());
            synchronized (heltal)
            {
                try
                {
                    heltal.add (p);
                }
                catch (interruptedException e)
                {}

                System.out.println (heltal.toString ());
            }
        }
    }
}
```

Här produceras ett slumpmässigt heltalvärde som är antingen `0`, `1` eller `2`. Det ensiffriga heltalet ökas med detta heltalsvärde, och sedan visas heltalet. Detta upprepas fyra gånger. Ökningen av ett heltal och dess utskrift utförs som en atomär operation, i ett synkroniserat block. En tråd ändrar ett ensiffrigt heltal och visar detta heltal direkt efter ändringen. Metoden `add` i klassen `EnsiffrigtHeltal` används för att ändra ett ensiffrigt heltal. Eftersom denna metod kan kasta ett undantag av typen `InterruptedException`, anropas metoden i ett `try`-block. Om inte metoden `interrupt`

© FADIL GALJIĆ

anropas kastas inte undantaget, och inga åtgärder behöver utföras i catch-blocket.

Klassen Anvandare2, där ett ensiffrigt heltal minskas fyra gånger, kan definieras på ett liknande sätt som klassen Anvandare1. Detta heltal kan minskas med antingen 1 eller 2 (ett slumpmässigt heltalsvärde som är antingen -1 eller -2 produceras, och metoden add anropas med detta heltalsvärde som argument).

På följande sätt kan ett program med ett ensiffrigt heltal, och två trådar som använder heltalet, skapas:

```
class SynkroniseraTillstandsberoendeOperationer
{
    public static void main (String[] args)
    {
        EnsiffrigtHeltal    n = new EnsiffrigtHeltal ();
        n.set (5);

        Thread    t1 = new Thread (new Anvandare1 (n));
        Thread    t2 = new Thread (new Anvandare2 (n));

        t1.start ();
        t2.start ();
    }
}
```

Här skapas ett objekt av typen EnsiffrigtHeltal, som representerar det ensiffriga heltalet 5. Sedan skapas trådarna t1 och t2, som har tillgång till detta heltal. Tråden t1 ökar heltalet med antingen 0, 1 eller 2. Detta upprepas fyra gånger. Tråden t2 minskar heltalet med antingen 1 eller 2. Även detta upprepas fyra gånger.

Programmet startar med ett ensiffrigt heltal vars värde är 5. Om tråden t1 får tillfälle att exekvera, kan det hända att det ökar heltalet först till 7, och sedan till 9. Om heltalet ökas ytterligare, överskrids dess övre gräns (det ensiffriga heltalet blir större än 9). Men metoden add i klassen EnsiffrigtHeltal tillåter inte denna ökning, och tråden t1 blockeras. Tråden t2 får möjlighet att exekvera, och den minskar det ensiffriga heltalet. Så småningom får även ökningstråden tillfälle att exekvera och slutföra sina ökningar. Tråden t2 skapar gynnsamma förhållanden för tråden t1. Utan tråden t2 kan det hända att tråden t1 inte kan utföra sitt jobb.

När programmet SynkroniseraTillstandsberoendeOperationer exekveras, kan följande utskrift skapas:

```
[7, sju]
[9, nio]
```

 © FADIL GALJIĆ

```
[8, åtta]
[9, nio]
[7, sju]
[9, nio]
[8, åtta]
[7, sju]
```

I det här programmet säkerställs en minimal minskning med `-4` (`4` gånger `-1`). Den maximala ökningen är `8` (`4` gånger `2`). Eftersom programmet startar med `5`, kan minskningen (åtminstone delvis) kompensera ökningen, och den övre gränsen ska inte överskridas (den nedre gränsen kan inte nås). Men om programmet startar med ett värde som är större än `5` (till exempel `7`), kan det hända att den hjälp som tråden `t2` ger till tråden `t1` inte är tillräcklig. Tråden `t1` kan i så fall inte utföra sitt arbete, och programmet hamnar i ett dödläge. I detta dödläge ska tråden `t1` vänta på hjälp, men det finns inte någon annan tråd som kan ge denna hjälp. En liknande situation kan uppstå om programmet börjar med ett heltalsvärde i närheten av den nedre gränsen, till exempel med `-7`. I detta fall hjälper tråden `t1` tråden `t2`. Men det kan hända att denna hjälp inte är tillräcklig, och i så fall förblir tråden `t2` blockerad.

Synkronisering på klientsidan

Man kan skapa en klass som definierar en objekttyp, och som är tänkt att användas i samband med olika trådar. I detta fall ska `wait`-anrop byggas in i de metoder i klassen som kan bara utföras om det aktuella objektet är i ett lämpligt tillstånd. Man ska också bygga in `notifyAll`-anrop i de metoder som ändrar det aktuella objektet på så sätt att även andra operationer kan utföras i samband med objektet. På så vis kan man bygga in en kontroll av ett objekts tillstånd redan när objektets typ definieras. Men det finns många klasser som inte har inbyggda `wait-notifyAll`-mekanismer. I detta fall kan man behöva bygga in dessa mekanismer på klientsidan, när klassen används.

`wait-notifyAll`-mekanismer kan behöva användas även om dessa mekanismer är inbyggda i definitionsklassen för de objekt som används. Det kan handla om en sammansatt operation (som består av olika operationer från definitionsklassen, och eventuellt även andra operationer), som kan utföras under vissa villkor. I så fall måste dessa villkor formuleras och inväntas i klientklassen.

Metoderna `wait` och `notifyAll` används på samma sätt på klientsidan som i definitionsklassen för en objekttyp. Metoderna anropas i samband

© FADIL GALJIĆ

med det objekt vars tillståndsändring inväntas. På klientsidan kan det vara svårare att formulera villkor, eftersom det inte går att komma åt det aktuella objektets variabler. Om man till exempel använder ett objekt av typen `EnsiffrigtHeltal`, måste man känna till objektets numeriska värde. Eftersom det inte går att direkt komma åt motsvarande instansvariabel (den är privat), måste motsvarande metoder användas. Om det inte finns några sådana metoder, går det inte att formulera nödvändiga villkor.

För att kunna formulera olika villkor i samband med ett objekt av typen `EnsiffrigtHeltal`, måste klassen innehålla en metod som returnerar objektets numeriska värde. Denna metod kan heta `getVarde` (metoden ska vara `synchronized`). Med hjälp av denna metod kan man bygga in `wait-notifyAll`-mekanismer på klientsidan. Man ska göra detta om man inte har gjort det i klassen `EnsiffrigtHeltal` (det bör dock göras i klassen `EnsiffrigtHeltal` – det är bättre och lättare än att göra det i klassens klienter). Så här kan detta göras:

```
class Anvandare1 implements Runnable
{
    private EnsiffrigtHeltal    heltal;

    public Anvandare1 (EnsiffrigtHeltal heltal)
    {
        this.heltal = heltal;
    }

    public void run ()
    {
        for (int i = 0; i < 4; i++)
        {
            int    p = (int) (3 * Math.random ());

            synchronized (heltal)
            {
                try
                {
                    while (heltal.getVarde () + p < -9  ||
                           heltal.getVarde () + p > 9)
                        heltal.wait ();
                }
                catch (InterruptedException e)
                {}

                heltal.add (p);
                heltal.notifyAll ();

                System.out.println (heltal.toString ());
            }
```


© FADIL GALJIĆ

```
        }
    }
}
```

Här beräknas det värde som heltalet behöver anta, och om detta värde hamnar utanför de tillåtna gränserna utförs inte `add`-operationen. Den exekverande tråden blockeras och väntar i `wait`-mängden för det motsvarande objektet (`heltal`), tills någon annan tråd ändrar det aktuella objektet, och notifierar alla väntande trådar (som finns i objektets `wait`-mängd) om denna ändring.

© FADIL GALJIĆ

Synkroniseringsobjekt

Synkroniseringslås

Alla objekt i Java har ett inbyggt lås, som kan användas för att synkronisera användningen av olika resurser i ett program. En tråd erhåller ett objekts lås när den kommer in i en synkroniserad metod eller ett synkroniserat block. Låset släpps när metoden eller blocket avslutas (antingen normalt eller genom ett undantag). Bara en tråd i taget kan komma in i en kritisk sektion, och på så sätt undviks oönskade interferenser mellan olika trådar.

Genom att använda inbyggda lås, och manipulera dem med nyckelordet `synchronized`, kan synkroniseringen implementeras på ett lätt och elegant sätt. Men det finns i Java även en annan möjlighet att implementera synkroniseringen. Man kan skapa särskilda objekt som representerar lås, och manipulera dem på olika sätt. Genom att använda dessa *synkroniseringsobjekt* kan en större flexibilitet åstadkommas. Man är inte vidare bunden till en metod eller ett block. Ett lås kan låsas på ett ställe i ett program, och låsas upp på ett helt annat ställe. Förutom det erbjuder synkroniseringsobjekt även andra möjligheter, som inte finns i samband med de inbyggda låsen.

Ett *synkroniseringslås* definieras i gränssnittet `java.util.concurrent.locks.Lock`. Huvudmetoder i gränssnittet är metoderna `lock` och `unlock`. Med metoden `lock`erhålls ett lås (tråden blockeras och väntar tills låset blir fritt), och med metoden `unlock` släpps låset. Dessa metoder kan anropas på två godtyckliga ställen (där låset är tillgängligt) i ett program. Förutom metoderna `lock` och `unlock`, definieras även metoderna `tryLock`, `lockInterruptibly` och `newCondition`. Dessa metoder definierar ytterligare möjligheter, som ett inbyggt lås inte har. Med metoden `tryLock` kan en tråd försöka erhålla ett lås. Om låset inte är upptaget, erhåller tråden det och metoden `tryLock` returnerar `true`. Om låset är upptaget för tillfället, avslutas metoden `tryLock` genom att returnera `false`. Tråden behöver inte vänta, utan kan utföra andra operationer under tiden (eventuellt kan den senare igen försöka att erhålla låset). Metoden `tryLock` kommer även i en variant som preciserar en väntetid. Om den metoden anropas, försöker tråden erhålla låset inom den angivna tiden (väntandet kan även avbrytas med metoden `interrupt` från någon annan tråd). I metoden `lockInterruptibly` försöker tråden få låset tills den erhåller det, eller tills den blir avbruten med `interrupt`-signalen från

© FADIL GALJIĆ

en annan tråd. Metoden `newCondition` används i samband med synkroniseringen av tillståndsberoende operationer.

Klassen `java.util.concurrent.locks.ReentrantLock` (ordet R`eentrant` syftar på att en tråd som håller ett lås, kan erhålla det på nytt, till exempel genom anropet till en metod som använder samma lås) implementerar gränssnittet `Lock`. Ett objekt av klassen representerar ett synkroniseringslås, och kan användas för att synkronisera användningen av en resurs i ett program. Följande klass visar hur ett lås av typen `ReentrantLock` skapas och används.

```
import java.awt.Dimension;
import java.util.concurrent.locks.*;

class Rectangle
{
    private int    width = 5;
    private int    height = 5;

    private String[]    name = {"A", "B", "C", "D"};

    private Lock    sizeLock = new ReentrantLock ();
    private Lock    nameLock = new ReentrantLock ();


    public Rectangle ()
    {}

    public void setSize (int width, int height)
    {
        sizeLock.lock ();
        this.width = width;
        this.height = height;
        sizeLock.unlock ();
    }

    public Dimension getSize ()
    {
        sizeLock.lock ();
        Dimension    d = new Dimension (this.width, this.height);
        sizeLock.unlock ();

        return d;
     }

    public void setName (String[] name)
    {
        nameLock.lock ();
        for (int i = 0; i < name.length; i++)
```

© FADIL GALJIĆ

```
            this.name[i] = name[i];
        nameLock.unlock ();
    }

    public String getName ()
    {
        nameLock.lock ();
        String    rectName = "";
        for (int i = 0; i < name.length; i++)
            rectName += name[i];
        nameLock.unlock ();

        return rectName;
     }
}
```

Klassen `Rectangle` definierar en rektangel med givna dimensioner (`width` och `height`) och namn (`name`). Den förvalda rektangen har dimensionerna `5` och `5`, och heter `ABCD`. Rektangelns dimensioner hanteras med metoderna `setSize` och `getSize`, och dess namn hanteras med metoderna `setName` och `getName`.

När flera trådar använder en rektangel av typen `Rectangle`, kan olika oönskade interferenser uppstå. En tråd kan till exempel försöka ange rektangelns dimensioner till `4` och `4`. Denna tråd kan avbrytas mitt i metoden `setSize` av en annan tråd, som anger dimensionerna till `7` och 7. När den första tråden sedan får möjlighet att exekvera, fortsätter den i metoden `setSize` och anger rektangelns andra dimension till `4`. På så sätt blir rektangelns dimensioner `7` och `4`, någonting som ingen av trådarna ville ha. För att förhindra detta, synkroniseras ändringen av rektangelns dimensioner. Bara när båda dimensionerna ändrats, kan en annan tråd få möjlighet att exekvera koden i metoden `setSize`.

För att åstadkomma en synkroniserad behandling av rektangelns dimensioner i metoderna `setSize` och `getSize`, skapas ett synkroniseringslås av typen `ReentrantLock`. Varje objekt av typen `Rectangle` (varje rektangel) har sitt eget lås. Låset skapas så här:

```
private Lock    sizeLock = new ReentrantLock ();
```

När en tråd kommer in i metoden `setSize`, erhåller den låset `sizeLock` (om det inte är upptaget) med metoden `lock`, och utför koden i metoden. När de båda två dimensionerna anges, släpps låset (med metoden `unlock`) så att även andra trådar kan komma in i metoden och modifiera rektangelns dimensioner. Eftersom bara en tråd i taget kan äga låset, kan inte

 © FADIL GALJIĆ

oönskade interferenser uppstå. Rektangeln kan inte hamna i ett inkonsistent tillstånd när det gäller dimensioner.

Även behandlingen av rektangelns namn synkroniseras. Om en tråd ska ange namnet till `ABCD` och en annan tråd till `PQRS`, ska det förhindras att rektangeln vid ett visst tillfälle heter till exempel `PQCD`. För att synkronisera namnmanipulationer, skapas ett särskilt synkroniseringslås. Detta lås heter `nameLock` och är helt oberoende av låset `sizeLock`. Dimensionsmanipulationer och namnmanipulationer är oberoende av varandra, och därför används två separata lås. Medan en tråd modifierar en rektangelns dimensioner, kan någon annan tråd samtidigt modifiera rektangelns namn. Det kan inte uppstå oönskade konsekvenser av dessa samtidiga operationer. En tråd som manipulerar rektangelns namn behöver inte vänta på de trådar som manipulerar dess dimensioner, eller tvärtom. På så sätt optimeras synkroniseringen.

En rektangel av typen `Rectangle` har tre lås: ett inbyggt lås, samt låsen `sizeLock` och `nameLock`. Ytterligare två lås gör att ett objekt av typen `Rectangle` tar upp mer minne. Därför ska nya lås skapas och användas bara om så behövs. I många situationer räcker det med att använda de inbyggda låsen (genom att skapa `synchronized`-metoder och `synchronized`-block) .

När ett synkroniseringslås av typen `Lock` används, ska man vara noggrann med att släppa låset även i fall att ett undantag uppstår. För att åstadkomma det, kan ett `finally`-block användas. Om synkroniseringslåset `oneLock` används, kan följande mönster användas:

```
oneLock.lock ();
try
{
    // en kritisk sektion
}
finally
{
    oneLock.unlock ();
}
```

Läslås och skrivlås

Vissa trådar kan modifiera ett objekt (skrivtrådar), medan andra trådar avläser olika uppgifter om objektet (lästrådar). För att förhindra att en läströd ser ett objekt mitt i ändringen (i ett inkonsistent tillstånd), syn-

© FADIL GALJIĆ

kroniserar man även olika läsoperationer. I så fall kan inte en läståd använda objektet medan modifieringen pågår. Men denna synkronisering kan även ha negativa följder. Lästrådarna måste också vänta på varandra. Det kan hända att objektet ändras bara ibland, men att det avläses mycket ofta och av många trådar. I så fall är det förståndigt att tillåta att lästrådarna samtidigt använder objektet. Det ska finnas ett slags synkroniseringslås som tillåter samtidiga avläsningar, men förbjuder att objektet används av flera trådar medan modifieringen pågår. Sådana lås definieras i gränssnittet `java.util.concurrent.locks.ReadWriteLock`. Gränssnittet har bara två metoder: `readLock` och `writeLock`. Metoden `readLock` returnerar ett synkroniseringslås av typen `Lock` (i paketet `java.util.concurrent.locks`) som kan ägas av flera lästrådar samtidigt. Metoden `writeLock` returnerar ett synkroniseringslås av typen `Lock`, som kan ägas av bara en tråd åt gången. Låset utesluter alla andra trådar.

Klassen `java.util.concurrent.locks.ReentrantReadWriteLock` implementerar gränssnittet `ReadWriteLock`. Den har metoden `readLock` som returnerar ett *läslås*, och metoden `writeLock` som returnerar ett *skrivlås*. Följande exempel visar hur klassen `ReentrantReadWriteLock` och dess metoder kan användas.

```
import java.awt.Dimension;
import java.util.concurrent.locks.*;

class Rectangle
{
    private int    width = 5;
    private int    height = 5;

    private ReentrantReadWriteLock    rwl =
                              new ReentrantReadWriteLock ();
    private Lock    readLock = rwl.readLock ();
    private Lock    writeLock = rwl.writeLock ();


    public Rectangle ()
    {}


    public void setSize (int width, int height)
    {
        writeLock.lock ();
        this.width = width;
        this.height = height;
        writeLock.unlock ();
     }
```


© FADIL GALJIĆ

```
    public Dimension getSize ()
    {
        readLock.lock ();
        Dimension    d = new Dimension (this.width, this.height);
        readLock.unlock ();

        return d;
     }

    public double getDiagonal ()
    {
        readLock.lock ();
        double    d = Math.sqrt (this.width * this.width +
                                 this.height * this.height);
        readLock.unlock ();

        return d;
     }
}
```

Klassen `Rectangle` definierar en rektangel. Rektangelns dimensioner kan ändras med metoden `setSize`. Det är en skrivoperation, och den synkroniseras med ett skrivlås. Bara en tråd åt gången kan utföra operationen. Klassen definierar även två läsoperationer: Metoden `getSize` returnerar rektangelns dimensioner (som ett objekt av typen `java.awt.Dimension`), och metoden `getDiagonal` returnerar rektangelns diagonal. Dessa operationer kan utföras samtidigt, och därför synkroniseras dem med ett läslås. Flera trådar kan samtidigt anropa vilken som helst av dessa två metoder. Men dessa metoder kan inte exekveras parallellt med metoden `setSize`.

Om användningen av separata läslås och skrivlås förbättrar ett programs egenskaper eller inte, beror på konkreta omständigheter. I vissa kritiska situationer behöver man utföra olika tidsmätningar för att se om programmet går snabbare. I de flesta situationer behöver man inte skilja mellan läsoperationer och skrivoperationer.

Villkorsobjekt

Ibland kan en viss operation bara utföras i samband med ett objekt om objektet är i ett lämpligt tillstånd. Det kan exempelvis bara gå att ta ut pengar från ett konto i en bank, om det finns tillräckligt med pengar på kontot. När uttagningsoperationen utförs i samband med ett kontoobjekt, måste kontots tillstånd först kontrolleras. Uttagningsoperationen är en *tillståndsberoende operation*.

© FADIL GALJIĆ

Om ett objekts tillstånd inte tillåter att en viss operation utförs, kan ett undantag av en lämplig typ kastas och därmed avslutas operationen. Om det endast är en tråd som använder objektet, finns det inte någon annan utväg. Men om flera trådar använder ett och samma objekt, kan en annan teknik användas. Man kan tillfälligt lämna operationen, och vänta. Andra trådar kan så småningom modifiera objektet på ett gynnsamt sätt, så att det blir möjligt att genomföra även den avbrutna operationen. Om det till exempel inte går att ta ut pengar på ett konto, kan man vänta tills en annan användare av kontot (en annan tråd i programmet) sätter in tillräckligt med pengar på kontot. Efter en eller flera insättningar kan eventuellt den önskade uttagningsoperationen utföras. En insättningsoperation ändrar ett konto på så sätt att en uttagningsoperation blir möjlig.

Ett tillståndsberoende operation i ett program med flera trådar kan implementeras med en synkroniserad metod eller ett synkroniserat block. När en tråd kommer in i den metoden eller blocket, erhåller den det inbyggda låset av motsvarande objekt, och kontrollerar om operationen kan utföras. Om villkor för operationen inte är uppfyllda, släpper den exekverande tråden låset och väntar i metoden `wait` (i klassen `java.lang.Object`). En annan tråd kan under tiden erhålla låset, och modifiera objektet på så sätt att även den avbrutna operationen eventuellt kan fortsätta. Den exekverande tråden anropar metoden `notifyAll`, och aktiverar alla de trådar som väntar i `wait`-mängden för objektet (trådarna blir körbara igen). Någon av dessa trådar får eventuellt möjlighet att exekvera, och slutför eventuellt den avbrutna operationen (eller väntar på nytt).

Om explicita synkroniseringslås (av typen `java.util.concurrent.locks.Lock`) används, kan en liknande strategi användas för att implementera tillståndsberoende operationer. Ett särskilt objekt som representerar ett villkor skapas i samband med låset. Detta *villkorsobjekt* är av typen `java.util.concurrent.locks.Condition` (ett gränssnitt), och erhålls med metoden `newCondition` (som definieras i gränssnittet `java.util.concurrent.locks.Lock`.) utifrån det aktuella synkroniseringslåset (metoden returnerar ett objekt av en klass som implementerar gränssnittet `Condition`). Villkorsobjektet används sedan som en väntepunkt för de trådar som väntar på det villkor som villkorsobjektet representerar. Trådarna blockeras och väntar i metoden `await` (i gränssnittet `Condition`), och aktiveras med metoden `signalAll` (eller `signal`, om bara en tråd ska aktiveras). Metoderna `await` och `signalAll` beter sig som metoderna `wait` och `notifyAll` i klassen `Object`. Metoden `await` överlagras, så att den maximala väntetiden kan preciseras.

 © FADIL GALJIĆ

Olika villkorsobjekt kan skapas i samband med ett och samma synkroniseringslås (genom att anropa metoden `newCondition` flera gånger). Ett villkorsobjekt representerar ett villkor, och används som väntepunkt för de trådar som väntar på detta villkor. Tack vare det behöver inte alla trådar aktiveras vid varje ändring av motsvarande objekt. Bara de trådar som kan dra nytta av en objektmodifiering aktiveras. På så sätt optimeras synkroniseringen. Denna möjlighet finns inte om de inbyggda låsen (och `syncronised`-metoder och `synchronized`-block) används.

Användningen av olika villkorsobjekt kan illustreras så här:

```
import java.util.concurrent.locks.*;

class Square
{
    public static final int    MIN = 1;
    public static final int    MAX = 10;

    private int    length = 5;

    private Lock    lock = new ReentrantLock ();

    private Condition    canBeIncreased = lock.newCondition ();
    private Condition    canBeDecreased = lock.newCondition ();


    public Square ()
    {}


    public int getLength ()
    {
        lock.lock ();
        int    len = this.length;;
        lock.unlock ();

        return len;
     }

    public void increase (int increment)
                           throws InterruptedException
    {
        lock.lock ();
        increment = Math.abs (increment);

        try
        {
            while (this.length + increment > MAX)
                canBeIncreased.await ();
```

```
            this.length += increment;

            canBeDecreased.signalAll ();
        }
        finally
        {
            lock.unlock ();
        }
    }

    public void decrease (int decrement)
                          throws InterruptedException
    {
        lock.lock ();
        decrement = Math.abs (decrement);

        try
        {
            while (this.length - decrement < MIN)
                canBeDecreased.await ();
            this.length -= decrement;

            canBeIncreased.signalAll ();
        }
        finally
        {
            lock.unlock ();
        }
     }
}
```

Klassen `Square` definierar en kvadrat vars sida kan variera mellan två givna gränser (`MIN` och `MAX`). Metoden `increase` ökar kvadratens sida, och metoden `decrease` minskar sidan. De båda operationerna (ökningen och minskningen) är tillståndsberoende operationer, som kan bara utföras om givna gränser inte ska överskridas.

Två villkor definieras i klassen, och dessa villkor representeras med två olika villkorsobjekt (av typen `Condition`). Objektet `canBeIncreaseed` representerar en väntepunkt för de trådar som väntar på att öka kvadraten. Dessa trådar aktiveras med metoden `signalAll` i metoden `decrease`. Objektet `canBeDeceased` representerar en väntepunkt för de trådar som väntar på att minska kvadraten. Dessa trådar aktiveras med metoden `signalAll` i metoden `increase`. På det här sättet skapas en förnuftig modell. De trådar som väntar på att öka kvadraten aktiveras inte efter kvadratsökningen. När en tråd ökat kvadraten, blir situationen för de trådar som väntar på att öka kvadraten inte bättre. De ska inte aktiveras. Situationen blir eventuellt bättre för de trådar som väntar på att minska kvadraten,

© FADIL GALJIĆ

och det är bara dessa trådar som ska aktiveras. På samma sätt ska inte de trådar som väntar på att minska kvadraten aktiveras efter att en tråd utfört minskningsoperationen. De ska aktiveras när en tråd utfört ökningen.

Genom att använda olika villkorsobjekt, kan de trådar som ska aktiveras väljas med en större noggrannhet. Alla väntande trådar behöver inte aktiveras efter varje ändring av den gemensamma resursen. På så sätt förbättras synkroniseringen.

© FADIL GALJIĆ

Kommunikation mellan trådar

Objektöverföring via en kanal

En kommunikationskanal

Två trådar i ett program kan utbyta olika uppgifter med varandra. En tråd kan komma fram till en uppgift, som av olika anledningar kan behövas även i en annan tråd. I så fall kan tråden skicka denna uppgift till den andra tråden.

Ett primitivt värde eller ett objekt kan skickas från en tråd till en annan tråd. När ett objekt skickas, skickas både data och beteende. När ett objekt tagits emot, kan objektets olika tjänster användas. Genom att skicka olika typer av objekt, skickar man olika typer av data och olika beteenden. Man kan även skicka ett `Runnable`-objekt (ett objekt vars klass implementerar gränssnittet `Runnable`) till en tråd. I så fall kan en ny tråd skapas när objektet tas emot, och den kod som finns i objektets `run`-metod kan exekveras parallellt med andra aktiviteter.

För att primitiva värden eller objekt ska kunna skickas från en tråd till en annan tråd, krävs det en *kanal* mellan trådarna. Denna kanal är en plats som kan lagra primitiva värden eller objekt. En tråd kan placera primitiva värden eller objekt på denna plats, och en annan tråd kan sedan ta dessa värden eller objekt från platsen.

För att kunna skapa en kommunikationskanal, krävs det en klass som definierar en typ av kommunikationskanaler. I denna klass ska finnas en lagringsstruktur, där de data som överförs kan lagras. Klassen ska definiera de operationer som gör det möjligt att placera data i kanalen, och att ta dessa data från kanalen.

Man kan definiera en kanal för överföring av objekt mellan två trådar. En sådan kanal är användbar även i samband med primitiva värden, eftersom primitiva värden kan packas in i motsvarande objekt (ett värde av typen `int`, till exempel, kan packas in i ett objekt av typen `Integer`). För att ett objekt ska kunna placeras i en kanal, krävs det en operation som gör detta möjligt. Denna operation kan kallas för `put`. Det behövs även en operation för att kunna ta ett objekt från kanalen. Denna operation kan heta `take`. I så fall kan en kanal definieras med ett gränssnitt. Detta gränssnitt kan heta `Channel`, och definieras så här:

```
interface Channel<T>
{
```

 © FADIL GALJIĆ

```
    void put (T object) throws InterruptedException;
    T take () throws InterruptedException;
}
```

En kanal kan ha en viss kapacitet (antalet objekt som samtidigt kan lagras i kanalen). Om en kanal är full, kan inte ett objekt placeras i den. Operationen `put` kan inte slutföras. Den måste vänta tills någon annan tråd tar ett objekt från kanalen, och på så sätt skapar plats för ett annat objekt. Under vissa omständigheter måste alltså operationen `put` vänta. Operationen ska vänta antingen i metoden `wait` i klassen `java.lang.Object`, eller i metoden `await` i (en klass som implementerar) gränssnittet `java.util.concurrent.locks.Condition`. Eftersom dessa metoder kan kasta ett undantag av typen `java.lang.InterruptedException`, behöver detta undantag deklareras i metoden `put` (man lämnar hanteringen av undantaget till en mer specifik applikation). Det kan också inträffa att metoden `take` måste vänta. Det går inte att ta ett objekt från en tom kanal. Alltså måste någon av metoderna `wait` eller `await` anropas även i metoden `take`, och därför deklareras undantag av typen `InterruptedException` även i metoden `take`.

När en klass som definierar en typ av kanaler skapas, kan man utgå från gränssnittet `Channel`. En lämplig lagringsstruktur väljs, och metoderna `put` och `take` implementeras. Man kan till exempel skapa en klass som definierar en kanal, som kan innehålla endast ett objekt vid ett visst tillfälle (en kanal vars kapacitet är 1). Kanaler vars kapacitet kan justeras, och kanaler vars kapacitet inte är begränsad, kan också definieras. Om en kanals kapacitet inte är begränsad behöver inte operationen `put` vänta, och därmed behöver inte undantag av typen `InterruptedException` deklareras i denna operation. Att en undantagstyp anges i en metod i ett gränssnitt, innebär inte att undantaget måste deklareras i varje klass som implementerar gränssnittet. Genom att deklarera en undantagstyp i en metod i ett gränssnitt, öppnar man möjlighet för att denna undantagstyp kan finnas (om den behövs) i konkreta implementeringar av metoden.

Genom att först definiera en kanal i ett gränssnitt, skapas möjligheten att använda referenser av detta gränssnitt i olika program. En sådan referens kan referera till vilken kanal som helst, så länge denna kanal implementerar gränssnittet. Denna referens kan aktivera metoderna `put` och `take`. Om man vill byta ut en kanal i ett program mot en kanal av en annan typ (men som implementerar gränssnittet), låter man bara referensen referera till den nya kanalen. Resten av programmet behöver inte ändras. Genom att använda referenser av ett gränssnitt, kan man skapa oberoende programenheter.

© FADIL GALJIĆ

En objektbehållare

En klass som definierar en behållare för ett objekt kan skapas. En sådan behållare kan bara innehålla ett objekt vid ett visst tillfälle. Ett objekt kan placeras i en *objektbehållare*, och det kan tas från denna objektbehållare. Behållarklassen kan kallas för `ObjectContainer`, och definieras och implementeras på följande vis:

```
class ObjectContainer<T> implements Channel<T>
{
    private T    object;
    private boolean    full;

    public ObjectContainer ()
    {
        object = null;
        full = false;
    }

    public synchronized void put (T object)
                              throws InterruptedException
    {
        while (full)
            this.wait ();

        this.object = object;

        full = true;
        this.notifyAll ();
    }

    public synchronized T take ()
                            throws InterruptedException
    {
        while (!full)
            this.wait ();

        full = false;
        this.notifyAll ();

        return object;
    }
}
```

Instansvariabeln `object` i denna klass representerar en referens till det objekt som lagras i objektbehållaren. Den booleska variabeln `full` indikerar om det finns något objekt i objektbehållaren (objektbehållaren full) eller inte (objektbehållaren inte full – den är tom). Denna variabel är `true` om det finns ett objekt i objektbehållaren, annars är variabeln `false`.

© FADIL GALJIĆ

Klassen `ObjectContainer` implementerar gränssnittet `Channel`, vilket innebär att klassen implementerar metoderna `put` och `take`. Metoden `put` placerar ett givet objekt i objektbehållaren genom att sätta behållarens referens att referera till detta objekt (en objektbehållare har ett objekt). Innan ett objekt placeras i objektbehållaren kontrolleras om behållaren är full. Om så är fallet väntar metoden. När ett objekt har placerats i objektbehållaren sätts variabeln `full` till `true`, för att indikera att objektbehållaren är full. Sedan aktiveras (med metoden `notifyAll`) alla de trådar som väntar i objektbehållarens `wait`-mängd, eftersom det kan finnas trådar som väntar på att ett objekt ska placeras i objekthållaren för att kunna ta det.

Metoden `take` returnerar det objekt som finns i objektbehållaren. Innan den gör det, kontrollerar den om det finns något objekt i objektbehållaren (att den inte är tom). Variabeln `full` sätts till `false`, för att indikera att objektbehållaren är tom (tom efter det att metoden `take` har avslutats – denna metod är synkroniserad, så att ingen annan tråd kan komma in i den metoden innan metoden avslutas i den exekverande tråden). Sedan aktiveras (med metoden `notifyAll`) alla de trådar som väntar i objektbehållarens `wait`-mängd, eftersom vissa av dessa trådar kanske väntar på att objektbehållaren blir tom, för att kunna placera ett objekt där.

Ett objekt av typen `ObjectContainer` kan användas som kanal för objektöverföring mellan olika trådar i ett program. En tråd kan placera ett objekt i en sådan kanal, och en annan tråd kan ta objektet från denna kanal. Denna kanal har kapaciteten `1`, eftersom den bara kan lagra ett objekt åt gången.

Kommunikation mellan två trådar

Klassen `EnsiffrigtHeltal` representerar ett ensiffrigt heltal. Denna klass hanterar ett heltals numeriska värde och namn. Klassen innehåller metoden `set`, som tar emot ett heltalsvärde och sätter det ensiffriga heltalets numeriska värde och namn utifrån detta heltalsvärde. Klassen kan även innehålla metoden `setNamn` som sätter heltalets namn till ett givet namn, metoden `getVarde` som returnerar heltalets numeriska värde, och metoden `toString` som returnerar en sträng som innehåller heltalets numeriska värde och dess namn.

Klassen `HeltalsGenerator` definierar en tråd som genererar slumpmässiga ensiffriga heltal. Klassen `HeltalsOversattare` definierar en tråd som översätter ett ensiffrigt heltal (översätter heltalets namn) till ett annat språk, till exempel engelska. En tråd av typen `HeltalsGenerator` skickar de en-

© FADIL GALJIĆ

siffriga heltalen som den genererar till en tråd av typen `HeltalsOversattare`. Denna översättningstråd tar emot heltalen och översätter dem. För att kunna skicka de genererade heltalen, har en tråd av typen `HeltalsGenerator` tillgång till en kommunikationskanal. För att kunna ta emot de skickade heltalen har även en tråd av typen `HeltalsOversattare` tillgång till en kommunikationskanal.

Klassen `HeltalsGenerator` kan implementeras så här:

```
class HeltalsGenerator implements Runnable
{
    private EnsiffrigtHeltal    heltal;
    private Channel<EnsiffrigtHeltal>    kanal;

    public HeltalsGenerator (Channel<EnsiffrigtHeltal> kanal)
    {
        this.kanal = kanal;
    }

    public void run ()
    {
        for (int i = 0; i < 5; i++)
        {
            int    p = (int) (10 * Math.random ());
            heltal = new EnsiffrigtHeltal ();
            heltal.set (p);
            System.out.println (heltal);

            try
            {
                kanal.put (heltal);
            }
            catch (InterruptedException e)
            {}
        }
    }
}
```

Ett slumpmässigt heltal genereras och visas. Därefter skickas heltalet till en annan tråd genom att det placeras i en kanal. Detta upprepas ett antal gånger.

Ett objekt skickas genom att en referens till objektet skickas (`heltal` är en referens till det genererade heltalet). På så vis får mottagartråden en referens till det skickade objektet. Detta innebär att både sändartråden och mottagartråden har tillgång till ett och samma objekt. Mottagartråden kan påverka det skickade objektet via den motsvarande referensen. Översättningstråden kan till exempel översätta det genererade heltalets namn till ett annat språk. Om man vill förhindra modifiering av det objekt som

© FADIL GALJIĆ

skickas, måste man först skapa en kopia av objektet, och sedan skicka en referens till denna kopia.

Klassen `HeltalsOversattare` definierar en översättningstråd. Klassen kan definieras och implementeras så här:

```
class HeltalsOversattare implements Runnable
{
    private EnsiffrigtHeltal    heltal;
    private Channel<EnsiffrigtHeltal>    kanal;

    public HeltalsOversattare (Channel<EnsiffrigtHeltal> kanal)
    {
        this.kanal = kanal;
    }

    public void run ()
    {
        for (int i = 0; i < 5; i++)
        {
            try
            {
                heltal = kanal.take ();
            }
            catch (InterruptedException e)
            {}

            int    v = heltal.getVarde ();
            String    n = null;
            int    absv = Math.abs (v);
            switch (absv)
            {
            case 0:
                n = "zero";
                break;
            case 1:
                n = "one";
                break;
            case 2:
                n = "two";
                break;
            case 3:
                n = "three";
                break;
            case 4:
                n = "four";
                break;
            case 5:
                n = "five";
                break;
            case 6:
```

```
                n = "six";
                break;
            case 7:
                n = "seven";
                break;
            case 8:
                n = "eight";
                break;
            case 9:
                n = "nine";
                break;
            default:
                n = "not correct value";
                break;
            }
            if (v < 0)
                n = "minus " + n;
            heltal.setNamn (n);

            System.out.println (heltal);
        }
    }
}
```

Ett skickat ensiffrigt heltal tas från kanalen. Heltalets numeriska värde bestäms, och utifrån detta värde bestäms heltalets nya namn (på engelska). På så sätt översätts det erhållna heltalet (sändaren och mottagaren har båda tillgång till samma heltal). Därefter visas heltalet. Detta upprepas ett antal gånger.

Man kan skapa en tråd som producerar ensiffriga heltal, en tråd som översätter ensiffriga heltal, och en kanal emellan dem. På så vis kan generatortråden skicka sina ensiffriga heltal till översättningstråden. Som kanal mellan trådarna kan man använda ett objekt av en klass som implementerar gränssnittet `Channel`, till exempel ett objekt av typen `ObjectContainer`:

```
class ObjectOverforingViaEnKanal
{
    public static void main (String[] args)
    {
        Channel<EnsiffrigtHeltal>    kanal =
                    new ObjectContainer<EnsiffrigtHeltal> ();

        Thread   t1 = new Thread (new HeltalsGenerator (kanal));
        Thread   t2 = new Thread (new HeltalsOversattare (kanal));

        t1.start ();
        t2.start ();
    }
}
```

 © FADIL GALJIĆ

Trådarna `t1` och `t2` har tillgång till en och samma kanal. Översättningstråden tar från denna kanal de ensiffriga heltal som placerar där av generatortråden. På så sätt överförs heltalen från generatortråden till översättningstråden.

När programmet `ObjektoverforingViaEnKanal` exekveras, får man en utskrift med de genererade heltalen och deras översättningar. Utskriften kan se ut så här:

```
[1, ett]
[-4, minus fyra]
[1, one]
[7, sju]
[-4, minus four]
[6, sex]
[7, seven]
[3, tre]
[6, six]
[3, three]
```

Generatortråden genererar heltalen `1, -4, 7, 6` och `3` och skriver ut dessa heltal. Översättningstråden översätter heltalen och skriver ut dem efter översättningen.

En buffert som kanal

En kanal av typen `ObjectContainer` kan användas för att överföra objekt från en tråd till en annan tråd. En kanal av denna typ kan bara innehålla ett objekt (som är i transit från en tråd till en annan tråd) åt gången. Denna kanal kan inte användas för att placera flera objekt vid ett och samma tillfälle. I vissa fall kan det behövas en kanal vars kapacitet är större än 1. En tråd (eller flera trådar) kan placera sina objekt i en sådan kanal, innan mottagartråden (eller mottagartrådarna) hinner ta och använda dem. En tråd som vill skicka ett objekt behöver inte vänta tills mottagartråden (eller mottagartrådarna) tar alla objekt från kanalen. En tråd kan lämna sitt objekt i kanalen och fortsätta med sin uppgift. En annan tråd kan eventuellt ta detta objekt vid ett senare tillfälle.

När man definierar en kanal, anger man en lagringsstruktur där de objekt som överförs lagras. En vektor av den inbyggda typen kan användas som lagringsplats för dessa objekt. Med metoden `put` placeras ett objekt i vektorn, och med metoden `take` tas ett objekt från vektorn. På så sätt skapas en ändlig buffert (en buffert med en bestämd kapacitet).

Metoderna `put` och `take` kan definieras på så sätt att objekten placeras i bufferten och tas från bufferten i samma ordning. Det objekt som placeras först ska tas först. I en sådan buffert placeras objekten från index `0` mot största index, och sedan börjar man åter med index `0`. Objekten tas i samma ordning. På så sätt skapas en *cirkulär buffert*. I denna buffert hanteras två index. Ett index betecknar den plats där nästa objekt ska placeras. Ett annat index betecknar den plats där det objekt som ska först tas finns. I början är båda indexen `0`.

Det är inte nödvändigt att begränsa buffertens kapacitet. Om man använder en vektor av typen `java.util.ArrayList` (istället för en vektor av den inbyggda typen) som lagringstruktur i en kanal, behöver man inte fastställa kanalens kapacitet. Längden på en sådan struktur kan anpassas efter behov. Metoden `put` kan baseras på metoden `add` i klassen `ArrayList`. Denna metod placerar ett givet objekt som sista objekt i vektorn. Metoden `take` kan baseras på metoderna `get` och `remove` i klassen `ArrayList`. Först får man tillgång till det objekt som finns på den första platsen, och sedan tas detta objekt bort från vektorn. På så sätt kan man åstadkomma att objekten placeras i en kanal och tas från denna kanal i samma ordning. Om bufferten är tom, måste operationen `take` vänta. Metoden `isEmpty` i klassen `ArrayList` kan användas för att kontrollera om en vektor är tom.

Klassen `Buffer`, som definierar en buffert med obegränsad kapacitet, kan definieras och implementeras på följande vis:

```
class Buffer<T> implements Channel<T>
{
    private java.util.ArrayList<T>    v;

    public Buffer ()
    {
        v = new java.util.ArrayList<T> ();
    }

    public synchronized void put (T object)
    {
        v.add (object);

        this.notifyAll ();
    }

    public synchronized T take () throws InterruptedException
    {
        while (v.isEmpty ())
            this.wait ();

        T    object = v.get (0);
```

 © FADIL GALJIĆ

```
        v.remove (0);

        return object;
    }
}
```

Eftersom klassen `Buffer` implementerar gränssnittet `Channel`, kan ett objekt av typen `Buffer` användas istället för ett objekt av typen `Object-Container` i programmet `ObejktOverforingViaEnKanal`. I så fall fungerar programmet som tidigare.

Man kan även skapa en kanal där man på ett speciellt sätt väljer det objekt som ska tas först. Särskilda kriterier kan införas för att bestämma de enskilda objektens prioritet.

Istället för att definiera egna gränssnitt och klasser, kan de gränssnitt och klasser som redan finns i standardbiblioteket användas. Gränssnittet `java.util.concurrent.BlockingQueue` definierar en *blockeringskö* som, förutom andra metoder, även har metoderna `put` och `take`. Klassen `java.util.concurrent.ArrayBlockingQueue` representerar en cirkulär buffert, som implementerar gränssnittet `BlockingQueue`. En sådan buffert kan skapas på följande vis:

```
BlockingQueue<EnsiffrigtHeltal>    kanal =
                 new ArrayBlockingQueue<EnsiffrigtHeltal> (10);
```

En blockeringskö (för objekt av typen `EnsiffrigtHeltal`) med kapaciteten `10` skapas. En blockeringskö med obegränsad kapacitet kan representeras med ett objekt av typen `java.util.concurrent.LinkedBlockingQueue`. En sådan kanal använder en länkad lista som lagringsstruktur. En behållare av den typen kan skapas så här:

```
BlockingQueue<EnsiffrigtHeltal>    kanal =
                 new LinkedBlockingQueue<EnsiffrigtHeltal> ();
```

Producenter, konsumenter och mellanstationer

En kanal kan användas för att överföra objekt från en tråd till en annan tråd. En tråd skapar/producerar objekt, och en annan tråd använder/konsumerar dessa objekt. Detta är en kommunikation av typen *producent-konsument*. Flera producenter kan använda en och samma kanal för att tillföra sina objekt till en konsument. Producenterna placerar objekten i kanalen, och konsumenten tar och använder dessa objekt. Man kan även ha en producent och flera konsumenter (som tar de skickade objekten från den gemensamma kanalen), eller flera producenter och konsumenter (som använder en och samma kommunikationskanal).

© FADIL GALJIĆ

Det kan finnas en eller flera stationer mellan en producent och en konsument. En *transformator* kan ta emot de skickade objekten och bearbeta dem på något sätt. Denna transformator skickar sedan de transformerade objekten till konsumenten. I detta fall finns det en kanal mellan producenten och transformatorn, och en kanal mellan transformatorn och konsumenten.

Ett *filter* kan placeras mellan en producent och en konsument. Filtret filtrerar de skickade objekten, och släpper endast igenom de objekt som uppfyller vissa krav.

Man kan placera en *växel* på en kommunikationslinje. Växeln tar emot ett objekt som skickas av producenten, och väljer en av flera möjliga konsumenter. Objektet skickas sedan till den valda konsumenten. I detta fall är varje konsument kopplad till växeln via en särskild kanal. Istället för att skicka ett objekt till en av flera konsumenter, kan man ha en *distributör* som skickar det producerade objektet till samtliga kopplade konsumenter.

En *kollektor* kan placeras på en kommunikationslinje. En kollektor är kopplad till flera producenter via olika kanaler. Kollektorn tar emot ett objekt från en av flera producenter, och skickar detta objekt vidare längs kommunikationslinjen.

Det kan även finnas en *kombinatör* på en kommunikationslinje. En kombinatör är kopplad till flera producenter via olika kanaler. Den tar emot objekt från samtliga producenter och kombinerar dem på något sätt. Ett nytt objekt skapas (utifrån de mottagna objekten) och skickas längs kommunikationslinjen.

Respons från mottagaren

En kontaktpunkt mellan sändaren och mottagaren

En tråd kan placera ett objekt i en kanal och fortsätta med sina aktiviteter. En annan tråd kan senare ta detta objekt och använda det på något sätt. På så sätt kan en tråd skicka ett meddelande till en annan tråd. Två trådar som kommunicerar på detta vis har inte någon gemensam kontaktpunkt. Om det finns flera trådar, kan det hända att sändartråden inte vet vilken tråd som ska ta det skickade meddelandet, och att mottagartråden inte vet vilken tråd som skickat meddelandet. Det finns inte heller någon samordning mellan sändarens och mottagarens aktiviteter. Detta kallas för *asynkron kommunikation*.

© FADIL GALJIĆ

I vissa fall kan sändaren behöva ett svar från mottagaren. Detta svar kan vara en bekräftelse på att mottagaren fått meddelandet och hanterat det på något vis. I andra situationer kan svaret representera något resultat av hanteringen av det erhållna meddelandet. Svaret kan representeras via ett objekt, och detta objekt kan sedan skickas tillbaka till sändaren. I så fall skickar en tråd ett objekt som representerar ett meddelande, och får tillbaka ett annat objekt som representerar svaret på detta meddelande.

För att kunna skicka ett svar tillbaka till sändaren, måste mottagaren ha en kontaktpunkt med denna sändare. Det måste finnas en plats som både sändaren och mottagaren känner till och har kontakt med. Via denna plats kan sändaren skicka ett meddelande och få tillbaka motsvarande svar.

En behållare, som kan innehålla både ett meddelande och motsvarande svar, kan skapas. Sändaren ska kunna placera sitt meddelande i denna behållare, och hämta svaret därifrån. Mottagaren ska kunna hämta meddelandet från behållaren, och placera sitt svar där. Om man vill få ett svar från mottagartråden, ska en sådan behållare användas.

Man kan skapa en klass, som definierar en behållare som kan innehålla både ett meddelande och ett svar. Eftersom en sådan behållare ska användas som en kontaktpunkt mellan sändaren och mottagaren, kan klassen kallas för `Intermediary` (mellanhand). Klassen kan definieras och implementeras så här:

```
class Intermediary<T1, T2>
{
    private T1    message;
    private T2    reply;
    private boolean    replied;

    public Intermediary (T1 message)
    {
        this.message = message;

        this.reply = null;
        this.replied = false;
    }

    public synchronized T1 getMessage ()
    {
        return message;
    }

    public synchronized void setReply (T2 reply)
    {
        this.reply = reply;
```

```
        this.replied = true;
        this.notifyAll ();
    }

    public synchronized T2 getReply () throws InterruptedException
    {
        while (!replied)
            this.wait ();

        return reply;
    }
}
```

Variabeln `message` representerar meddelandet (ett objekt av någon typ `T1`) och variabeln `reply` representerar svaret (ett objekt av någon typ `T2`). Variabeln `replied` är en boolesk variabel, som indikerar om mottagaren svarat eller inte.

Klassen `Intermediary` har en konstruktor, som packar det meddelande som ska skickas i den behållaren som skapas. Mottagaren kan sedan ta meddelandet från behållaren med metoden `getMessage`. När mottagaren får meddelandet, skapar den ett svar. Svaret placeras i behållaren med metoden `setReply`. Sändaren hämtar sedan svaret från behållaren med metoden `getReply`. I denna metod väntar sändaren tills svaret blir klart. Mottagaren indikerar att svaret är klart genom att sätta variabeln `reply` till `true`. Mottagaren aktiverar samtidigt sändaren som (eventuellt) väntar, genom att anropa metoden `notifyAll`.

En synkron kommunikation

När en tråd vill skicka ett meddelande till en annan tråd och få svar tillbaka, packar den först meddelandet i en behållare av typen `Intermediary`. Därefter placeras denna behållare i en kanal. Sändartråden tappar inte kontakten med behållaren, utan behåller en referens till denna. En annan tråd hämtar sedan behållaren från denna kanal. På så vis får även denna tråd en referens till behållaren. En plats skapas därmed, som både sändaren och mottagaren har kontakt med. Mottagaren hämtar meddelandet från behållaren, skapar ett svar och placerar svaret i behållaren. Sändaren hämtar sedan svaret från behållaren.

När sändartråden har placerat behållaren i en kanal, kan den anropa metoden `getReply` i samband med behållaren. Detta får tråden att vänta på ett svar från mottagartråden. Den aktiveras och fortsätter med sina aktiviteter först när den har fått ett svar. På så sätt kan man åstadkomma en

© FADIL GALJIĆ

synkron kommunikation. Sändarens och mottagarens aktiviteter är samordnade. Sändaren väntar på mottagaren, och när meddelandet är färdigbehandlat fortsätter de tillsammans.

Denna teknik kan användas för att synkronisera aktiviteter mellan sändaren och mottagaren, även om sändaren inte behöver något konkret svar från mottagaren. Sändaren kanske bara vill vänta tills mottagaren får och färdigbehandlar meddelandet. I så fall kan mottagaren skicka en `null`-referens som svar. Sändaren kan anropa metoden `getReply` utan att använda svaret på något sätt (svaret behöver inte lagras). På detta vis väntar sändaren bara.

Sändartråden behöver inte anropa metoden `getReply` direkt efter att den placerat behållaren i kommunikationskanalen. Den kan fortsätta med sina aktiviteter och anropa denna metod vid ett senare tillfälle, när den blir beroende av svaret.

Man kan definiera en tråd som genererar slumpmässiga ensiffriga heltal (objekt av typen `EnsiffrigtHeltal`), och skickar dessa heltal till en tråd som översätter dem till ett annat språk. Denna tråd kan vänta på översättningen från översättningstråden, och bara fortsätta när den har fått denna översättning. I detta fall är det ensiffriga heltalet som genereras ett meddelande, och översättningen av detta heltal (som är också ett heltal av typen `EnsiffrigtHeltal`) är ett svar. En sådan generatortråd (sändaren) kan definieras så här:

```
import java.util.concurrent.*;

class HeltalsGenerator implements Runnable
{
    private EnsiffrigtHeltal    heltal;
    private BlockingQueue<
      Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>>    kanal;

    public HeltalsGenerator (BlockingQueue<
         Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>> kanal)
    {
        this.kanal = kanal;
    }

    public void run ()
    {
        Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>    im;
        EnsiffrigtHeltal    oversattning = null;

        for (int i = 0; i < 5; i++)
        {
            int    p = (int) (19 * Math.random () - 9);
```

```
            heltal = new EnsiffrigtHeltal ();
            heltal.set (p);
            System.out.println (heltal);

            im = new Intermediary<
                   EnsiffrigtHeltal, EnsiffrigtHeltal> (heltal);
            try
            {
                kanal.put (im);

                oversattning = im.getReply ();
            }
            catch (InterruptedException e)
            {}

            System.out.println (oversattning);
        }
    }
}
```

De genererade ensiffriga heltalen placeras inte direkt i kommunikationskanalen. Först packas ett ensiffrigt heltal i motsvarande behållare (ett objekt av typen `Intermediary`), och denna behållare placeras i kanalen (av typen `java.util.concurrent.BlockingQueue`). Tråden behåller kontakten med behållaren via referensen `im`, och kan när som helst begära ett svar med metoden `getReply`. Översättningstråden producerar översättningen, som är ett nytt objekt av typen `EnsiffrigtHeltal`, och lägger den i behållaren (den gemmensamma punkten mellan trådarna). Översättningen används sedan av generatortråden (visas på standardutmatningsenheten).

Den tråd som utför översättningen kan definieras så här:

```
import java.util.concurrent.*;

class HeltalsOversattare implements Runnable
{
    private EnsiffrigtHeltal    heltal;
    private BlockingQueue<
      Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>>    kanal;

    public HeltalsOversattare (BlockingQueue<
       Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>> kanal)
    {
        this.kanal = kanal;
    }

    public void run ()
    {
        Intermediary<EnsiffrigtHeltal, EnsiffrigtHeltal>    im;
        EnsiffrigtHeltal    oversattning = null;
```


© FADIL GALJIĆ

```
for (int i = 0; i < 5; i++)
{
    try
    {
        im = kanal.take ();
        heltal = im.getMessage ();
    }
    catch (InterruptedException e)
    {}

    int    v = heltal.getVarde ();
    String    n = null;
    int    absv = Math.abs (v);
    switch (absv)
    {
    case 0:
        n = "zero";
        break;
    case 1:
        n = "one";
        break;
    case 2:
        n = "two";
        break;
    case 3:
        n = "three";
        break;
    case 4:
        n = "four";
       break;
    case 5:
        n = "five";
        break;
    case 6:
        n = "six";
        break;
    case 7:
        n = "seven";
        break;
    case 8:
        n = "eight";
        break;
    case 9:
        n = "nine";
        break;
    default:
        n = "not correct value";
        break;
    }
    if (v < 0)
```

```
                n = "minus " + n;

            oversattning = new EnsiffrigtHeltal ();
            oversattning.set (v);
            oversattning.setNamn (n);

            im.setReply (oversattning);
        }
    }
}
```

Översättningstråden (mottagaren) hämtar först den skickade behållaren från kommunikationskanalen. Tråden extraherar sedan det skickade heltalet (meddelandet) från behållaren med metoden `getMessage`. Därefter översätts heltalet, och ett nytt heltal som representerar översättningen skapas (ett nytt objekt skapas, det skickade objektet ändras inte). Sedan placeras översättningen (svaret) i behållaren (med metoden `setReply`), så att sändaren kan hämta den.

Man kan skapa en tråd av typen `HeltalsGenerator`, en tråd av typen `HeltalsOversattare`, och en kanal (av typen `java.util.concurrent.BlockingQueue`) mellan generatortråden och översättningstråden. När dessa trådar startas, genererar generatortråden slumpmässiga ensiffriga heltal, visar dem och skickar dem till översättningstråden. Översättningstråden tar heltalen, översätter dem och skickar översättningarna tillbaka till generatortråden. Generatortråden tar emot översättningarna och visar dem. Eftersom generatortråden väntar på översättningstråden, är detta en synkroniserad aktivitet. Generatortråden genererar inte ett nytt heltal innan den har fått och visat motsvarande svar. En utskrift som har följande form skapas:

```
[2, två]
[2, two]
[6, sex]
[6, six]
[-2, minus två]
[-2, minus two]
[0, noll]
[0, zero]
[1, ett]
[1, one]
```


© FADIL GALJIĆ

Kapitel 2

Kommunikation mellan program

Kommunikation mellan två program
Ett nät av datorer
Ansluta ett Javaprogram till en server
Kommunikation mellan två Javaprogram

Ett serverprogram
En flertrådad server
En tråd per klient
En pool av trådar
Kommunikation via en server

© FADIL GALJIĆ

Kommunikation mellan två program

Ett nät av datorer

Datorer i ett nät

Flera datorer kan kopplas samman i ett nät. Via detta nät kan datorerna kommunicera med varandra. Ett program som exekveras i en dator kan skicka olika uppgifter till ett program som exekveras i en annan dator, och ta emot uppgifter från det andra programmet. Från en dator kan man komma åt uppgifter som är lagrade i en annan dator. Ett program på en dator kan använda olika tjänster som (olika program i) en annan dator erbjuder.

Det finns ett stort nät av datorer, som sträcker sig över hela världen. Detta är Internet. Ett stort antal datorer är anslutna till detta nät. Via Internet kan en dator i ett land kommunicera med en annan dator, som finns i ett annat land. Ett stort antal datorer i detta nät är aktiva hela tiden. Man kan i en dator köra ett program som tillför en tjänst till olika användare (andra program i andra datorer i nätet). En sådan dator och ett sådant program kallas för *server*. De datorer som utnyttjar tjänsten kallas för *klienter*. Även motsvarande program kallas för klienter. En och samma serverdator kan köra flera program samtidigt. Man kan till exempel köra ett program som hanterar elektronisk post, och ett program som skickar olika filer från det aktuella filsystemet till klienterna.

Ett Javaprogram kan utnyttja olika tjänster som finns tillgängliga på Internet. Ett Javaprogram kan via Internet kommunicera med ett annat Javaprogram. Man kan skapa olika Javaapplikationer som kommunicerar med varandra. Tack vare Internet kan ett Javaprogram nå andra program, som samtidigt körs på andra datorer, oavsett hur avlägsna dessa datorer är.

Identifiera en dator i nätet

I ett nät kan det finnas en dator som ett visst program körs på. En sådan dator brukar kallas för *värddator* (eng. host). Om man vill nå detta pro-

 © FADIL GALJIĆ

gram från en annan dator, måste man på något sätt identifiera denna dator och programmet. En och samma dator kan köra flera olika program, och därför måste man precisera det program som man vill kommunicera med.

Varje dator i Internet har en egen adress. Denna adress består av fyra byte, och anges vanligtvis på formen `192.121.232.103`. Med fyra byte kan många olika adresser skapas. Men antalet datorer i Internet ökar ständigt, och fler och fler adresser behövs. Därför finns det nu möjlighet att använda sexton byte per adress. En Internetadress kallas även för IP-adress (IP = Internet Protocol). En IP-adress är unik och preciserar en viss dator i nätet.

Internetadresser lämpar sig inte för att användas i vardagliga sammanhang. Det är lättare att ange en dator via ett namn. Därför namnger man de olika datorerna i nätet. Namn som till exempel `java.sun.com`, `www.ams.se` och `www.kth.se` används.

En Internetadress kan representeras med ett objekt av klassen `java.net.InetAddress` (klassen har två subklasser: `Inet4Address` och `Inet6Address`). Denna klass saknar konstruktorer, men man kan använda motsvarande statiska metoder för att erhålla ett objekt av denna typ. Ett objekt av typen `InetAddress` kan fås utifrån en dators namn. Detta namn anges som argument till metoden `getByName`:

```
InetAddress    adress = InetAddress.getByName ("www.ams.se");
```

Man kan få adressen som en sträng med metoden `getHostAddress`:

```
String    adressString = adress.getHostAddress ();
```

Detta ger strängen `192.121.232.103`

Metoden `getByName` kastar ett undantag av typen `java.net.UnknownHostException` (en subtyp till `java.io.IOException`), om ett olämpligt namn anges som argument. Detta är ett kontrollerat undantag, som antingen måste fångas och hanteras, eller deklareras.

Internetadressen för den aktuella (lokala) datorn kan erhållas med metoden `getLocalHost`:

```
InetAddress    adress = InetAddress.getLocalHost ();
```

Datorns namn kan sedan erhållas med metoden `getHostName`. Med metoden `getHostAddress` kan man även få datorns adress i form av en sträng.

Vissa platser på Internet används så ofta, att flera datorer (som utför samma funktion) behöver placeras bakom ett och samma namn. I så fall svarar flera adresser mot detta namn. Dessa adresser erhålls med metoden

© FADIL GALJIĆ

`getAllByName`. Denna metod returnerar en vektor med objekt av typen `InetAddress`. Metoden kan användas så här:

```
InetAddress[]    ia = InetAddress.getAllByName ("java.sun.com");
```

Identifiera ett program på en given dator

En viss dator i nätet kan identifieras med motsvarande adress. Men detta är inte tillräckligt för att identifiera ett program på en dator. Det kan finnas flera aktiva program på en och samma dator, och man måste välja ett av dessa program. För att kunna göra detta har begreppet *port* införts. En port är en adress i en viss dator, som representeras med ett heltal mellan `0` (inklusive) och `65535` (inklusive). Varje program som är tänkt att användas från nätet tilldelas en unik port. På så vis går det att inte bara lokalisera en viss dator, utan också ett specifikt program i denna dator.

För vissa typiska tjänster används alltid fastställda portar. Port `80`, till exempel, används för http-kommunikation (en webbsida hämtas via denna port), port `21` för ftp-kommunikation (en fil hämtas via denna port) och port `13` för aktuellt datum och tid. Portarna upp till `1024` har reserverats för olika standardtjänster. När man skapar ett program som ska användas via Internet och som inte tillför någon standartjänst, ska man använda portar över `1024`.

Ett Javaprogram kan anknytas till en viss port på en fjärrdator, och kommunicera med det program som körs på den porten. En viss tjänst kan begäras, eller så kan olika informationer utväxlas. För att kunna anknyta till ett program, anges adressen till den dator som programmet körs på, och den port på datorn på vilket programmet körs.

Ansluta ett Javaprogram till en server

Ansluta till en server

Man kan skapa ett Javaprogram som ansluter sig till en server på Internet, och utnyttjar denna servers tjänst. Ett program kan, till exempel, ansluta sig till en webbserver, begära en `html`-fil (en fil som beskriver en webbsida), och ta emot denna fil från servern.

En anslutning till ett aktivt program på en annan dator skapas med ett objekt av typen `java.net.Socket`. Ett sådant objekt kallas för *socket*. När ett objekt av denna typ skapas, anges fjärrdatorn (antingen via datorns

 © FADIL GALJIĆ

namn eller via ett objekt av typen `InetAddress`) och porten för det program som anslutningen gäller. Så här kan man göra:

```
Socket    s = new Socket ("www.kth.se", 80);
```

Konstruktorn skapar en anslutning till datorn `www.kth.se` på port `80` (som är en standardport – används för http-kommunikation). När koden i konstruktorn exekveras, skapas en förbindelse till den angivna datorn på den angivna porten. Den underliggande programvaran och det befintliga nätet används för att upprätta en förbindelse. Konstruktorn kastar ett undantag av typen `java.net.UnknownHostException` om ett olämpligt namn anges, och ett undantag av typen `java.io.IOException` om en förbindelse av någon anledning inte kan skapas.

Begära en fil

När en anslutning till ett annat program har skapats, måste ett meddelande skickas till detta program. Detta meddelande måste utformas så att programmet kan förstå det. Man måste följa ett i förväg bestämt sätt att kommunicera, ett fastställt *protokoll*. Ett protokoll reglerar vilka uppgifter som ska skickas och i vilken ordning, och hur dessa uppgifter behöver formateras.

En `html`-fil begärs från en webbserver med ett meddelande av följande form:

```
GET /fil HTTP/1.0
```

Efter detta meddelande skickas en tom rad. Meddelandet måste formateras på detta sätt, annars kan inte webbserver förstå det. Ordet `GET` anger att det är en fil som begärs. Detta följs av filens namn och protokollversionen. Här anges att det är version `1.0` av `HTTP`-protokollet som ska följas.

Ett textmeddelande måste alltså skickas till webbservern. Data skickas till ett annat program via en lämplig ström. Ett objekt av typen `Socket` innehåller både en utström och en inström, och dessa strömmar kan användas för kommunikation. Strömmarna erhålls med motsvarande metoder i klassen `Socket`. Utströmmen i objektet `s` kan extraheras så här:

```
OutputStream    os = s.getOutputStream ();
```

Metoden returnerar (en referens till) ett objekt av en icke-abstrakt subklass till klassen `java.io.OutputStream`, och referensen `os` sätts att referera till detta objekt. Via denna referens kan metoderna i (den abstrakta) klassen `OutputStream` aktiveras. Metoden `write` kan användas för att mata ut en

© FADIL GALJIĆ

serie byte. För att kunna skicka primitiva värden till det andra programmet, måste en dataström skapas utifrån strömmen `os`:

```
DataOutputStream    dos = new DataOutputStream (os);
```

För att kunna överföra objekt (av en klass som implementerar gränssnittet `java.io.Serializable`), skapar man en objektström. Detta kan göras så här:

```
ObjectOutputStream    oos = new ObjectOutputStream (os);
```

För att effektivisera dataöverföringen skapar man buffrade strömmar, på samma sätt som i samband med filer.

Vid textöverföring skapas en lämplig teckenström utifrån strömmen `os`. Man kan, till exempel, skapa ett objekt av typen `java.io.PrintWriter`:

```
PrintWriter    out = new PrintWriter (os, true);
```

Objektet `out` (av typen `PrintWriter`) och dess metod `println` kan användas för att skicka motsvarande meddelande (en sträng och en tom rad) till webbservern:

```
out.println ("GET /index.html HTTP/1.0");
out.println ();
```

När webbservern får detta meddelande (och analyserar det), hittar den filen `index.html` och skickar den tillbaka till Javaprogrammet.

När buffrad utmatning används, måste utmatningsbufferten tömmas då och då (det som finns i bufferten ska skickas till destinationsprogrammet). Annars kan det hända att destinationsprogrammet inte får vissa uppgifter i rätt tid, eller att vissa uppgifter inte skickas alls. Utmatningsbufferten töms med metoden `flush`. Denna metod finns både i `OutputStream`-hierarkin och i `Writer`-hierarkin.

Om `true` anges som andra argument när ett objekt av typen `PrintWriter` skapas, töms utmatningsbufferten vid varje anrop till metoden `println`. Om metoden `print` används, måste även metoden `flush` användas (annars skickas inte data från bufferten). I detta fall skickas meddelandet till webbservern på detta vis:

```
out.print ("GET /index.html HTTP/1.0\n\n");
out.flush ();
```


© FADIL GALJIĆ

Ta emot svaret

För att kunna läsa de uppgifter som skickas från ett annat program, krävs det en lämplig inström. Ett objekt av typen `Socket` (som representerar en viss förbindelse) innehåller en grundläggande inström. Denna ström kan extraheras med metoden `getInputStream` (i klassen `Socket`):

```
InputStream   is = s.getInputStream ();
```

Metoden returnerar (en referens till) ett objekt av en icke-abstrakt subklass till klassen `java.io.InputStream`, och referensen `is` sätts att referera till detta objekt. Via denna referens kan metoderna i (den abstrakta) klassen `InputStream` aktiveras. Metoden `read` kan användas för att läsa en serie byte. För att kunna läsa in primitiva värden från det andra programmet, måste man skapa en dataström (ett objekt av typen `DataInputStream`) utifrån strömmen `is`. För att läsa in objekt från det andra programmet, skapas en objektström (ett objekt av typen `ObjectInputStream`) utifrån strömmen `is`. Om en serie tecken ska läsas in, skapas en lämplig teckenström.

Vid begäran av en fil skickar webbservern först en header (information om servern, datum och tid, filens typ, filens längd och så vidare) och en tomrad. Därefter skickas själva filen. Alla dessa uppgifter bildar en serie tecken, fördelade över flera rader. Dessa rader kan läsas in med ett objekt av typen `BufferedReader`. Ett sådant objekt kan skapas utifrån strömmen `is` på följande vis:

```
BufferedReader    in = new BufferedReader (
                           new InputStreamReader (is));
```

Så här kan det som skickas från webbservern läsas och skrivas ut till standardutmatningsenheten:

```
String    rad = in.readLine ();
while (rad != null)
{
    System.out.println (rad);
    rad = in.readLine ();
}
```

Man kan även använda ett objekt av typen `java.util.Scanner` (och dess metoder `nextLine`, `next`, `nextInt`, `nextDouble` och så vidare) för att mata in olika uppgifter som skickas (som en serie tecken) från ett annat program. Ett sådant objekt skapas i så fall så här:

```
Scanner    in = new Scanner (is);
```

© FADIL GALJIĆ

Genom att skapa ett objekt av typen `Socket`, skapar man en förbindelse till ett annat program. Samtidigt skapas verktyg för kommunikation över denna förbindelse. Detta verktyg finns i det objekt (av typen `Socket`) som representerar förbindelsen. Verktyget tas med metoderna `getOutputStream` och `getInputStream`, och en grundläggande utström och en grundläggande inström erhålls. Dessa strömmar kan användas för att mata ut en serie byte till det avlägsna programmet, och till att mata in en serie byte från detta program. För att kunna överföra data av andra typer, måste man skapa mer passande strömmar utifrån dessa grundläggande strömmar. Strömmar för kommunikation via ett nätverk används på samma sätt som vid utmatning till en fil och inmatning från en fil. Samma metoder och samma strategier används.

Data överförs sekventiellt mellan två olika program, uppgift efter uppgift. Det ena programmets utdata blir det andra programmets indata, och vice versa. Kommunikationen baseras på TCP/IP-protokollet, som garanterar att de uppgifter som skickas från ett program når fram till det andra programmet, och att de når det andra programmet i den ordning som de skickades i. Om ett fel uppstår vid överföringen, meddelas detta fel (motsvarande metoder kastar undantag av typen `java.io.IOException`).

Stänga förbindelsen

Efter att kommunikationen med ett program avslutats enligt det gällande protokollet, måste förbindelsen stängas. Detta frigör de system- och nätverksresurser som används under kommunikationen. En förbindelse till ett annat program stängs genom att metoden `close` (i klassen `Socket`) anropas i samband med den socket (det objekt av typen `Socket`) som representerar förbindelsen:

```
s.close ();
```

Vanligtvis används ett `try`-block där kommunikationen med ett program utförs, och ett `catch`-block där de undantag (av typen `IOException`) som kan uppstå under kommunikationen fångas och hanteras. Motsvarande förbindelse måste stängas, oavsett om kommunikationen avslutas normalt eller via ett undantag. Detta kan åstadkommas genom att metoden `close` anropas både i `try`-blocket och i `catch`-blocket. Ett bättre alternativ är att placera ett anrop till metoden `close` i ett `finally`-block. Detta block kan organiseras så här:

```
finally
{
    try
```


© FADIL GALJIĆ

```
    {
        s.close ();
    }
    catch (IOException e)
    {
        e.printStackTrace ();
    }
}
```

Anropet till metoden `close` placeras i ett `try`-block, eftersom även denna metod kan kasta ett (kontrollerat) undantag av typen `IOException`.

När en socket stängs, stängs automatiskt även de strömmar som är knutna till socketen. Även det omvända gäller: om en av strömmarna stängs, stängs även motsvarande socket, och därmed alla andra strömmar (som är knutna till denna socket). Det innebär att det räcker med att stänga den socket som representerar en förbindelse, eller att stänga en av strömmarna som är knutna till denna socket.

Ett klientprogram

Ett Javaprogram kan ansluta sig till en *webbserver* och begära en fil från denna server, och läsa det som servern skickar. Ett sådant program kallas för en *webbklient*. Programmet kan skapas så här:

```
import java.io.*;
import java.net.*;

class WebbKlient
{
    public static void main (String[] args)
    {
        Socket    s = null;

        try
        {
            s = new Socket ("www.kth.se", 80);

            OutputStream    os = s.getOutputStream ();
            PrintWriter    out = new PrintWriter (os, true);

            InputStream    is = s.getInputStream ();
            BufferedReader    in = new BufferedReader (
                                 new InputStreamReader (is));

            out.println ("GET /index.html HTTP/1.0");
            out.println ();
```

© FADIL GALJIĆ

```
            String    rad = in.readLine ();
            while (rad != null)
            {
                System.out.println (rad);
                rad = in.readLine ();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace ();
        }
        finally
        {
            try
            {
                s.close ();
            }
            catch (IOException e)
            {
                e.printStackTrace ();
            }
        }
    }
}
```

Här skapas först en socket (referensen `s` skapas utanför `try`-blocket, så att den blir tillgänglig även i `finally`-blocket), och därmed skapas en förbindelse till den angivna webbservern. Sedan extraheras en utström och en inström, och ett lämpligt kommunikationsverktyg bildas utifrån dessa strömmar. Därefter begärs en fil från webbservern, och sedan läser programmet det som webbservern skickar, och skriver ut det till standardutmatningsenheten (programmet tar emot en `html`-fil). Till sist stängs förbindelsen.

Kommunikation mellan två Javaprogram

Skapa en förbindelse mellan två program

Två Javaprogram kan kommunicera med varandra. Dessa två program skapas enligt olika mönster. Ett av programmen måste aktiveras först. Detta program väntar sedan på det andra programmet. Detta program är en *väntande part* i kommunikationen. Sedan aktiveras det andra programmet, och skickar sin begäran (eng. request) om anslutning. Detta program är en *anslutande part* i kommunikationen. När den väntande parten tar emot en anslutningsbegäran, skapar den en förbindelse till den

© FADIL GALJIĆ

anslutande parten. De två programmen kan sedan kommunicera via denna förbindelse.

Det väntande programmet använder ett objekt av typen `java.net.ServerSocket` för att vänta på ett annat program. När ett sådant objekt skapas, anges den port som programmet ska vänta på (och på vilken den andra parten ska ansluta sig). Detta görs på följande vis:

```
ServerSocket    ss = new ServerSocket (1201);
```

Portar upp till `1024` är reserverade för standardtjänster, och därför används en port utanför det reserverade intervallet. Om en redan upptagen port väljs, kastar motsvarande konstruktor ett undantag av typen `IOException`.

Metoden `accept` (i klassen `ServerSocket`) används för att vänta på ett annat program. När en anslutningsbegäran kommer, skapas en förbindelse till det anslutande programmet och ett objekt av typen `Socket` returneras. Detta objekt representerar den skapade förbindelsen:

```
Socket    anslutandePart = ss.accept ();
```

Från det returnerade objektet kan sedan motsvarande inström och utström extraheras, och dessa strömmar kan användas för kommunikation med den anslutande parten. Den anslutande partens utdata blir den väntande partens indata, och vice versa.

När en förbindelse till det anslutande programmet har skapats, kan det väntande objektet stängas. Detta görs med metoden `close`:

```
ss.close ();
```

Metoderna `accept` och `close` kan båda kasta ett undantag av typen `IOException`.

Den skapade förbindelsen stängs inte när det väntande objektet (av typen `ServerSocket`) stängs. Denna förbindelse representeras av ett objekt av typen `Socket` på den anslutande sidan och ett objekt av typen `Socket` på den väntande sidan. Förbindelsen stängs inte så länge inte något av dessa två objekt stängs (eller någon av de tillhörande strömmarna).

Två program som kommunicerar med varandra

Det går att skapa en förbindelse mellan två Javaprogram, och använda denna förbindelse för kommunikation. De två programmen kan köras på en och samma dator, eller på två olika datorer i nätet.

© FADIL GALJIĆ

Två program som kommunicerar med varandra måste följa ett i förväg bestämt protokoll (ett applikationsprotokoll). Det måste framgå hur en kommunikation ska påbörjas, hur den sedan ska utvecklas och hur den ska avslutas. Den väntande parten kan till exempel skicka ett meddelande för att bekräfta att förbindelsen har upprättats. Den anslutande parten kan ta emot denna bekräftelse, och börja skicka olika dataenheter. En dataenhet kan vara ett primitivt värde, en sträng eller ett objekt. Den väntande parten kan ta emot dessa dataenheter, bearbeta dem och skicka svar. Det kan finnas en loop både på den anslutande sidan och på den väntande sidan. Den anslutande sidan kan skicka en dataenhet, ta emot motsvarande svar, hantera detta svar på något sätt, skicka nästa dataenhet, och så vidare. Den väntande sidan kan ta emot en dataenhet, bearbeta den och skapa ett svar, skicka svaret, ta emot nästa dataenhet, och så vidare. Kommunikationen kan avslutas när den anslutande parten skickar ett i förväg bestämt värde (`-1`, `null` eller dylikt).

Man kan skapa ett program, som ansluter sig till ett väntande program och skickar ett (i förväg bestämt) antal strängar till det väntande programmet. Det väntande programmet kan ta emot dessa strängar och lagra dem i en vektor. Programmet kan skicka ett svar (som också är en sträng) för varje mottagen sträng. Det anslutande programmet kan ta emot dessa svar och lagra dem i en vektor. Därefter kan de båda parterna visa de strängar som lagrats under kommunikationen. Om detta protokoll följs, kan det anslutande programmet implementeras så här:

```
import java.net.*;
import java.io.*;

class AnslutandePart
{
    public static void main (String[] args)
    {
        Socket    vantandePart = null;

        try
        {
            vantandePart = new Socket ("localhost", 1201);

            OutputStream   os = vantandePart.getOutputStream ();
            PrintWriter    out = new PrintWriter (os, true);

            InputStream    is = vantandePart.getInputStream ();
            BufferedReader    in = new BufferedReader (
                                  new InputStreamReader (is));

            String[]    u = {"noll", "ett", "två", "tre", "fyra"};
            String[]    v = new String[u.length];
```

 © FADIL GALJIĆ

```
            for (int i = 0; i < u.length; i++)
            {
                out.println (u[i]);
                v[i] = in.readLine ();
            }

            for (int i = 0; i < v.length; i++)
                System.out.print (v[i] + " ");
            System.out.println ();
        }
        catch (IOException e)
        {
            e.printStackTrace ();
        }
        finally
        {
             try
             {
                 vantandePart.close ();
             }
             catch (IOException e)
             {
                 e.printStackTrace ();
             }
        }
    }
}
```

Här skapas en anslutning till ett program som körs på port `1201` på samma dator. Som datorns namn anges `localhost`, vilket innebär att allt som skickas kommer tillbaka till samma dator. Den adress som svarar mot namnet `localhost` är `127.0.0.1` (en så kallad lokal loopback-adress).

Strängar som finns i en vektor skickas, och efter varje sändning tas motsvarande svar emot. Dessa svar lagras i en annan vektor. Slutligen visas de mottagna svaren.

Det väntande programmet kan implementeras så här:

```
import java.net.*;
import java.io.*;

class VantandePart
{
    public static void main (String[] args)
    {
        ServerSocket    ss = null;
        Socket    anslutandePart = null;

        try
```

```
        {
            ss = new ServerSocket (1201);
            anslutandePart = ss.accept ();
            ss.close ();

            OutputStream   os = anslutandePart.getOutputStream ();
            PrintWriter    out = new PrintWriter (os, true);

            InputStream    is = anslutandePart.getInputStream ();
            BufferedReader    in = new BufferedReader (
                                  new InputStreamReader (is));

            String[]  u = {"zero", "one", "two", "three", "four"};
            String[]  v = new String[u.length];

            for (int i = 0; i < u.length; i++)
            {
                v[i] = in.readLine ();
                out.println (u[i]);
            }

            for (int i = 0; i < v.length; i++)
                System.out.print (v[i] + " ");
            System.out.println ();
        }
        catch (IOException e)
        {
            e.printStackTrace ();
        }
        finally
        {
            try
            {
                ss.close ();
                anslutandePart.close ();
            }
            catch (IOException e)
            {
                e.printStackTrace ();
            }
        }
    }
}
```

Programmet väntar på en angiven port, och skapar en förbindelse till det anslutande programmet. Sedan tas en sträng emot från det anslutande programmet, strängen lagras och motsvarande sträng skickas tillbaka. Detta upprepas ett antal gånger. Slutligen visas de lagrade strängarna.

 © FADIL GALJIĆ

För att testa dessa två program, ska man först starta programmet `VantandePart` och sedan programmet `AnslutandePart`. På den väntande sidan skapas följande utskrift:

```
noll ett två tre fyra
```

På den anslutande sidan blir utskriften den följande:

```
zero one two three four
```

Det ena programmet tar emot det som skickas från det andra programmet, lagrar det och visar det. De två programmen kan även köras på olika datorer. I detta fall måste den väntande datorns namn anges på den anslutande sidan (när ett objekt av typen `Socket` skapas i det anslutande programmet).

© FADIL GALJIĆ

Ett serverprogram

En flertrådad server

En *server* (ett serverprogram) är ett program som tillhandahåller en tjänst till andra program. Ett sådant program körs på en dator i nätet (en serverdator, eller kortare, server), så att andra program kan ansluta sig till detta program och utnyttja motsvarande tjänst. Dessa andra program kallas för *klienter* (eller klientprogram).

Ett serverprogram är hela tiden aktivt, och lyssnar på en viss port efter olika klienter. Så snart en klient vill ansluta sig, skapar servern en förbindelse till klienten och läser in klientens meddelande. Servern tolkar sedan meddelandet, och utför en viss tjänst. Vissa uppgifter skickas till klienten, och efter en viss tid avbryts förbindelsen. Servern fortsätter därefter att lyssna efter andra klienter.

Det kan hända att flera klienter samtidigt vill ansluta sig till en och samma server. I detta fall tar serverdatorns operativsystem dessa anslutningsbegäranden och placerar dem i en kö. Serverprogrammet hämtar den första begäran från denna kö, skapar en förbindelse till motsvarande klient och betjänar denna klient. Därefter hämtas nästa begäran från kön, en ny förbindelse skapas, och en ny klient betjänas. Klienterna betjänas i en följd, klient efter klient.

En server kan betjäna klienterna sekventiellt, klient efter klient. I så fall väntar klienterna medan en viss klient betjänas. Kommunikationen med denna klient kan ta lång tid, och väntetiderna kan därmed bli oacceptabelt långa. En klient kan vara tvungen att vänta på flera andra klienter. Detta innebär att kommunikationen med servern kan bli oacceptabelt långsam. Det krävs därför en bättre strategi för att på ett effektivt sätt kunna betjäna flera klienter.

Ett serverprogram med flera trådar (*flertrådad server*, *multitrådad server*) kan skapas. Så snart en förbindelse till en klient har skapats, skapas en ny tråd som betjänar denna klient. Huvudtråden fortsätter att lyssna efter andra klienter, medan den nyskapade tråden betjänar sin klient. Varje klient får en egen tråd på serversidan, och behöver därför inte vänta på andra klienter. Kommunikationen sker mellan en klient och motsvarande servertråd. När kommunikationen med en klient avslutas, avslutas även den motsvarande tråden.

© FADIL GALJIĆ

En tråd per klient

På en serverdator finns vanligtvis vissa uppgifter som kan vara intressanta för andra program. Om dessa uppgifter inte finns direkt tillgängliga, kan de produceras eller erhållas via kontakter med olika program på andra datorer. Olika klienter kan ansluta sig till ett serverprogram som körs på serverdatorn, och begära vissa av dessa lokala uppgifter. Serverprogrammet analyserar dessa begäranden och reagerar på ett lämpligt sätt.

En server (en exempelserver, för att illustrera principer) kan ha tillgång till namn på alla heltal från `0` till `10`. Dessa namn kan definieras i en särskild klass på serverdatorn. Denna klass kan kallas för `HeltalsNamn`, och skapas så här:

```
class HeltalsNamn
{
    public static final String[]    namn = {
        "noll", "ett", "två", "tre", "fyra",
        "fem", "sex", "sju", "åtta", "nio", "tio"
        };
}
```

Dessa namn som finns på serverdatorn kan vara av intresse för olika program, som i så fall kan skicka olika heltal till denna namnserver och begära motsvarande namn.

En namnserver kan definieras och implementeras i två klasser. Klassen `NamnServer` kan definiera en tråd som lyssnar efter olika klienter, och skapar förbindelser till dessa klienter. Så snart en förbindelse till en klient har skapats, skapas en särskild tråd som ska hantera kommunikationen med just den klienten. Denna tråd kan definieras i en klass som heter `KlientHanterare`.

Huvudtråden (den tråd som lyssnar efter klienter) kan definieras så här:

```
import java.net.*;
import java.io.*;

class NamnServer
{
    public static void main (String[] args) throws IOException
    {
        ServerSocket    ss = new ServerSocket (1201);

        while (true)
        {
            Socket    klient = ss.accept ();
```

© FADIL GALJIĆ

```
            Thread    t = new Thread (
                            new KlientHanterare (klient));
            t.start ();
        }
    }
}
```

Servern lyssnar efter klienter på en given port. Så snart en förbindelse skapats, skapas en tråd av typen `KlientHanterare`, och den socket som representerar förbindelsen skickas till denna tråd. Huvudtråden fortsätter att lyssna efter nästa klient, medan den nyskapade tråden sköter kontakten med sin klient.

Den tråd som hanterar en klient kan definieras så här:

```
import java.net.*;
import java.io.*;

class KlientHanterare implements Runnable
{
    private Socket    klient;

    public KlientHanterare (Socket klient)
    {
        this.klient = klient;
    }

    public void run ()
    {
        try
        {
            OutputStream    os = klient.getOutputStream ();
            PrintWriter    out = new PrintWriter (os, true);

            InputStream    is = klient.getInputStream ();
            BufferedReader    in = new BufferedReader (
                                    new InputStreamReader (is));

            int    heltal = Integer.parseInt (in.readLine ());
            String    namn = null;
            while (heltal != -1)
            {
                namn = HeltalsNamn.namn[heltal];
                out.println (namn);

                heltal = Integer.parseInt (in.readLine ());
            }
        }
        catch (IOException e)
        {
            e.printStackTrace ();
```


© FADIL GALJIĆ

```
        }
        finally
        {
            try
            {
                klient.close ();
            }
            catch (IOException e)
            {
                e.printStackTrace ();
            }
        }
    }
}
```

Här skapas först motsvarande kommunikationsverktyg (strömmar), så att olika textmeddelanden kan tas emot och skickas. När tråden tar emot ett heltal från sin klient (de mottagna strängarna omvandlas till motsvarande heltal), hittar den motsvarande namn och skickar det till klienten. När heltalet `-1` tas emot, avslutas kommunikationen. Sedan stängs motsvarande socket, och därmed frigörs motsvarande system- och nätverksresurser (ett `finally`-block används, så att socketen stängs oavsett om kommunikationen avslutats normalt eller via ett undantag). Därefter avslutas tråden och motsvarande resurser frigörs. En tråd av denna typ skapas för att betjäna en enda klient.

I klassen `KlientHanterare` används ett visst protokoll som måste följas av alla klienter. Endast heltal mellan `0` och `10` kan skickas, och klienten ska avsluta med att skicka `-1`. Vissa förbättringar kan införas i klassen `KlientHanterare`, så att förbindelsen avbryts och motsvarande tråd avslutas även om en klient inte följer protokollet (till exempel om klienten aldrig skickar heltaet `-1`).

Ett klientprogram, som skickar flera heltal till namnservern och tar emot och visar motsvarande namn, kan skapas på följande vis:

```
import java.net.*;
import java.io.*;

class NamnKlient
{
    public static void main (String[] args)
    {
        Socket    server = null;

        try
        {
            server = new Socket ("localhost", 1201);
```

© FADIL GALJIĆ

```
            OutputStream    os = server.getOutputStream ();
            PrintWriter    out = new PrintWriter (os, true);

            InputStream    is = server.getInputStream ();
            BufferedReader    in = new BufferedReader (
                                   new InputStreamReader (is));

            int    heltal = 0;
            String    namn = null;
            for (int i = 0; i < 5; i++)
            {
                heltal = (int) (11 * Math.random ());
                out.println (heltal);

                namn = in.readLine ();
                System.out.println (heltal + " " + namn);
                Thread.sleep (7000);
            }
            out.println (-1);
        }
        catch (Exception e)
        {
            e.printStackTrace ();
        }
        finally
        {
            try
            {
                server.close ();
            }
            catch (IOException e)
            {
                e.printStackTrace ();
            }
        }
    }
}
```

Servern och flera klienter kan köras på en och samma dator. Servern ska startas först. Därefter startar man klientprogrammet ett antal gånger. På så sätt kan man följa händelseförloppet när flera klienter samtidigt kontaktar servern. Man ska märka att servern samtidigt betjänar alla klienter. På klientsidan skapas en utskrift som har följande form:

```
5 fem
7 sju
6 sex
0 noll
9 nio
```


© FADIL GALJIĆ

En pool av trådar

En server med en pool av trådar

Ett serverprogram kan skapa en särskild tråd för hantering av en klient som ansluter sig. När kommunikationen med klienten avslutas, avslutas även motsvarande tråd. För varje klient som ansluter sig skapas en särskild tråd. Denna strategi är speciellt lämplig när den uppgift som tråden utför är tidskrävande. Den tid som krävs för att skapa och starta en tråd, och för att förstöra tråden, är i så fall väsentligt mindre än den tid som går åt för att betjäna en klient. Det kan dock hända att den tid som krävs för att betjäna en klient är jämförbar med den tid det tar att skapa, starta och förstöra en tråd. Om servern är högtrafikerad, går det i så fall åt mycket tid för att skapa och starta nya trådar, och för att förstöra dem. Serverns hastighet kan vara otillräcklig, och man måste därför hitta en mer passande strategi för dessa situationer.

Ett antal trådar kan skapas när en server startas, och dessa trådar kan sedan användas för att betjäna olika klienter. Man kan skapa en *pool av trådar*, som väntar på olika klienter. Så snart en klient ansluter sig, tilldelas denna klient till en ledig tråd. Denna tråd ger service åt denna klient, och väntar sedan på en ny klient. På så vis kan man undvika den tidsförlust som uppstår när en ny tråd skapas och startas för varje klient.

En plats för lediga trådar

En tråd i poolen kan vara ledig, eller vara upptagen med att betjäna en konkret klient. Det måste finnas en lagringsplats där de lediga trådarna kan lagras. När servern startas, placeras alla trådar i poolen på denna plats. När en klient ansluter sig, tas en ledig tråd och klienten tilldelas till denna tråd. Tråden betjänar klienten, och därefter placerar sig tråden åter på platsen för de lediga trådarna. På så sätt kan samma tråd användas för att betjäna många klienter.

Det behövs en lämplig datastruktur som lediga trådar kan lagras i. En tråd ska kunna placeras i denna datastruktur, och hämtas från den. En datastruktur som har denna funktionalitet brukar kallas för *en kanal*. Man kan definiera en kanal i ett gränssnitt som heter `Channel`, på följande vis:

```
public interface Channel<T>
{
    void put (T object) throws InterruptedException;
    T take () throws InterruptedException;
}
```

© FADIL GALJIĆ

Med metoden `put` placeras ett objekt i en kanal. Metoden `take` används för att hämta ett objekt från kanalen. Objekt tas från kanalen i samma ordning som de placeras i den (först in, först ut). Om det inte finns plats i kanalen för ett objekt, måste metoden `put` vänta. Om kanalen är tom, väntar man i metoden `take`. Både metoden `put` och metoden `take` kan kasta ett undantag av typen `java.lang.InterruptedException` (om avbrottssignalen tas emot).

Gränssnittet `Channel` kan implementeras i olika klasser. Ett objekt av en sådan klass kan sedan användas som lagringsplats för lediga trådar i en pool av trådar.

Istället för att definiera egna gränssnitt och klasser, kan de gränssnitt och klasser som redan finns i standardbiblioteket användas. Gränssnittet `java.util.concurrent.BlockingQueue` definierar en *blockeringskö* som, förutom andra metoder, även har metoderna `put` och `take`. Klassen `java.util.concurrent.ArrayBlockingQueue` representerar en cirkulär buffert, som implementerar gränssnittet `BlockingQueue`. Man kan skapa en sådan buffert (kapaciteten anges som argument), och använda den som lagringsplats för lediga trådar.

Definiera en uppgift

På en server preciserar man hur en viss klient ska betjänas. En klass skapas, och den kod som ska utföras vid kommunikationen placeras i en lämplig metod i klassen. Normalt implementerar en sådan klass gränssnittet `java.lang.Rinnable`, och kommunikationskoden placeras i `run`-metoden i klassen. Man kan kalla klassen för `KlienHanterare`, och implementera den enligt följande mönster:

```
import java.net.*;
import java.io.*;

class KlientHanterare implements Runnable
{
    private Socket    klient;

    public KlientHanterare (Socket klient)
    {
        this.klient = klient;
    }

    public void run ()
    {
```

 © FADIL GALJIĆ

```
        // kommunikationskoden här
    }
}
```

Klassen `KlientHanterare` definierar en kommunikationsuppgift. När en klient ansluter sig kan man skapa en tråd av klassen, och överlämna klienten till den tråden. Tråden ska betjäna klienten enligt det mönster som byggts in i metoden `run` i klassen. Men klassen `KlientHanterare` kan även användas på ett annat sätt. Man kan skapa ett objekt av klassen, och tillföra objektet till en redan existerande tråd. Denna tråd ska sedan anropa metoden `run` i samband med objektet, och på så sätt betjäna klienten. Fördelen med denna strategi är att en ny tråd inte behöver skapas när en klient ansluter sig. Man kan ha en pool av (redan skapade) *arbetstrådar* (eng. executors) och använda en av dessa trådar. Det enda man behöver göra är att definiera och tillföra en arbetsuppgift till en sådan tråd. Man gör det genom att skapa ett `Runnable`-objekt (till exempel ett objekt av klassen `KlientHanterare`), och tillföra objektet till arbetstråden. Arbetstråden anropar (i sin `run`-metod) objektets `run`-metod, och betjänar på så sätt klienten. När kommunikationen har avslutats kan arbetstråden få en ny uppgift (i form av ett nytt `Runnable`-objekt), och utföra denna. På så sätt kan en och samma tråd betjäna olika klienter.

Definiera en tråd i poolen

Man kan skapa en pool med trådar, och använda dessa trådar för att betjäna olika klienter. Man behöver inte skapa en ny tråd varje gång när en klient ansluter sig. Det enda som man behöver göra är att skapa ett objekt som representerar klienten och kommunikationen med denna, och tillföra detta objekt till en av arbetstrådarna i poolen. En klient och kommunikationen med den kan representeras via ett `Runnable`-objekt. När man skapar ett sådant objekt, tillför man en referens som refererar till klienten till detta objekt. Objektets `run`-metod representerar kommunikationen med klienten. På så sätt kan ett `Runnable`-objekt utnyttjas för att representera en kommunikationsuppgift.

För att definiera en arbetstråd i poolen, skapar man en särskild klass. Klassen kan heta `ArbetarTrad`, och implementeras på följande vis:

```
class ArbetarTrad extends Thread
{
    private BlockingQueue<ArbetarTrad>    ledigaTradar;
    private BlockingQueue<Runnable>       uppgiftBehallare;

    public ArbetarTrad (BlockingQueue<ArbetarTradd> ledigaTradar)
```

© FADIL GALJIĆ

```
    {
        this.ledigaTradar = ledigaTradar;
        uppgiftBehallare = new ArrayBlockingQueue <Runnable> (1);
    }

    public void run ()
    {
        while (true)
        {
            try
            {
                ledigaTradar.put (this);

                Runnable    uppgift = uppgiftBehallare.take ();

                uppgift.run ();
            }
            catch (InterruptedException e)
            {}
        }
    }

    public void utfor (Runnable uppgift)
                      throws InterruptedException
    {
        uppgiftBehallare.put (uppgift);
    }
}
```

En tråd i poolen (ett objekt av typen `ArbetarTrad`) har tillgång till två kanaler. Kanalen `ledigaTradar` är den plats där alla lediga trådar i poolen lagras. Kanalen `uppgiftBehallare` används för att tillföra en konkret uppgift till en tråd i poolen. En uppgift representeras genom ett `Runnable`-objekt (till exempel ett objekt av typen `KlientHanterare`), och tillförs via metoden `utfor`. Det måste finnas en huvudtråd, som skapar och använder arbetstrådar. Det är den tråden som anropar metoden `utfor`, och på så sätt placerar ett `Runnable`-objekt i motsvarande kanal (tilldelar en uppgift till arbetstråden).

Beteendet för en tråd i poolen definieras i metoden `run`. En sådan tråd skapas och startas när servern startas. Tråden börjar exekvera koden i `run`-metoden, och placerar sig i kanalen för lediga trådar. Där väntar den tills en annan tråd tar den från kanalen och tillför den en uppgift (ett `Runnable`-objekt) via metoden `utfor`. Tråden väntar i metoden `wait`, som anropas i metoden `take`. Tråden är blockerad tills en uppgift inkommer i kanalen `uppgiftBehallare`. När en uppgift placeras i denna behållare aktiveras tråden igen, och hämtar denna uppgift. Tråden utför sedan uppgiften genom att anropa metoden `run` i samband med det objekt som tillförs. En

 © FADIL GALJIĆ

tråd från poolen aktiverar en run-metod, som utför kommunikationen med en viss klient (denna klient anges när det motsvarande Runnable-objektet skapas, till exempel när ett objekt av typen KlientHanterare skapas). När kommunikationen avslutas (när den anropade run-metoden avslutas), placerar sig tråden igen i kanalen med de lediga trådarna. Den väntar tills en annan tråd tar den från kanalen, och tillför den till en annan uppgift. Sedan upprepas allt på samma sätt.

Skapa och använda en trådpool

En trådpool kan skapas i den klass där huvudtråden definieras. Även kanalen för lediga trådar kan skapas där. Så här kan denna klass se ut:

```
class EnServerMedEnTradpool
{
    public static void main (String[] args) throws Exception
    {
        ArbetarTrad[]    pool = new ArbetarTrad[10];
        BlockingQueue    ledigaTradar =
              new ArrayBlockingQueue<ArbetarTrad> (pool.length);
        for (int i = 0; i < pool.length; i++)
        {
            pool[i] = new ArbetarTrad (ledigaTradar);
            trad[i].start ();
        }

        ServerSocket    ss = new ServerSocket (1201);
        while (true)
        {
            Socket    klient = ss.accept ();
            Runnable    r = new KlientHanterare (klient);

            ArbetarTrad    ledigTrad = ledigaTradar.take ();
            ledigTrad.utfor (r);
        }
    }
}
```

Här skapas en pool med 10 arbetstrådar (10 objekt av typen ArbetarTrad). En kanal för lediga trådar skapas, och denna kanal tillförs till var och en av trådarna i poolen (så att de kan placera sig i denna kanal när de är lediga). Alla trådar i poolen startas. Dessa trådar lever så länge servern är aktiv.

När trådpoolen har skapats, lyssnar programmet på en given port efter klienterna. När en förbindelse skapas, skapas ett Runnable-objekt (av typen KlientHanterare) som kan hantera den aktuella klienten i sin run-

© FADIL GALJIĆ

metod. Men denna metod måste aktiveras på något sätt. Detta sker via en arbetartråd i poolen. En ledig tråd tas från kanalen för lediga trådar, och den lediga trådens `utfor`-metod anropas. På så sätt placeras det `Runnable`-objekt som skapats i arbetartrådens kanal. Denna tråd tar objektet från kanalen (tråden väntar i metoden `take`, och tar objektet via denna metod) och anropar dess `run`-metod. Denna metod utför kommunikation med klienten. Huvudtråden fortsätter med att vänta på en ny klient, efter att den har lämnat uppgiften till den första lediga tråden från poolen.

Man kan här se hur huvudtråden och trådarna i poolen samarbetar. Lediga trådar i poolen väntar på en uppgift, och huvudtråden skickar en viss uppgift till en av dessa trådar. Uppgiften representeras av ett `Runnable`-objekt, vars `run`-metod definierar kommunikationen med en klient. Tråden som får objektet aktiverar sedan objektets `run`-metod, och på så sätt utförs kommunikationen.

Använda både en trådpool och tillfälliga trådar

När man skapar en trådpool, fixerar man antalet trådar i poolen. Tillräckligt många trådar behöver skapas, så att flera klienter kan betjänas samtidigt. Man ska dock inte skapa för många trådar, eftersom varje tråd utnyttjar systemresurser. Ett problem uppstår om en klient vill ansluta sig när alla trådar i poolen är upptagna. Olika strategier kan användas för att hantera denna situation. En lösning är att skapa en ny tråd för just denna klient (en tillfällig tråd som skapas utifrån ett `Runnable`-objekt). På så sätt går det att kombinera den strategi som skapar en tråd per klient med den strategi som använder en pool av trådar. Ett antal permanenta trådar skapas, som normalt betjänar alla klienter. Om det behövs flera trådar, skapas tillfälliga trådar så att alla klienter kan betjänas.

Trådpoolen och tillfälliga trådar kan kombineras genom att loopen i huvudtråden modifieras. Detta kan göras så här:

```
while (true)
{
    Socket    klient = ss.accept ();
    Runnable    r = new KlientHanterare (klient);

    if (ledigaTradar.isEmpty ())
    {
        Thread    t = new Thread (r);
        t.start ();
    }
    else
    {
```

 © FADIL GALJIĆ

```
        ArbetarTrad    ledigTrad = ledigaTradar.take ();
        ledigTrad.utfor (r);
    }
}
```

Om det inte finns någon ledig tråd i poolen, skapas en tillfällig tråd för att betjäna motsvarande klient. I annat fall tas den första lediga tråden från kanalen, och kommunikationsuppgiften tillförs till denna tråd. Metoden `isEmpty` används för att kontrollerara om kanalen är tom. Gränssnittet `BlockingQueue` ärver denna metod från sitt supergränssnitt `java.util.Collection`.

En trådpool i standardbiblioteket

Klassen `java.util.concurrent.ThreadPoolExecutor` i Javas standardbibliotek definierar en trådpool. En trådpool av typen `ThreadPoolExecutor` kan skapas så här:

```
ThreadPoolExecutor    pool =
         (ThreadPoolExecutor) Executors.newFixedThreadPool (10);
```

Den statiska metoden `newFixedThreadPool` i klassen `java.util.concurrent.Executors` skapar en trådpool med ett angivet antal arbetstrådar (`10`). Metoden returnerar en referens av typen `java.util.concurrent.ExecutorService`, som refererar till det skapade objektet. Denna referens kan omvandlas till en referens av typen `ThreadPoolExecutor`, om så behövs.

En uppgift representerad med ett `Runnable`-objekt tillförs till poolen med metoden `submit` (som definieras i gränssnittet `ExecutorService`, så att en referens av detta gränssnitt kan aktivera den). Metoden hämtar den första lediga tråden från poolen, och tilldelar den uppgiften. Uppgiften utförs sedan (metoden `run` i `Runnable`-objektet anropas automatiskt).

Klassen `Executoes` innehåller flera metoder, som kan generera arbetstrådar enligt olika mönster. Metoden `newSingleThreadExecutor` skapar bara en tråd, som olika uppgifter tilldelas sekventiellt till. Metoden `newCachedThreadPool` skapar en trådpool som genererar arbetstrådar enligt behov. Om en tråd blir ledig i 60 sekunder, avslutas den. Metoderna returnerar en referens av gränssnittet `ExecutorService`. Förutom metoden `submit`, definieras i detta gränssnitt även metoden `shutdown`, som stänger motsvarande tjänst (de uppgifter som redan tilldelats utförs i sin helhet).

Kommunikation via en server

Man kan skapa en server, som kopplar samman ett antal klienter. En sådan server tar emot ett meddelande från en av klienterna, och skickar meddelandet vidare till alla andra klienter. Olika klienter kan på så vis kommunicera med varandra.

En klient kan kommunicera hur länge som helst med andra klienter. Man ska därför skapa en särskild tråd för varje klient som ansluter sig. En sådan klienthanterare ska ha en loop, där en klients meddelande tas emot och skickas till de övriga klienterna. För att kunna skicka meddelandet till de andra klienterna, måste en klienthanterare ha tillgång till motsvarande utströmmar. Det måste finnas en datastruktur där utströmmarna till samtliga anslutna klienter placeras. Klienthanteraren går igenom denna datastruktur och skickar ett meddelande via de lagrade strömmarna.

En vektor av typen `java.util.ArrayList` kan användas för att lagra utströmmar till alla anslutna klienter. Denna vektor kan skapas som en statisk resurs i den klass som definierar en klienthanterare. På så vis blir den tillgänglig för alla klienthanterare. När en klienthanterare skapar en lämplig utström, ska den lagra denna utström i vektorn. När förbindelsen till motsvarande klient avbryts, ska klienthanteraren ta bort utströmmen från vektorn. På så sätt kan utströmmarna till alla anslutna klienter hållas i en gemensam datastruktur. Varje klienthanterare kan komma åt dessa utströmmar, och skicka det meddelande som den fått från sin klient till de övriga klienterna.

Klassen `KlientHanterare`, som hanterar kommunikation med en klient, kan implementeras så här:

```
import java.net.*;
import java.io.*;
import java.util.*;

class KlientHanterare implements Runnable
{
    private static ArrayList<ObjectOutputStream>    outStreams =
                             new ArrayList<ObjectOutputStream> ();

    private Socket    klient;

    public KlientHanterare (Socket klient)
    {
        this.klient = klient;
    }
```


© FADIL GALJIĆ

```
    public void run ()
    {
        ObjectInputStream     in  = null;
        ObjectOutputStream    out = null;

        try
        {
            in = new ObjectInputStream (klient.getInputStream ());
            out = new ObjectOutputStream (
                                 klient.getOutputStream ());
            outStreams.add (out);

            Object    obj = null;
            int    size  = 0;
            int    index = 0;
            ObjectOutputStream    oos = null;
            while (true)
            {
                obj = in.readObject ();

                synchronized (outStreams)
                {
                    size = outStreams.size ();
                    for (index = 0; index < size; index++)
                    {
                        try
                        {
                            oos = outStreams.get (index);
                            if (oos != out)
                                oos.writeObject (obj);
                        }
                        catch (IOException e)
                        {}
                    }
                }
            }
        }
        catch (Exception e)
        {
            outStreams.remove (out);

            try
            {
                klient.close ();
            }
            catch (IOException ex)
            {}
        }
    }
}
```

© FADIL GALJIĆ

I klassen används objektströmmar för att kommunicera med klienterna. På så sätt kan meddelanden av olika slag överföras mellan dessa klienter. Det skapas en inström från klienten och en utström till klienten. Utströmmen lagras i vektorn `outStreams`, där alla utströmmar finns.

Kommunikationen definieras i en `while`-loop. I loopen tas ett meddelande från en klient emot, och skickas vidare till alla övriga klienter. Detta upprepas tills kommunikationen med klienten fungerar bra. När en undantagssituation inträffar i samband med denna kommunikation (normalt när klienten avbryter sin förbindelse), avslutas kommunikationen. Den motsvarande utströmmen tas bort från vektorn `outStreams` och motsvarande socket stängs.

Ett mottaget meddelande skickas till de övriga klienterna i ett synkroniserat block. Vektorn `outStreams` låses, så att inga nya strömmar kan placeras i vektorn, eller befintliga strömmar tas bort. Ström efter ström erhålls från vektorn, och meddelandet skickas via dessa strömmar. Meddelandet skickas till samtliga klienter, med undantag för den klient som skickat meddelandet (meddelandet skickas inte tillbaka, det vidarebefordras till de andra klienterna). Denna klients utström går att känna igen bland de övriga strömmarna i vektorn, eftersom det finns en referens till denna ström (referensen `out`), och denna referens kan jämföras med referenserna i vektorn.

Det kan inträffa att en klient stänger sin förbindelse till servern medan ett meddelande vidarebefordras. I så fall uppstår ett undantag om meddelandet skickas till denna klient efter stängningen. I detta fall ska inte den aktuella klienthanteraren avslutas, eftersom det kan finnas andra anslutna klienter som väntar på meddelandet. Därför ignoreras de undantag som uppstår när ett meddelande vidarebefordras. Klienthanteraren som hanterar den klient som avslutat kommunikationen ska själv ta bort motsvarande ström från vektorn, och avsluta exekveringen.

En klienthanterare tar emot ett meddelande från en klient, och tillför detta meddelande till alla andra klienter. En klienthanterare (och motsvarande tråd) skapas för varje klient som ansluter sig, vilket möjliggör kommunikationen mellan klienterna. Det finns en server som utgör en länk mellan olika klienter.

En klient som vill kommunicera med andra klienter måste vara flertrådad. En tråd ska skicka meddelanden till servern (och via den till andra klienter), och en annan tråd ska ta emot de meddelanden som kommer från servern (från olika klienter via servern). Eftersom man inte vet hur många meddelanden som kommer, och när de kommer, måste det finnas en särskild tråd som tar emot meddelandena i en loop.

© FADIL GALJIĆ

© FADIL GALJIĆ

Kapitel 3

Fönster

En ram

Ett grafiskt program
Skapa och hantera en ram
Definiera en egen ram
En allmän ram
Ett fönster utan dekorationer

Dialoger

En dialog
Standarddialoger
Genvägar till standarddialoger
En fildialog
En färgdialog

En ram

Ett grafiskt program

Kommunikationen med ett Javaprogram kan ske via ett konsolfönster. Till programmet kan olika uppgifter tillföras, och olika resultat och meddelanden kan erhållas från programmet. Även programmets flöde kan styras via detta fönster. Ett sådant program, som kommunicerar med sin omgivning via ett konsolfönster, kallas för ett *konsolprogram*.

Ett konsolfönster är ett enkelt gränssnitt (eng. interface) mellan användaren och programmet. De kan kommunicera via detta gränssnitt. Men det går också att skapa mer sofistikerade gränssnitt. Man kan utforma och skapa sina egna gränssnitt. Ett programs gränssnitt mot användaren kan definieras som en del av programmet. Ett eller flera fönster, och en rad andra grafiska komponenter, kan skapas. På så sätt skapas ett *grafiskt gränssnitt*. Ett sådant gränssnitt kallas även för *GUI* (eng. Graphical User Interface – *grafiskt användargränssnitt*). Ett program som har ett grafiskt gränssnitt mot användaren kallas för ett *grafiskt program*.

Skapa och hantera en ram

Skapa och visa en ram

En standardkomponent i ett grafiskt program är ett fönster, som används som behållare för andra grafiska komponenter. Kommunikationen med ett program sker via detta fönster och dess komponenter. Ett sådant fönster är en grundsten i ett grafiskt gränssnitt. Det är programmets toppnivåbehållare (top-level container). Andra komponenter kan placeras i denna behållare, men behållaren kan inte placeras i en annan behållare. Ett sådant fönster kallas för en *ram* (eng. *frame*) (se nedanstående bild).

© FADIL GALJIĆ

En ram representeras i ett Javaprogram med ett objekt av klassen `JFrame`. Denna klass finns i paketet `javax.swing`, som utgör en samling klasser som representerar olika grafiska komponenter. Ett program som skapar och visar en ram, kan skapas så här:

```
import javax.swing.*;

class SkapaVisaEnRam
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame ();
        frame.setSize (340, 240);
        frame.setVisible (true);
    }
}
```

Först skapas här en ram med namnet `frame`. Ramens förvalda dimensioner är `0 x 0` pixlar, och därför anges nya dimensioner för ramen. Metoden `setSize` används, och bredden och höjden (i antalet pixlar) anges som argument. En ram visas med metoden `setVisible`. Argumentet `true` anger att den aktuella ramen ska vara synlig.

En ram är ett fönster som har kanter, och en yta där andra grafiska komponenter kan placeras. Den har också en ikon (en meny kan öppnas via denna ikon), en titelrad och tre knappar (för att kunna minimera, maximera och stänga ramen). En ram kan manipuleras manuellt via dess kanter och hörn, och via dess meny, knappar och titelrad.

Avsluta ett program via en ram

När `main`-metoden (`main`-tråden) avslutas i ett grafiskt program, avslutas inte programmet. Programmet fortlever i en särskild tråd som hanterar det grafiska gränssnittet (GUI-tråden). Programmet måste därför avslutas via detta gränssnitt.

Ett naturligt sätt att avsluta ett program är att klicka på knappen längst upp till höger i programmets ram, eller att välja alternativet `Stäng` i ramens meny. Men dessa alternativ avslutar inte programmet. Ramen döljs, men fortsätter att leva och ta upp systemresurser. Programmet måste avslutas utifrån, på ett systemberoende sätt (Ctrl+Alt+Del i Windows, Ctrl+C i Unix).

En rams förvalda beteende vid stängningen kan ändras med metoden `setDefaultCloseOperation`. En av följande konstanter kan tillföras som argument till denna metod: `DO_NOTHING_ON_CLOSE` (gör ingenting), `HIDE_ON_CLOSE` (dölj ram, default val), `DISPOSE_ON_CLOSE` (förstör ram) och `EXIT_ON_CLOSE` (förstör ram, avsluta programmet). Dessa konstanter definieras i gränssnittet `java.awt.WindowConstants`, som implementeras i klassen `JFrame`. Metoden `setDefaultCloseOperation` kan anropas så här:

```
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
```

Efter det här anropet kan ramen användas för att avsluta programmet. Så snart ramen stängs (via motsvarande knapp, eller via dess meny), avslutas programmet. Samtidigt förstörs ramen och alla komponenter i den.

Ramens titel

Med metoden `setTitle` kan en text anges i ramens titelrad:

```
frame.setTitle ("Java frame");
```

Texten `Java frame` visas i ramens titelrad när ramen visas.

Texten som ska visas i ramens titelrad kan även anges när ramen skapas:

```
JFrame    frame = new JFrame ("Java frame");
```

Ramens ikon

En ram i Java har en förvald (default) ikon (Javas kaffekopp). I Windows visas denna ikon längst upp till vänster i ramen (menyn kan öppnas via ikonen).

 © FADIL GALJIĆ

Istället för den förvalda ikonen kan man använda en egen ikon. En ikon definieras utifrån en fil som innehåller en bild. Ett objekt av typen `java.awt.Image` (`Image` är en abstrakt superklass till alla klasser som representerar bilder) skapas utifrån denna fil, och objektet används som argument till metoden `setIconImage`. Metoden sätter den aktuella ramens ikon till den angivna bilden.

När ett objekt av typen `Image` skapas utifrån en fil, kontaktas det underliggande systemet i den dator som programmet exekveras på. Detta sker via ett objekt av typen `java.awt.Toolkit`. Ett objekt av denna typ erhålls med (den statiska) metoden `getDefaultToolkit` i klassen `Toolkit`. Objektets `getImage`-metod används sedan för att få ett objekt av typen `Image`, utifrån filen som innehåller motsvarande bild.

Om filen `enBild.gif` innehåller en bild, kan ramens ikon anges så här:

```
Toolkit    tk = Toolkit.getDefaultToolkit ();
Image    bild = tk.getImage ("enBild.gif");
frame.setIconImage (bild);
```

Ramens dimensioner och placering

En rams dimensioner anges med metoden `setSize`. Bredden och höjden (i antalet pixlar) anges som argument till metoden.

När en ram visas, hamnar dess övre vänstra hörn i skärmens övre vänstra hörn. Det går att ändra detta förvalda beteende och ange en annan position. En rams placering på skärmen bestäms med metoden `setLocation`. Det övre vänstra hörnets `x`-koordinat och `y`-koordinat anges som argument till metoden. `x`-koordinater på skärmen räknas från vänster till höger (i antalet pixlar), och `y`-koordinater från den övre kanten och nedåt. Skärmens övre vänstra hörn ligger i punkten `(0, 0)`.

Så här kan en rams placering på skärmen bestämmas:

```
frame.setLocation (200, 100);
```

Den aktuella ramen placeras så att dess övre vänstra hörn hamnar i punkten `(200, 100)` på skärmen.

Olika skärmar har olika upplösning. En skärm kan till exempel ha upplösningen 800 x 600 (800 pixlar på bredden, 600 pixlar på höjden). En annan skärm kan ha en större eller mindre upplösning. Därför ser en ram olika ut på olika skärmar. På en skärm kan en ram ta upp hela ytan. På en annan skärm kanske den bara tar upp hälften av skärmens yta. För att undvika detta kan ramens storlek och placering anges relativt skärmen.

© FADIL GALJIĆ

För att kunna göra detta, måste man ha tillgång till den aktuella skärmens dimensioner. Detta får man med metoden `getScreenSize` i klassen `java.awt.Toolkit`. Metoden returnerar ett objekt av typen `java.awt.Dimension`. Detta objekt har skärmens bredd och höjd (i antalet pixlar) som publika medlemmar.

Så här erhålls den aktuella skärmens dimensioner:

```
Toolkit     tk = Toolkit.getDefaultToolkit ();
Dimension    d = tk.getScreenSize ();
int    skarmBredd = d.width;
int    skarmHojd  = d.height;
```

Variabeln `skarmBredd` representerar skärmens bredd och variabeln `skarm-Hojd` dess höjd. Dessa värden kan användas för att bestämma en rams dimensioner och placering på skärmen:

```
int    frameBredd = 6 * skarmBredd / 10;
int    frameHojd  = 4 * skarmHojd  / 10;
frame.setSize (frameBredd, frameHojd);

int    xHorn = (skarmBredd - frameBredd) / 2;
int    yHorn = (skarmHojd  - frameHojd ) / 2;
frame.setLocation (xHorn, yHorn);
```

Ramens bredd sätts till 60 % av skärmens bredd och dess höjd till 40 % av skärmens höjd. Koordinaterna för ramens övre vänstra hörn beräknas så att den hamnar i mitten av skärmen. På detta sätt kommer ramen alltid att ha samma dimensioner och placering relativt skärmen.

En ram som behållare

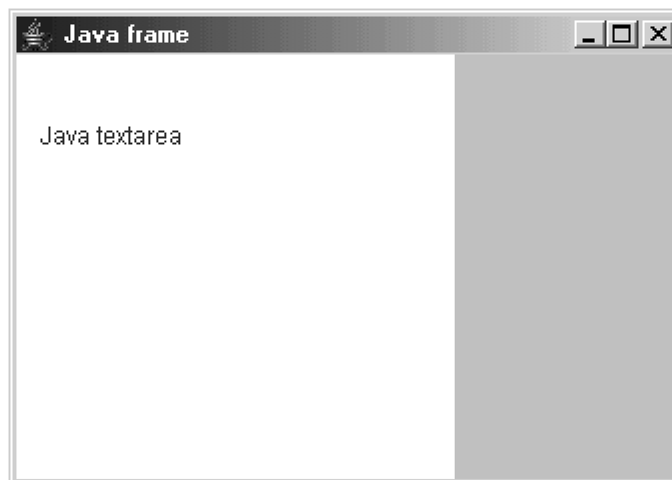
En ram kan användas som behållare för andra grafiska komponenter. På följande vis kan man skapa en ram och flera andra komponenter, och placera dessa komponenter i ramen:

```
import java.awt.*;
import javax.swing.*;

class EnRamSomBehallare
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame (" Java frame");
        frame.setSize (340, 240);
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        JPanel    panel = new JPanel ();
```

 © FADIL GALJIĆ

```
        panel.setBackground (Color.LIGHT_GRAY);

        JTextArea    textArea = new JTextArea (30, 20);
        textArea.append ("\n\n    Java textarea");

        frame.add (panel, "Center");
        frame.add (textArea, "West");

        frame.setVisible (true);
    }
}
```

Här skapas först en ram. Därefter skapas en panel (ett objekt av typen `javax.swing.JPanel`) och dess bakgrundsfärg anges. En färg representeras med ett objekt av klassen `java.awt.Color`. I denna klass definieras flera vanliga färger via motsvarande konstanter. Dessa konstanter är: `Color.YELLOW` (gul), `Color.RED` (röd),. `Color.WHITE` (vit) och så vidare. Förutom panelen skapas också en textarea (ett objekt av typen `javax.swing.JTextArea`), och en text placeras i denna textarea. Därefter placeras panelen och textarean i den skapade ramen. Panelen placeras i mitten, och textrutan till vänster i ramen (se bilden nedan).



Metoden `add` används för att placera en komponent i en ram. Om flera komponenter placeras i ramen, måste komponenternas positioner i ramen anges. Ett sätt att göra detta är att ange en lämplig teckensträng som andra argument till metoden `add`. En av följande teckensträngar kan användas: `Center`, `North`, `South`, `East` och `West`.

Superklasser till klassen JFrame

Metoderna `setDefaultCloseOperation` och `setIconImage` definieras i klassen `JFrame`. Metoderna `setSize`, `setLocation`, `setVisible` och många andra metoder som kan användas i samband med en ram, definieras inte i denna klass. De erhålls via arv från olika superklasser till klassen `JFrame`.

Klassen `JFrame` och dess superklasser kan representeras så här:

```
java.lang.Object
  |
  +--java.awt.Component
       |
       +--java.awt.Container
            |
            +--java.awt.Window
                 |
                 +--java.awt.Frame
                      |
                      +--javax.swing.JFrame
```

Klassen `Component` är en abstrakt klass som finns i roten av den klasshierarki som hanterar olika grafiska komponenter. Metoderna `setSize`, `setLocation` och `setVisible` tillhör denna klass. Klassen finns i ett paket som heter `java.awt`. Detta paket omfattar många grafikrelaterade klasser. Klassen `java.awt.Container` är en subklass till klassen `Component`, och denna klass är rotklass till alla behållare (eng. containers). En behållare är en grafisk komponent som kan innehålla andra grafiska komponenter. Metoden `add` finns i klassen `Container`. Olika fönster utgör en speciell grupp behållare. Man kan inte ha ett fönster i en annan behållare. Ett fönster definieras först i klassen `java.awt.Window`. Denna klass definierar två metoder som gör det möjligt att placera ett fönster framför eller bakom ett annat fönster. Dessa metoder heter `toFront` och `toBack`, och anropas efter det att metoden `setVisible` anropats med argumentet `true`. Ett praktiskt användbart fönster definieras i klassen `java.awt.Frame`. Metoden `setTitle` definieras i denna klass. Ursprungligen användes ett fönster av typen `Frame` som huvudfönster i ett grafiskt program. Senare infördes nya, bättre komponenter. Dessa komponenter definieras i paketet `javax.swing`, och en av dessa Swing-komponenter är en ram av typen `JFrame`. Ett fönster av typen `JFrame` används som huvudfönster i ett grafiskt program.

© FADIL GALJIĆ

Definiera en egen ram

I en konkret applikation kan det behövas en ram med ett speciellt utseende och/eller beteende. Det kan till exempel behövas en ram som avslutar programmet vid stängningen, och som har en given storlek och placering på skärmen. I detta fall kan man skapa ett objekt av typen `JFrame`, och anpassa detta objekt genom att använda olika metoder i samband med det. En annan strategi är att skapa en särskild klass som definierar en ram som motsvarar konkreta behov.

Man definierar en egen typ av ramar genom att skapa en subklass till klassen `JFrame`. I denna klass utformas en rams utseende och beteende:

```
import javax.swing.*;

class SpecialFrame extends JFrame
{
    public SpecialFrame (String titel)
    {
        super (titel);

        this.setSize (600, 400);
        this.setLocation (100, 100);
        this.setDefaultCloseOperation (EXIT_ON_CLOSE);
    }
}
```

Varje ram av typen `SpecialFrame` har en fördefinierad storlek och placering på skärmen. En sådan ram avslutar programmet vid stängningen. En ram av typen `SpecialFrame` har ett speciellt utseende och beteende, som kan behövas i en konkret applikation. En sådan ram kan användas på följande vis:

```
class EnEgenRam
{
    public static void main (String[] args)
    {
        SpecialFrame  frame = new SpecialFrame (" En egen ram");
        frame.setVisible (true);
    }
}
```

Här skapas och visas en ram av typen `SpecialFrame`. Eftersom klassen `SpecialFrame` härleds från klassen `JFrame`, kan alla metoder i klassen `JFrame` (och dess superklasser) även användas i samband med ett fönster av typen `SpecialFrame`. Man kan till exempel använda metoden `setVisible`.

© FADIL GALJIĆ

En allmän ram

Det går att definiera en ram som bara ska användas i en konkret applikation. Man kan även definiera en ram som kan användas i flera olika applikationer. Det kan till exempel vara en ram vars dimensioner och placering anges relativt skärmen. En sådan ram kan vara användbar i många olika sammanhang, eftersom det kan finnas behov av en ram som passar bra på vilken skärm som helst.

På följande vis kan man skapa klassen `RFrame` (Relative Frame), som definierar en allmänt användbar ram:

```
package fjava.edu;

import java.awt.*;
import javax.swing.*;

public class RFrame extends JFrame
{
    public RFrame ()
    {
        super ();

        initiateFrame ();
    }

    public RFrame (String title)
    {
        super (title);

        initiateFrame ();
    }

    public void setSize (double relativeWidth,
                         double relativeHeight)
    {
        Toolkit    tk = Toolkit.getDefaultToolkit ();
        Dimension    d = tk.getScreenSize ();
        int    screenWidth   = d.width;
        int    screenHeight  = d.height;

        int  frameWidth   = (int) (relativeWidth * screenWidth);
        int  frameHeight  = (int) (relativeHeight * screenHeight);

        super.setSize (frameWidth, frameHeight);
    }

    public void setLocation (double relativeX, double relativeY)
    {
```

 © FADIL GALJIĆ

```
        Toolkit    tk = Toolkit.getDefaultToolkit ();
        Dimension    d = tk.getScreenSize ();
        int    screenWidth = d.width;

        int    x  = (int) (relativeX * screenWidth);
        int    y = (int) (relativeY * screenHeight);

        super.setLocation (x, y);
    }

    private void initiateFrame ()
    {
        this.setSize (0.8, 0.7);
        this.setLocation (0.1, 0.15);

        this.setDefaultCloseOperation (EXIT_ON_CLOSE);
    }
}
```

Klassen `RFrame` ärver metoderna `setSize` och `setLocation` via klassen `JFrame`. Dessa klasser har heltalsparametrar, som preciserar en rams dimensioner och placering i antalet pixlar. Klassen `RFrame` definierar en ny metod `setSize`, som har flyttalsparametrar. Dessa parametrar preciserar en rams dimensioner som en andel av skärmens bredd och höjd. Metoden `setSize` överlagras på så sätt (både den ärvda metoden och den nya metoden kan användas i samband med en ram av typen `RFrame`). Även metoden `setLocation` överlagras. Den nya metoden har två flyttalsparametrar, som preciserar koordinaterna för en rams övre vänstra hörn som en andel av skärmens bredd och höjd.

En ram av typen `RFrame` kan skapas så här:

```
RFrame    frame = new RFrame (" En ram av typen RFrame");
```

Den här ramens bredd är 80 % av skärmens bredd, och dess höjd är 70 % av skärmens höjd. Denna ram placeras i mitten av skärmen, och programmet avslutas när den stängs. Detta är en ram som kan passa i många olika applikationer. För att ändra ramens storlek och placering, kan man göra på följande vis:

```
frame.setSize (0.6, 0.4);
frame.setLocation (0.05, 0.15);
```

Ramen får nu en bredd som är 60 % av skärmens bredd, och en höjd som är 40 % av skärmens höjd. Ramens övre vänstra hörn ligger i den punkt vars `x`-koordinat är 5 % av skärmens bredd, och vars `y`-koordinat är 15 % av skärmens höjd.

© FADIL GALJIĆ

Även ramens beteende vid stängningen kan ändras. Det görs med (den ärvda) metoden `setDefaultCloseOperation`:

```
frame.setDefaultCloseOperation (RFrame.DISPOSE_ON_CLOSE);
```

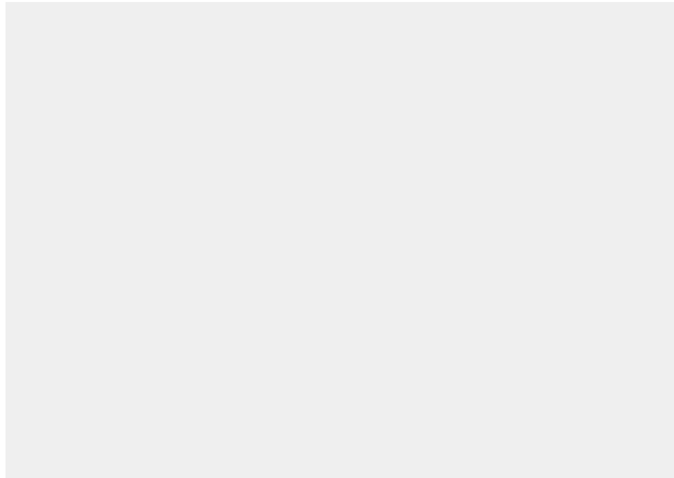
En ram av typen `RFrame` kan visas med (den ärvda) metoden `setVisible`:

```
frame.setVisible (true);
```

Alla metoder som kan användas i samband med en ram av typen `JFrame`, kan också användas i samband med en ram av typen `RFrame`. Förutom dessa metoder kan även de nya metoderna `setSize` och `setLocation` (som bestämmer en rams dimensioner och placering relativt skärmen) användas. En konstruktor i klassen `RFrame` skapar en ram som är användbar i många applikationer redan från början. Klassen `RFrame` kan placeras i ett särskilt paket (till exempel `fjava.edu`), och sedan användas i olika applikationer istället för klassen `JFrame`, och tillsammans med klassen `JFrame`.

Ett fönster utan dekorationer

I vissa sammanhang kan det behövas ett fönster utan ram, titelrad, ikon och knappar (se bilden nedan). Olika komponenter kan lagras i ett sådant fönster, och fönstret kan sedan visas tillsammans med dess komponenter. Ett sådant fönster kan stängas inifrån (till exempel genom en klickning på en knapp i fönstret) eller utifrån (till exempel genom att ett annat fönster stängs).



© FADIL GALJIĆ

Ett fönster utan ram, titelrad, ikon och knappar kan representeras med ett objekt av typen `javax.swing.JFrame`. Ett fönster av denna typ skapas, och sedan används metoden `setUndecorated` (med argumentet `true`) i samband med fönstret:

```
JFrame    fonster = new JFrame ();
fonster.setUndecorated (true);
```

Ett annat sätt att skapa ett fönster utan ram, titelrad, ikon och knappar är att använda ett objekt av typen `javax.swing.JWindow`. Ett fönster av denna typ saknar olika dekorationer redan från början. Ett sådant fönster kan skapas så här:

```
JPanel    panel = new JPanel ();
panel.setBackground (Color.BLACK);

JWindow    fonster = new JWindow ();
fonster.setSize (160, 120);
fonster.setLocation (300, 80);
fonster.add (panel, "South");
fonster.setVisible (true);
```

Här skapas en (svart) panel och ett fönster av typen `JWindow`. Fönstrets storlek och position på skärmen bestäms. Sedan placeras panelen i fönstret, och fönstret visas.

Ett fönster av typen `JWindow` används på ett liknande sätt som ett fönster av typen `JFrame`. Klasserna `JFrame` och `JWindow` har ett antal gemensamma förfäder (superklasser), och har därför ett stort antal gemensamma metoder.

Klasshierarkin för klassen `JWindow` kan representeras så här:

```
java.lang.Object
   |
   +--java.awt.Component
         |
         +--java.awt.Container
               |
               +--java.awt.Window
                     |
                     +--javax.swing.JWindow
```

Ett fönster av typen `JWindow` kan bindas till ett fönster av typen `JFrame`. I detta fall hamnar fönstret av typen `JWindow` i fokus om fönstret av typen `JFrame` visas först. Annars kan inte fönstret av typen `JWindow` placeras i fokus. Man binder ett fönster av typen `JWindow` till ett fönster av typen `JFrame` genom att ange fönstret av typen `JFrame` som argument när fönstret av typen `JWindow` skapas.

© FADIL GALJIĆ

Dialoger

En dialog

Skapa och visa en dialog

Ett program kan behöva ett kort informationsutbyte med användaren, vid ett eller flera tillfällen under programmets gång. Programmet kan behöva lämna ett viktigt meddelande till användaren, eller få någon viktig information från användaren. Användaren kan bekräfta eller välja något, eller mata in en eller flera uppgifter. Den information som användaren lämnar till programmet kan vara nödvändig för programmets vidare exekvering.

I ett grafiskt program implementeras en dialog med användaren med en speciell typ av fönster. Dessa fönster kallas för *dialogfönster*, eller kortare, *dialoger*. Ett sådant fönster representeras i ett Javaprogram med ett objekt av typen `javax.swing.JDialog`. En dialog kan skapas och visas så här:

```
import javax.swing.*;

class SkapaVisaEnDialog
{
    public static void main (String[] args)
    {
        JDialog    dialog = new JDialog ();
        dialog.setTitle (" Java dialog");
        dialog.setModal (true);

        dialog.setSize (200, 150);
        dialog.setLocation (120, 100);
        dialog.setDefaultCloseOperation (
                                JDialog.DISPOSE_ON_CLOSE);

        dialog.setVisible (true);

        System.exit (0);
    }
}
```

En dialog skapas, och dess titel bestäms. Även dialogens storlek och position på skärmen, och dess beteende vid stängningen preciseras. Dialogen visas, och till sist avslutas programmet (därmed avslutas GUI-tråden). När programmet exekveras, visas ett dialogfönster (se bilden nedan) på skärmen. Programmet avslutas när fönstret stängs.

© FADIL GALJIĆ

En dialog kan vara *modal* eller *icke-modal*. Detta anges med metoden `setModal`, som tar ett booleskt värde som argument. Genom att ange `true` som argument till metoden `setModal`, gör man dialogen till en modal dialog.

När en modal dialog visas, stannar programmet och väntar på användaren. Användaren måste först tolka det meddelande som visas i dialogen och bekräfta att han har förstått det, välja något genom att trycka på en av flera knappar, mata in något genom motsvarande textfält, eller stänga dialogen via dess stängningsknapp. Först när detta är gjort, kan programmet fortsätta exekveras. Så länge programmet väntar kan användaren inte använda andra grafiska komponenter i det grafiska gränssnittet. På så sätt tvingas användaren till en dialog. Programmet kan inte fortsätta förrän användaren gör det möjligt. Om en dialog ska visas på skärmen medan programmet går, ska en icke-modal dialog användas (`false` anges som argument till metoden `setModal`).

En dialog skiljer sig inte så mycket från en ram. En dialog har ett något speciellt utseende, och kan vara modal (en ram kan inte vara modal). Dessa två typer av fönster (dialoger och ramar) har ett stort antal gemensamma metoder, eftersom de har gemensam härstamning (de har gemensamma indirekta superklasser). Metoderna `setTitle`, `setSize`, `setLocation`, `setDefaultCloseOperation` (alternativet `EXIT_ON_CLOSE` kan inte användas i samband med en dialog), `setVisible` och en del andra metoder kan användas i samband med både ramar och dialoger. En dialog är en behållare: andra grafiska komponenter kan placeras i en dialog. Metoden `add` används för detta, på samma sätt som i samband med en ram.

Klassen `JDialog` och dess superklasser kan representeras så här:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Dialog
                          |
                          +--javax.swing.JDialog
```

Det är tydligt att en dialog, liksom en ram, i grunden är ett fönster (ett objekt av typen `java.awt.Window`). Därför har en dialog och en ram så många gemensamma egenskaper.

Definiera en egen dialog

Det är möjligt att definiera en egen typ av dialoger. Detta görs genom att en subklass till klassen `JDialog` skapas. I denna subklass definieras utseende och beteende för en dialog av denna typ. Klassens konstruktorer och metoder kan ha ett antal parametrar, som gör det möjligt att påverka en konkret dialogs utseende och beteende.

En dialogtyp kan definieras så här:

```
import javax.swing.*;

class OptionDialog extends JDialog
{
    public OptionDialog ()
    {
        this.setTitle (" Valdialog");
        this.setModal (true);
        this.setDefaultCloseOperation (DISPOSE_ON_CLOSE);

        this.setSize (300, 160);
        this.setLocation (120, 80);

        JLabel    meddelande = new JLabel ("    Välj:");
        JButton   val1 = new JButton ("Tillbaka");
        JButton   val2 = new JButton ("Nästa");
        JButton   val3 = new JButton ("Avbryt");

        JPanel    val = new JPanel ();
        val.add (val1);
        val.add (val2);
```


© FADIL GALJIĆ

```
        val.add (val3);

        this.add (meddelande, "Center");
        this.add (val, "South");
    }
}
```

En dialog av klassen `OptionDialog` kan skapas, och sedan visas med metoden `setVisible`. En dialog av denna typ har en titel, ett meddelande och tre knappar (se bilden nedan). Ett textmeddelande representeras med ett objekt av typen `javax.swing.JLabel`, och en knapp med ett objekt av typen `javax.swing.JButton`. En knapp representerar ett bestämt val. Knapparna grupperas i en panel, och denna panel placeras längst ner i dialogen. Klassen `OptionDialog` måste naturligtvis utvidgas, så att en viss logik byggs in i samband med dialogens knappar. Man måste precisera vilken kod som ska utföras när användaren klickar på en konkret knapp.

Standarddialoger

En standarddialogruta

Ett objekt av typen `javax.swing.JOptionPane` representerar en dialogruta med standardutseende och standardbeteende. Klassen `JOptionPane` har ett antal konstruktorer och `set`-metoder som gör det möjligt att utforma en dialogruta efter de aktuella behoven. En dialogruta kan innehålla ett meddelande och en ikon. En dialogruta kan också ha ett antal knappar, som representerar olika alternativ. Genom att klicka på en knapp, väljer man ett av dessa alternativ. En dialogruta kan även innehålla ett textfält, som gör det möjligt att mata in en teckensträng.

En dialogruta kan skapas så här:

```
JOptionPane    ruta = new JOptionPane ("Förbindelse upprättad!");
```

Ett meddelande tillförs som argument till konstruktorn, och det är detta meddelande som visas i dialogrutan. Normalt innehåller en dialogruta ett meddelande.

När en dialogruta skapats, ska den visas vid ett lämpligt tillfälle. Men en dialogruta kan inte visas direkt. En dialogruta är inte ett fönster (ett objekt av typen `java.awt.Window`). För att en dialogruta ska kunna visas, måste ett särskilt fönster skapas och dialogrutan placeras i detta fönster. Ett fönster av typen `javax.swing.JDialog` passar bra för detta ändamål. Klassen `JOptionPane` har en särskild metod, som skapar ett fönster av typen `JDialog` och placerar den aktuella dialogrutan i detta fönster. Denna metod heter `createDialog`. Normalt används denna metod, som skapar en dialog med en dialogruta som sin enda komponent. Dialogrutan och dialogfönstret integreras i en enhet.

Så här kan en dialog skapas utifrån en given dialogruta:

```
JDialog    dialog = ruta.createDialog (frame, "En dialog");
```

Det andra argumentet till metoden `createDialog` är en teckensträng, som representerar dialogens titel. Det första argumentet till metoden är en komponent (ett objekt av en subklass till klassen `java.awt.Component`), som representerar dialogens förälderkomponent. När dialogen visas, hamnar den ovanpå denna komponent, i mitten av den, i komponentens ram (om förälderkomponenten saknar ram, hamnar dialogen på en förvald plats). Man kan till exempel välja en ram (ett objekt av typen `javax.swing.JFrame`) som förälderkomponent till en dialog.

När en dialog har skapats, kan den justeras och visas, till exempel på detta vis:

```
dialog.setModal (true);
frame.setVisible (true);
dialog.setVisible (true);
```

Dialogen görs till en modal dialog (den kan även vara en icke-modal dialog), och visas sedan. Även dialogens förälderram visas. Dialogen placeras ovanpå och i mitten av denna ram. Om `null` anges som första argument till metoden `createDialog` (en dialog utan förälderkomponent), hamnar dialogen på en förvald plats i mitten av skärmen. Med metoderna `setSize` och `setLocation` kan man själv bestämma dialogens storlek och position.

Klassen `JOptionPane` och dess superklasser kan representeras så här:

© FADIL GALJIĆ

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JOptionPane
```

En dialogruta (ett objekt av typen JOptionPane) är, som synes, en grafisk komponent, men inte ett fönster (den är inte av typen java.awt.Window). Därför kan en dialogruta inte visas som en självständig enhet. Det behövs ett fönster som behållare för dialogrutan.

Visa ett meddelande

En dialogruta kan innehålla ett meddelande, en ikon som passar ihop med meddelandet och en OK-knapp. Dialogrutan stängs genom att användaren klickar antingen på dess stängningsknapp eller på OK-knappen (se bilden nedan).

En dialogruta med ett meddelande, en ikon och en OK-knapp kan skapas så här:

```
JOptionPane    ruta = new JoptionPane ("Förbindelse upprättad!");
ruta.setMessageType (JOptionPane.INFORMATION_MESSAGE);
```

En av flera fördefinierade typer av meddelanden kan väljas. Man väljer en typ av meddelande genom att välja en statisk konstant i klassen JOptionPane. En av följande typer kan väljas: PLAIN_MESSAGE (förvald typ), INFORMATION_MESSAGE, QUESTION_MESSAGE, WARNING_MESSAGE eller ERROR_MESSAGE. Valet av typ för den aktuella dialogrutan görs med metoden

`setMessageType` (denna typ kan även anges som andra argument till motsvarande konstruktor).

Förutom det aktuella meddelandet (som anges antingen som argument till motsvarande konstruktor eller som argument till metoden `setMessage`), innehåller en dialogruta även en ikon. Denna ikon beror på meddelandets typ. I samband med ett meddelande av typen `INFORMATION_MESSAGE`, till exempel, visas en ikon som har formen av bokstaven `i`. När ett meddelande av typen `PLAIN_MESSAGE` visas, skapas ingen ikon. Med metoden `setIcon` kan man även ange en egen ikon. Om en bild finns i en fil, kan en ikon skapas utifrån filen och tilldelas till dialogrutan så här:

```
java.awt.ImageIcon    ikon = new java.awt.ImageIcon (fil);
ruta.setIcon (ikon);
```

En standarduppsättning alternativ

En dialogruta med en standarduppsättning alternativ kan skapas. Varje alternativ representeras med en knapp. En dialogruta kan till exempel innehålla en fråga (ett meddelande), en `Ja`-knapp och en `Nej`-knapp (se bilden nedan). Frågan besvaras genom att någon av knapparna klickas.

En dialogruta med en standarduppsättning alternativ kan skapas så här:

```
JOptionPane    ruta = new JOptionPane ("Vill du fortsätta?");
ruta.setMessageType (JOptionPane.QUESTION_MESSAGE);
ruta.setOptionType (JOptionPane.YES_NO_OPTION);
```

Man väljer en uppsättning alternativ genom att välja en statisk konstant i klassen `JOptionPane`. En av följande typer kan väljas: `DEFAULT_OPTION` (förvalt alternativ, en `OK`-knapp), `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` eller `OK_CANCEL_OPTION`. Med metoden `setOptionType` väljs en standard-

 © FADIL GALJIĆ

uppsättning alternativ för den aktuella dialogrutan (denna typ kan även anges som tredje argument till motsvarande konstruktor).

Den dialog som visar dialogrutan stängs antingen med dess stängningsknapp eller med en klickning på någon av knapparna i dialogrutan. Efterföljande steg i programmet beror på hur dialogen har stängts. Därför måste man ta reda på detta. Denna information erhålls med metoden `getValue` (i klassen `JOptionPane`), som returnerar ett objekt av typen `java.lang.Object`. Om det returnerade värdet är `null`, har användaren stängt dialogen via dess stängningsknapp. Om användaren klickat på någon av knapparna inuti dialogrutan, returnerar metoden `getValue` ordningsnumret för den tryckta knappen. Detta värde returneras som ett objekt av typen `java.lang.Integer`. Knapparna räknas från vänster till höger, och räkningen börjar med `0` (knappen längst till vänster). Den valda knappens ordningsnummer erhålls på följande vis:

```
Object    val = ruta.getValue ();
int    knappNummer = -1;
if (val != null)
    knappNummer = (Integer) val;
```

Med en icke-modal dialog kan det hända att metoden `getValue` anropas innan dialogen stängts. I detta fall returnerar `getValue` ett värde som representeras med konstanten `UNINITIALIZED_VALUE` från klassen `JOptionPane` (följande test kan utföras: `if(val = = JOptionPane.UNINITIALIZED_VALUE))`.

En egen uppsättning alternativ

Man kan skapa en dialogruta med en egen uppsättning alternativ (se bilden nedan). Alternativen anges via en vektor av objekt, och binds till dialogrutan med metoden `setOptions` (i klassen `JOptionPane`).

Alternativen kan definieras, till exempel, via ett antal teckensträngar:

```
JOptionPane    ruta = new JOptionPane ("Välj nästa steg:");
Object[]    alternativ = { new String ("Avbryt"),
                           new String ("Tillbaka"),
                           new String ("Nästa") };
ruta.setOptions (alternativ);
ruta.setInitialValue (alternativ[2]);
```

Här skapas en dialogruta med tre knappar, som representerar tre angivna alternativ (`Avbryt`, `Tillbaka` och `Nästa`). Metoden `setInitialValue` används för att markera (ange som förvalt) ett av dessa alternativ.

Alternativen kan även anges via en vektor av ikoner, eller via en vektor av grafiska komponenter. Dessa ikoner, eller grafiska komponenter, visas i så fall inuti dialogrutan.

Med metoden `getValue` tar man reda på hur användaren stängde motsvarande dialog. Om teckensträngar används som alternativ, returnerar metoden en teckensträng (det valda alternativet) om användaren tryckt på någon av knapparna i dialogrutan. Denna teckensträng erhålls så här:

```
Object    val = ruta.getValue ();
String    alt = null;
if (val != null)
    alt = (String) val;
```

Mata in ett värde

Förutom ett meddelande, en ikon och en uppsättning alternativ, kan en dialogruta även innehålla ett textfält (se bilden nedan). Detta textfält kan användas för att mata in en uppgift (en teckensträng).

Så här kan en dialogruta med ett textfält skapas:

© FADIL GALJIĆ

```
JOptionPane    ruta = new JOptionPane ("Ditt namn:");
ruta.setWantsInput (true);
```

Med metoden `setWantsInput` anges att dialogrutan även ska innehålla ett textfält.

Med metoden `getValue` tar man reda på hur användaren har stängt motsvarande dialog. Om metoden inte returnerar `null` (och inte returnerar `JOptionPane.UNINITIALIZED_VALUE` i händelse av en icke-modal dialog), har användaren tryckt på `OK`-knappen (eller någon annan knapp, om det finns flera knappar). I så fall kan det värde som användaren matat in (som även kan vara en tom teckensträng) avläsas med metoden `getInputValue` (i klassen `JOptionPane`):

```
String    input = (String) ruta.getInputValue ();
```

Värdet kan sedan användas på olika sätt.

Genvägar till standarddialoger

Statiska metoder som skapar och visar dialoger

Klassen `JOptionPane` har ett antal statiska metoder som gör det enkelt att skapa och visa en standarddialog. Dessa metoder är `showMessageDialog`, `showConfirmDialog`, `showOptionDialog` och `showInputDialog`. Var och en av dessa metoder skapar en standarddialogruta och motsvarande dialog, och visar denna dialog. En sådan dialog kan stängas antingen via dess stängningsknapp eller via någon av knapparna i dialogrutan. Metoden `showMessageDialog` har inte något returvärde. Andra metoder returnerar det alternativ som användaren valt eller det värde som användaren matat in.

Dialoger som skapas med statiska metoder från klassen `JOptionPane` är modala. En icke-modal dialog kan inte skapas på detta sätt. När en dialog stängs försvinner den, och kan inte återanvändas. Vid varje anrop skapas en ny dialogruta och en ny dialog. Om man vill använda en och samma dialog vid flera tillfällen i programmet, ska man själv skapa en särskild dialog (istället för att anropa en av de statiska metoderna), och visa den varje gång det behövs.

© FADIL GALJIĆ

Metoden showMessageDialog

Metoden `showMessageDialog` visar en dialog med ett meddelande och en `OK`-knapp. Metoden kan anropas så här:

```
JOptionPane.showMessageDialog (null,
                "Programmet avslutas!",
                "MessageDialog",
                JOptionPane.INFORMATION_MESSAGE);
```

Det första argumentet representerar dialogens förälderkomponent. Dialogen visas ovanpå denna komponent, eller på en förvald plats (i mitten av skärmen) om `null` anges som första argument. Det andra argumentet representerar det meddelande som visas inuti dialogrutan. Det tredje argumentet är dialogens titel. Det fjärde argumentet representerar meddelandets typ, och denna typ bestämmer den ikon som visas i dialogrutan. Metoden `showMessageDialog` överlagras, så att även en annan uppsättning argument kan anges. Man kan till exempel ange en egen ikon som femte argument.

Metoden showConfirmDialog

Metoden `showConfirmDialog` visar en dialog med ett meddelande och en ikon, och en standarduppsättning knappar (se bilden nedan). Varje knapp representerar ett alternativ.

Metoden kan anropas så här:

```
int    val = JOptionPane.showConfirmDialog (null,
                             "Vill du fortsätta?",
                             "ConfirmDialog",
                             JOptionPane.YES_NO_OPTION,
                             JOptionPane.QUESTION_MESSAGE);
```

Det första argumentet representerar dialogens förälderkomponent, det andra argumentet det meddelande som visas i dialogrutan, och det tredje argumentet representerar dialogens titel. Det fjärde argumentet bestäm-

 © FADIL GALJIĆ

mer vilka alternativ som ska visas. En uppsättning alternativ preciseras med en lämplig konstant från klassen `JOptionPane`. Följande konstanter finns att välja mellan: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` och `OK_CANCEL_OPTION`. Det femte argumentet representerar meddelandets typ. Detta argument kan utelämnas. Om en egen ikon ska användas, anges den som sjätte argument.

Metoden `showConfirmDialog` returnerar ett heltalsvärde av typen `int`. Detta heltal representerar ordningsnumret för den valda knappen (det valda alternativet). Om dialogen stängs via dess stängningsknapp, returnerar metoden `-1`. För att avgöra vilket alternativ som valts, kan det returnerade värdet (`val`) jämföras med följande konstanter i klassen `JOptionPane`: `CLOSED_OPTION` (dialogrutan har stängts via dess stängningsknapp), `OK_OPTION`, `YES_OPTION`, `NO_OPTION` och `CANCEL_OPTION`. Man kan till exempel göra så här:

```
if (val == JOptionPane.YES_OPTION)
    System.out.println ("Vi fortsätter!");
else
    System.exit (0);
```

Metoden showOptionDialog

Metoden `showOptionDialog` visar en dialog med ett meddelande och en ikon, och en egen uppsättning alternativ. Dessa alternativ definieras som en vektor av objekt. Vanligtvis beskrivs olika alternativ via motsvarande teckensträngar. I så fall visas dessa teckensträngar på motsvarande knappar. Metoden `showOptionDialog` kan anropas så här:

```
Object[]    alternativ = { new String ("Jag vill!"),
                           new String ("Jag vill inte!") };
int    val = JOptionPane.showOptionDialog (null,
                             "Vill du fortsätta?",
                             "OptionDialog",
                             JOptionPane.DEFAULT_OPTION,
                             JOptionPane.QUESTION_MESSAGE,
                             null,
                             alternativ,
                             alternativ[0]);
```

Det första argumentet representerar dialogens förälderkomponent, det andra argumentet representerar det meddelande som visas i dialogrutan, och det tredje argumentet representerar dialogens titel. Det fjärde argumentet bestämmer alternativens typ (detta argument har inte någon speciell betydelse i detta fall, eftersom egna alternativ används). Det femte

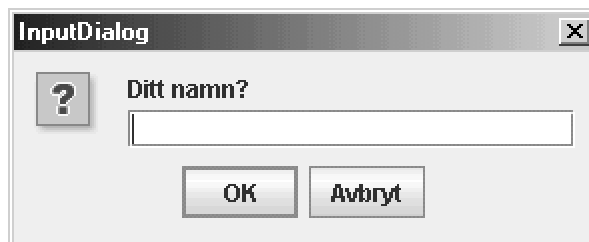
© FADIL GALJIĆ

argumentet representerar meddelandets typ och det sjätte argumentet representerar en egen ikon. De egna alternativen anges via det sjunde argumentet. Det sista argumentet anger det alternativ som ska vara förvalt.

Metoden `showOptionDialog` returnerar ett heltalsvärde av typen `int`. Detta heltal representerar ordningsnumret för det valda alternativet. Om dialogen stängs via dess stängningsknapp, returnerar metoden `-1`.

Metoden showInputDialog

Metoden `showInputDialog` visar en dialog med ett meddelande, en ikon, en `OK`-knapp, en `Avbryt`-knapp och ett textfält (se bilden nedan). Normalt skriver användaren in en teckensträng i textfältet och trycker på `OK`-knappen. Metoden returnerar den inmatade teckensträngen (som även kan vara en tom sträng) som ett objekt av typen `java.lang.Object`, och detta objekt måste därför omvandlas till motsvarande teckensträng. Metoden returnerar `null` om dialogen stängs på annat sätt än med `OK`-knappen.



Metoden `showInputDialog` kan anropas så här:

```
Object    input = JOptionPane.showInputDialog (null,
                              "Ditt namn?",
                              "InputDialog",
                              JOptionPane.QUESTION_MESSAGE);
```

Det första argumentet representerar dialogens förälderkomponent, det andra argumentet representerar det meddelande som visas i dialogrutan, och det tredje argumentet representerar dialogens titel. Det fjärde argumentet är meddelandets typ.

Metoden `showInputDialog` kan även visa en dialog som innehåller en valruta med olika alternativ (se bilden nedan) istället för ett textfält. Ett av dessa alternativ är förvalt, och det är detta alternativ som visas. Men det

 © FADIL GALJIĆ

går att öppna valrutan och välja ett annat alternativ. Valet bekräftas med en tryckning på `OK`-knappen.

En dialog med en valruta med olika alternativ kan skapas och visas så här:

```
Object[]    farger = { new String ("gul"), new String ("rosa"),
                       new String ("röd"), new String ("blå") };
Object    input = JOptionPane.showInputDialog (null,
                                  "Din färg?",
                                  "InputDialog",
                                  JOptionPane.QUESTION_MESSAGE,
                                  null,
                                  farger,
                                  farger[0]);
```

De fyra första argumenten har sina vanliga betydelser. Det femte argumentet är en egen ikon. Som sjätte argument anges olika alternativ, och det sjunde argumentet är det förvalda alternativet. De olika alternativen definieras i en vektor av objekt. Dessa objekt kan vara teckensträngar. Metoden `showInputDialog` returnerar i detta fall den valda teckensträngen (som ett objekt av typen `Object`).

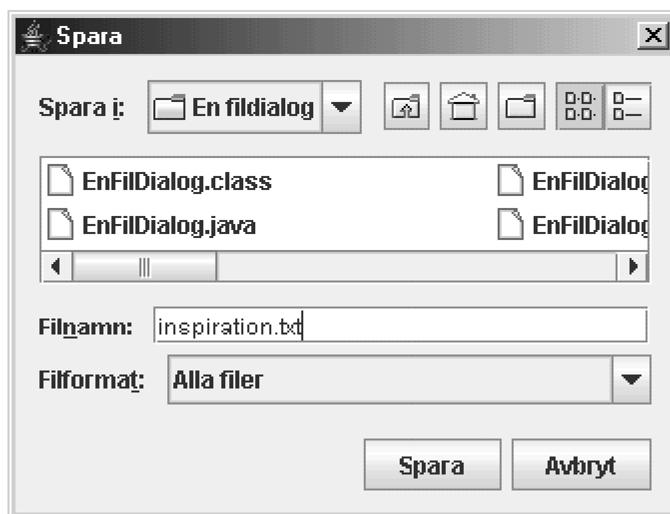
En fildialog

Välja en fil

Ett Javaprogram kan behöva information som finns på en fil på disken. Denna fil kan vara okänd när programmet skapas, så att filens namn, och dess plats i filsystemet, inte kan byggas in i programmet. Valet av filen kan överlämnas till programmets användare. Ett program kanske behöver visa en bild som finns i en fil. Programmet kan i så fall låta användaren välja mellan olika filer, och därmed välja olika bilder. En liknande situation kan uppstå när uppgifter ska sparas till en fil. Även i detta fall kan det vara vettigt att låta användaren välja filens namn och plats i filsystemet.

© FADIL GALJIĆ

En fils namn och plats i filsystemet kan anges på olika sätt. När man skapar ett grafiskt program, kan man använda en fildialog som skapas utifrån en dialogruta av typen `javax.swing.JFileChooser`. En sådan fildialog liknar de fildialoger som normalt används i olika program för att spara eller öppna en fil (se bilden nedan). Fildialogen gör det möjligt att navigera i filsystemet och välja kataloger och filer. För att öppna en fil används en `Öppna`-fildialog, och för att spara en fil används en `Spara`-fildialog.



En fildialog gör det möjligt för användaren att *välja* (`JFileChooser`) mellan olika filer, men inte att öppna eller spara en fil. Ytterligare kod, som gör detta, måste tillföras. För att användaren ska kunna öppna en fil, måste man skapa den kod som läser filen och som på något sätt presenterar den inlästa informationen. För att användaren ska kunna spara en fil, måste man skapa den kod som sparar aktuella uppgifter i filen. En fildialog är bara ett lämpligt sätt att ange en fils namn och dess plats i filsystemet.

Ett objekt av klassen `JFileChooser` representerar en fildialogruta. En fildialogruta är en Swing-komponent (klassen `JFileChooser` är en direkt subklass till klassen `javax.swing.JComponent`), men inte ett fönster (`JFileChooser` är inte en subklass till klassen `java.awt.Window`). En fildialogruta kan därför inte visas som ett självständigt fönster. Men klassen `JFileChooser` har metoderna `showDialog`, `showOpenDialog` och `showSaveDialog` som skapar en lämplig (modal) dialog, placerar fildialogrutan i denna dialog och visar dialogen. På så sätt fungerar en fildialogruta som en fildi-

 © FADIL GALJIĆ

alog när den används. Om man vill, kan man skapa en fildialogruta och placera den i en godtycklig dialog eller ram.

En fildialogruta kan skapas och visas i en `Spara`-dialog på följande vis:

```
File    startKatalog = new File (".");
JFileChooser    filValjare = new JFileChooser (startKatalog);
int    val = filValjare.showSaveDialog (null);
```

Här definieras en katalog via ett objekt av typen `java.io.File` (katalogen kan även definieras via en teckensträng, som anger katalogens sökväg). Utifrån denna katalog skapas en fildialogruta, som visas som en fildialog med metoden `showSaveDialog`. Fildialogens förälder anges som argument till denna metod (`null` innebär att fildialogen saknar förälderkomponent). I fildialogen visas katalogerna och filerna i den angivna katalogen (startkatalogen). Det är naturligtvis möjligt att bläddra mellan olika kataloger i filsystemet. Användaren väljer en katalog och en fil i denna katalog, och trycker på `Spara`-knappen. Det går även att skriva filens namn direkt i motsvarande textfält (data kan sparas i en ny fil).

När användaren stängt fildialogen, måste man ta reda på vilken fil som användaren valt. Programmet måste först kontrollera hur användaren stängt fildialogen. Detta sker genom att heltalsvärdet som returneras av metoden `showSaveDialog` analyseras. Detta heltalsvärde jämförs med olika konstanter i klassen `JFileChooser`. Om användaren tryckt på knappen `Spara`, så returnerar metoden `showSaveDialog` ett heltalsvärde som är lika med konstanten `APPROVE_OPTION`. Om användaren stängt dialogen med knappen `Avbryt` eller via dialogens stängningsknapp, returnerar metoden ett heltalsvärde som är lika med konstanten `CANCEL_OPTION`.

Metoden `getSelectedFile` ger den valda filen som ett objekt av typen `java.io.File`. Denna metod kan anropas så här:

```
File    fil = null;
if (val == JFileChooser.APPROVE_OPTION)
    fil = filValjare.getSelectedFile ();
```

När väl destinationsfilen har bestämts, kan olika uppgifter överföras till denna fil.

En fildialog som preciserar en fil som ska öppnas, kan skapas på ett liknande sätt. Till detta används metoden `showOpenDialog` istället för metoden `showSaveDialog`. Även metoden `showDialog`, som tar emot två argument, kan användas. Det första argumentet representerar dialogens förälderkomponent. Det andra argumentet är en teckensträng, som anger den text som placeras överst i dialogen och på motsvarande knapp i den högra

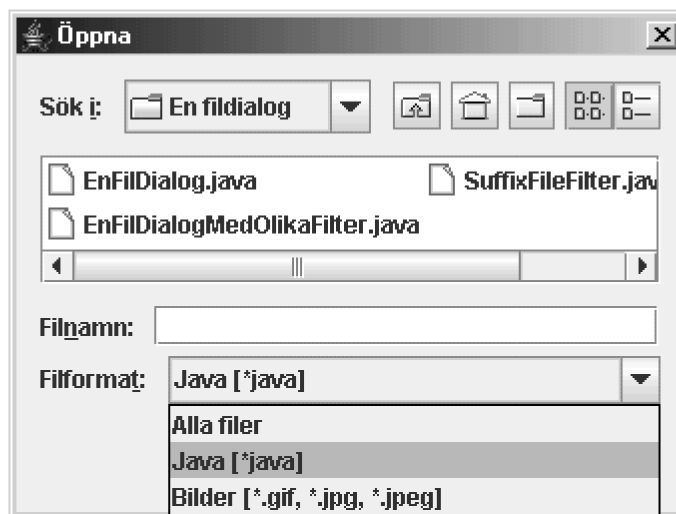
© FADIL GALJIĆ

nedre delen. Denna teckensträng kan vara `Öppna`, `Spara`, `Välj` eller något annat.

En fildialogruta kan hanteras på olika sätt. Man kan skapa en fildialogruta utan att ange något argument. I detta fall visas en fildialog med kataloger och filer i en förvald katalog (användarens hemkatalog). Med metoden `setCurrentDirectory` kan man välja vilken katalog som ska visas när fildialogrutan öppnas. Denna startkatalog anges som ett argument (av typen `File`) till metoden. En fildialog kan begränsas till endast kataloger, eller endast filer. Till detta används metoden `setFileSelectionMode`, och en lämplig konstant från klassen `JFileChooser` tillförs som argument till denna metod. En av följande konstanter kan användas: `FILES_ONLY`, `DIRECTORIES_ONLY` eller `FILES_AND_DIRECTORIES`. Det är också möjligt att ange en förvald fil, som visas i inmatningsfältet i fildialogen. Till detta används metoden `setSelectedFile`, och den förvalda filen anges som argument (av typen `File`) till denna metod.

Ett filfilter

När en katalogs innehåll visas i en fildialog, visas alla kataloger och filer i denna katalog. Ett förvalt filter som heter `Alla filer` används, och detta filter accepterar alla kataloger och filer. Men det finns även andra typer av filter som kan användas, och det innehåll som visas kan på så vis begränsas (se bilden nedan). Filtret kan till exempel acceptera bara Javafiler, och ignorera filer av andra typer.



© FADIL GALJIĆ

Ett eller flera filter kan läggas till i en fildialogruta. För att kunna göra detta, måste man först skapa en klass som definierar en filtertyp. Denna klass måste vara en subklass till klassen `javax.swing.filechooser.FileFilter` (paketet `java.io` innehåller ett gränssnitt med samma namn – dessa ska inte förväxlas). Klassen `FileFilter` är en abstrakt klass, som (bara) har två abstrakta metoder, metoderna `accept` och `getDescription`. Metoden `accept` tar emot en fil eller en katalog (ett objekt av typen `File`) som argument, och avgör om denna fil eller katalog ska accepteras eller inte. Metoden anger restriktioner, släpper fram eller förhindrar passering. Metoden `getDescription` returnerar en teckensträng, som representerar en beskrivning av ett filter. Det är denna beskrivning som visas i en fildialogruta.

Ett filter (mer exakt, en typ av filter) kan definieras så här:

```
class JavaFileFilter extends javax.swing.filechooser.FileFilter
{
    public boolean accept (java.io.File fil)
    {
        return fil.getName ().endsWith (".java");
    }

    public String getDescription ()
    {
        return "Java [*.java]";
    }
}
```

Ett filter av typen `JavaFileFilter` släpper bara fram Javafiler (de filer vars namn avslutas med `.java`). Ett sådant filter beskrivs i en fildialogruta med teckensträngen `Java [*.java]`.

Ett filter läggs till i en fildialogruta med metoden `addChoosableFileFilter`. Med metoden `setFileFilter` kan ett visst filter anges som förvalt filter i en fildialogruta. Ett filter av typen `JavaFileFilter` kan skapas, läggas till i en fildialogruta och väljas som förvalt filter, på följande vis:

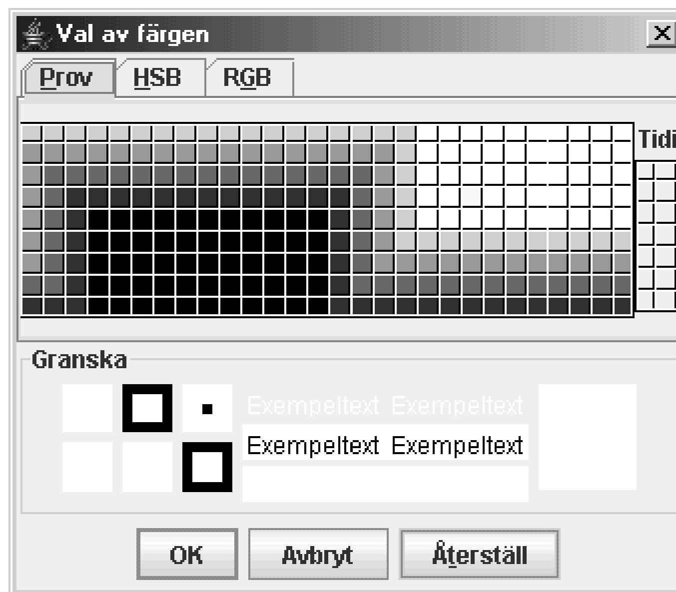
```
JFileChooser    filValjare = new JFileChooser (".");
JavaFileFilter    filter = new JavaFileFilter ();
filValjare.addChoosableFileFilter (filter);
filValjare.setFileFilter (filter);
```

Om man vill, kan man ta bort filtret `Alla filer` från en fildialogruta. Till detta används metoden `setAcceptAllFileFilterUsed`, och `false` anges som argument till metoden. Ett godtyckligt filter kan tas bort från en fildialogruta med metoden `removeChoosableFileFilter`.

© FADIL GALJIĆ

En färgdialog

Man kan skapa och visa en dialog som gör det möjligt att välja en färg. En av många fördefinierade färger, och olika nyanser, kan väljas (se bilden nedan). Användaren kan växla mellan tre olika lägen i dialogen (`Prov`, `HSB` och `RGB`), och få olika valmöjligheter.



En färgdialogruta representeras via ett objekt av typen `javax.swing.JColorChooser`. En dialogruta med färger är en komponent (ett objekt av typen `javax.swing.JComponent`), och inte ett fönster (inte ett objekt av typen `java.awt.Window`). Detta innebär att en sådan ruta inte kan visas som ett självständigt fönster. Den måste placeras i ett fönster (i en dialog eller en ram, till exempel) när den ska visas. Men det finns en genväg, som gör det möjligt att skapa och visa en färgdialog, och låta användaren välja en färg via denna färgdialog. Metoden `showDialog` i klassen `JColorChooser` kan användas. Det är en statisk metod som skapar en dialogruta med färger och en dialog, placerar dialogrutan i dialogen, och visar denna dialog. Metoden kan användas så här:

```
Color    farg = JColorChooser.showDialog (null,
                                          "Val av färgen",
                                           Color.WHITE);
```


© FADIL GALJIĆ

Dialogens förälderkomponent anges som första argument, dialogens rubrik som andra argument, och en förvald färg som tredje argument. Metoden returnerar den valda färgen som ett objekt av typen `java.awt.Color`. Användaren bekräftar sitt val med `OK`-knappen. Om dialogen stängs på något annat sätt, returnerar metoden `showDialog` `null`.

Den valda färgen kan användas på olika sätt. Den kan till exempel anges som bakgrundsfärg i en standarddialogruta:

```
JOptionPane    ruta = new JOptionPane ("Titta FÄRGEN!!!");
ruta.setBackground (farg);
```

© FADIL GALJIĆ

Kapitel 4
Grafik

Geometriska figurer
En punkt • En geometrisk figur • Linjer och kurvor
Rektanglar, ellipser och bågar • Areor • Sammansatta figurer

Rita i en panel
Rita figurer • Rita tecken • Rita figurer av olika typer

Hantera bilder
Lagra en bild • Spara en bild • Definiera en bilds pixlar
Bearbeta en bild • Skriva ut en bild

Olika rittekniker
Olika typer av linjer • Olika fyllnadsmönster • Rita inuti en figur
Förbättra en bilds kvalitet • Komposition av två bilder

Rörliga figurer
Flytta en figur • En figur som rör sig • Definiera en rörlig figur
Flera rörliga figurer

Koordinatbyte
Komponentkoordinater och användarkoordinater • En egen skala
Förflytta en figur • Rotera en figur • Skjuva en figur
Komposition av transformationer

© FADIL GALJIĆ

Geometriska figurer

En punkt

En punkt preciserar en plats (eng. location) i ett koordinatsystem. Man anger en punkt genom att ange dess koordinater. En punkt kan till exempel ha koordinaterna `3` och `4` enheter. Olika koordinatsystem kan använda olika enheter, till exempel centimeter, meter eller någon annan enhet.

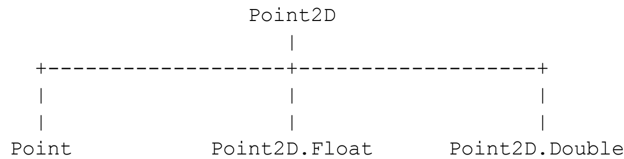
Paketet `java.awt.geom` innehåller en abstrakt klass, som definierar en punkt i ett plan. Det är klassen `Point2D`. I denna klass definieras flera metoder, som är gemensamma för alla subklasser till klassen `Point2D`. Metoderna `getX` och `getY`, (abstrakta metoder) som returnerar en punkts koordinater, definieras. Metoden `distance` returnerar avståndet mellan två punkter och metoden `distanceSq` returnerar kvadraten av detta avstånd. Klassen definierar även metoden `setLocation` som ändrar en punkts koordinater (dess placering), metoden `equals` som jämför två punkter, och så vidare.

I paketet `java.awt.geom` definieras tre icke-abstrakta subklasser till klassen `Point2D`. Det är klasserna `Point`, `Point2D.Float` och `Point2D.Double`. Alla dessa klasser har en punkts koordinater som publika medlemmar (en punkts koordinater kan kommas åt direkt, istället för via metoderna `getX` och `getY`). I klasserna definieras även en förvald (default) konstruktor (som skapar en punkt med koordinaterna `0` och `0`), och en konstruktor som tar emot en punkts koordinater och skapar en punkt utifrån dessa. Klasserna omdefinierar de abstrakta metoderna i superklassen `Point2D`, och metoden `toString` i klassen `java.lang.Object`.

Klassen `Point` definierar en punkt med heltalskoordinater av typen `int`. Klassen `Point2D.Float` definierar en punkt med flyttalskoordinater av typen `float`. Klassen `Point2D.Double` definierar en punkt med flyttalskoordinater av typen `double`. Klasserna `Point2D.Float` och `Point2D.Double` definieras som publika, statiska klasser (nästlade klasser) inuti klassen `Point2D` (de är subklasser till klassen `Point2D`, och dessutom definieras de inuti denna klass).

Den klasshierarkin (i paketet `java.awt.geom`) som hanterar en punkt i planet, kan representeras så här:

 © FADIL GALJIĆ



En punkt kan skapas så här:

```
Point2D.Double    p = new Point2D.Double (1.2, 6.0);
```

Även en referens av typen `Point2D` (en superklassreferens) kan användas för att referera till en punkt:

```
Point2D    p = new Point2D.Double (1.2, 6.0);
```

En punkts koordinater kan inte kommas åt direkt via en superklassreferens (superklassen `Point2D` känner inte till dem). Men med en sådan referens kan man använda alla de metoder som finns med i klassen `Point2D`, till exempel så här:

```
Point2D    p1 = new Point2D.Double (1.2, 6.0);
Point2D    p2 = new Point2D.Double (-2.2, 2.0);
double    d = p1.distance (p2);
p2.setLocation (0.0, 0.0);
```

Här skapas två punkter, och två superklassreferenser som refererar till dessa punkter. Sedan bestäms avståndet mellan de två punkterna. Därefter ändras placeringen av en av de två punkterna (den placeras i origo).

Punkter används även när man definierar och använder olika geometriska figurer.

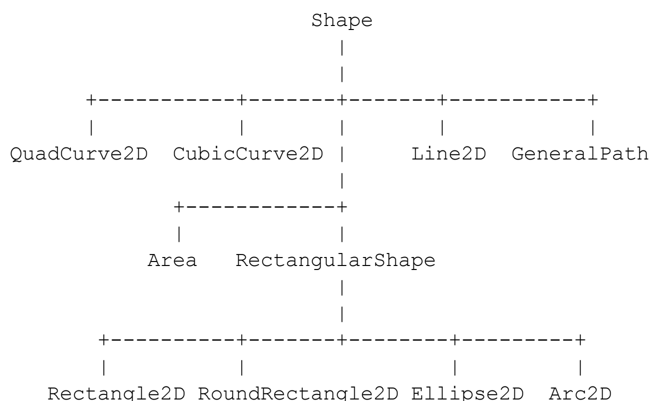
En geometrisk figur

En geometrisk figur definieras via gränssnittet `java.awt.Shape` (inte `java.awt.geom.Shape`). I detta gränssnitt definieras flera metoder som kan appliceras i samband med en figur. Metoden `contains` definieras i flera varianter. Metoden kontrollerar om en given punkt eller en given rektangulär area ligger inuti en figur. Metoden `intersects` kontrollerar om en figur skär en given rektangulär area. Gränssnittet definierar även metoden `getBounds2D` som returnerar den rektangel som omsluter en figur, och metoden `getPathIterator` som ger tillgång till en figurs kontur.

© FADIL GALJIĆ

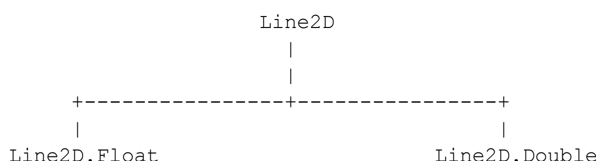
Paketet `java.awt.geom` innehåller flera klasser som implementerar gränssnittet `Shape`. En sådan klass definierar en geometrisk figur. Klassen `Line2D` definierar en rak linje, klassen `QuadCurve2D` definierar en kvadratisk kurva och klassen `CubicCurve2D` definierar en kubisk kurva. Klassen `GeneralPath` definierar en allmän figur, som kan omfatta flera linjära segment, kurvor och andra figurer. Klasserna `Area` och `RectangularShape` definierar plana figurer. Klassen `RectangularShape` är en abstrakt klass som har flera icke-abstrakta subklasser. Klasserna `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D` och `Arc2D` är subklasser till klassen `Rectangular Shape`.

Gränssnittet `Shape`, och den klasshierarkin som definierar olika geometriska figurer, kan representeras så här:



De klasser i denna klasshierarki vars namn avslutas med `2D` är abstrakta klasser. Var och en av dessa klasser har två icke-abstrakta subklasser. Dessa subklasser skiljer sig i hur de lagrar olika uppgifter om den figur som de representerar. Den ena subklassen lagrar dessa uppgifter som flyttalsvärden av typen `float`, och den andra subklassen lagrar motsvarande uppgifter som flyttalsvärden av typen `double`. Dessa två subklasser definieras som publika, statiska klasser (nästlade klasser) inuti motsvarande superklass. Klassen `Line2D`, till exempel, är en abstrakt superklass, som har en subklass som heter `Line2D.Float` och en subklass som heter `Line2D.Double`. Dessa två klasser definieras inuti superklassen `Line2D`. Motsvarande klasshierarkin kan representeras så här:

 © FADIL GALJIĆ



Metoderna i de klasser som definierar olika geometriska figurer har parametrar och returvärden av typen `double` (konstruktorer i `Float`-klasserna har parametrar av typen `float`). Därför är det bekvämare att använda de klasser vars namn avslutas med `Double`. De olika returvärdena behöver inte omvandlas till `float`, och man behöver inte lägga till `F` när ett konkret värde anges (som `3.2F`). Klasserna vars namn avslutas med `Float` kan användas när ett stort antal figurer skapas (för att spara minne, datatypen `float` tar upp mindre minne).

I gränssnittet `Shape` definieras flera metoder, som kan användas i samband med alla figurer (alla `Shape`-objekt). Metoden `getBounds2D` (som definieras i gränssnittet `Shape`), till exempel, kan användas för att få den rektangel som omsluter en figur. Denna rektangel erhålls som ett objekt av typen `Rectangle2D`.

Metoden `contains` (som definieras i gränssnittet `Shape`) kan användas för att avgöra om en given punkt eller en given rektangulär area finns (i sin helhet) inuti en figur. Denna metod tar emot antingen en punkt eller en rektangulär area, och returnerar ett booleskt värde. Argumentpunkten anges antingen via dess koordinater, eller som ett objekt av typen `Point2D`. En rektangulär area anges antingen som ett objekt av typen `Rectangle2D`, eller genom koordinaterna för rektangelns vänstra övre hörn och rektangelns bredd och höjd.

Metoden `intersects` (som definieras i gränssnittet `Shape`) kan användas för att avgöra om en figur har gemensamma inre punkter med en given rektangulär area. Denna metod tar emot en rektangulär area och returnerar ett booleskt värde. En rektangulär area anges antingen som ett objekt av typen `Rectangle2D`, eller genom koordinaterna för rektangelns vänstra övre hörn och rektangelns bredd och höjd.

© FADIL GALJIĆ

Linjer och kurvor

En linje

Ett linjesegment (och därmed en linje) definieras i klassen `Line2D` och dess subklasser `Line2D.Float` och `Line2D.Double`. Flera metoder i klassen `Line2D` är abstrakta och omdefinieras i klassens subklasser. Ett linjesegment bestäms med dess startpunkt och dess slutpunkt.

Ett linjesegment kan skapas så här:

```
Point2D.Double    p1 = new Point2D.Double (1, 2);
Point2D.Double    p2 = new Point2D.Double (3, 4);
Line2D.Double    linje = new Line2D.Double (p1, p2);
```

Ett linjesegments startpunkt och slutpunkt erhålls med metoderna `getP1` och `getP2`. Istället för att ange punkter, kan motsvarande koordinater anges när ett linjesegment skapas:

```
Line2D.Double    linje = new Line2D.Double (1, 2, 3, 4);
```

Objektet (som refereras av referensen) `linje` representerar ett linjesegment mellan punkterna (`1, 2`) och (`3, 4`). En punkts koordinater är publika medlemmar i klassen `Line2D.Double` (och i klassen `Line2D.Float`) med namnen `x1`, `y1`, `x2` och `y2`. Dessa koordinater går därför att komma åt direkt (till exempel med `linje.x1`, `linje.y1`, `linje.x2` och `linje.y2`).

En referens av typen `Line2D` kan användas för att referera till olika objekt av typen `Line2D.Float` och till olika objekt av typen `Line2D.Double`. Alla relevanta metoder definieras i klassen `Line2D`, och en referens av denna typ känner till dessa metoder och kan aktivera dem. Det går att göra så här:

```
Line2D    linje = new Line2D.Double (p1, p2);
```

Om en superklassreferens (en referens av typen `Line2D`) används, kan inte koordinaterna för ett linjesegments startpunkt och slutpunkt kommas åt direkt (de är publika medlemmar i klasserna `Line2D.Double` och `Line2D.Float`, inte i klassen `Line2D`). I detta fall måste man använda metoderna `getX1`, `getY1`, `getX2` och `getY2`.

Förutom de metoder som definieras i gränssnittet `Shape`, definieras i klassen `Line2D` även flera linjespecifika metoder. Metoden `intersectsLine` kontrollerar om ett linjesegment skär ett annat linjesegment. Ett av dessa linjesegment tillförs som argument till metoden. Detta linjesegment kan anges antingen som ett objekt eller via koordinaterna för dess ändpunkter. Metoden `ptSegDist` returnerar avståndet från en given punkt (som

 © FADIL GALJIĆ

anges som ett objekt eller via dess koordinater) till ett linjesegment. Metoden `ptSegDistSq` returnerar kvadraten av detta avstånd. Ett linjesegment bestämmer en (obegränsad) linje som även går bortom ändpunkterna för detta linjesegment. Avståndet mellan en punkt och denna linje (den kortaste vägen) kan bestämmas med metoden `ptLineDist` (eller via metoden `ptLineDistSq`, som returnerar kvadraten av detta avstånd). Ett linjesegments placering kan ändras med metoden `setLine` (nya ändpunkter anges som argument).

En kvadratisk kurva

I klassen `QuadCurve2D` och dess subklasser `QuadCurve2D.Double` och `QuadCurve2D.Float` definieras ett segment av en kvadratisk kurva (och därmed en kvadratisk kurva). Man definierar ett sådant segment genom att ange segmentets startpunkt, kontrollpunkt och slutpunkt. En kontrollpunkt bestämmer ett kurvsegments utseende. Linjer från en kontrollpunkt genom ett kurvsegments ändar är tangenter till detta kurvsegment i dessa ändar. Genom att ändra kontrollpunktens position, ändrar man ett kurvsegments utseende (se nedanstående figurer).

Man kan skapa ett kvadratiskt kurvsegment genom att ange kurvsegmentets startpunkt, kontrollpunkt och slutpunkt som argument till motsvarande konstruktorer. Punkterna anges via dess koordinater, till exempel så här:

```
QuadCurve2D    kurva = new QuadCurve2D.Double (0.0, 0.0,
                                               1.0, 4.0,
                                               2.0, 0.0);
```

Här skapas ett kvadratiskt kurvsegment mellan punkterna `(0.0, 0.0)` och `(2.0, 0.0)`, med kontrollpunkten `(1.0, 4.0)`. Koordinaterna för alla dessa punkter lagras som publika medlemmar med namnen `x1`, `y1`, `ctrlx`, `ctrly`, `x2` och `y2`. Dessa punkter kan antingen kommas åt direkt, eller via metoderna `getX1`, `getY1`, `getCtrlX`, `getCtrlY`, `getX2` och `getY2`. Om en superklassreferens (av typen `QuadCurve2D`) används, kan bara motsvarande metoder användas. De motsvarande uppgifterna kan även erhållas via de metoder som returnerar startpunkten, kontrollpunkten och slutpunkten

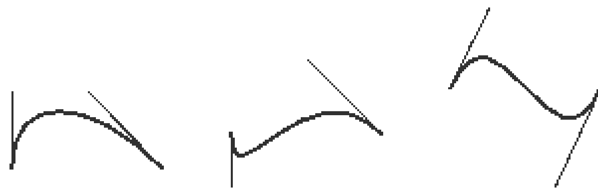
© FADIL GALJIĆ

som objekt av typen `Point2D`. Dessa metoder är `getP1`, `getCtrlPt` (inte `getCtrlP`) och `getP2`. Ett kurvsegment kan ändras med metoden `setCurve` (nya punkter anges som argument – antingen som objekt av typen `Point2D` eller via deras koordinater).

Klassen `QuadCurve2D` definierar den statiska metoden `solveQuadratic`, som hittar reella lösningar för en kvadratisk ekvation med reella koefficienter. Denna metod tar emot en flyttalsvektor som representerar ekvationen (den innehåller ekvationens koefficienter - ekvationen $ax^2 + bx + c = 0$ lagras som en vektor som innehåller c, b och a), och en flyttalsvektor som representerar ekvationens rötter (om endast koefficentvektorn anges som argument, lagras rötterna där). Metoden löser ekvationen, lagrar de reella (icke-komplexa) rötterna i motsvarande vektor, och returnerar antalet reella rötter. Om ekvationen är omöjlig, eller om den alltid är uppfylld, returnerar metoden `-1`.

En kubisk kurva

I klassen `CubicCurve2D` och dess subklasser `CubicCurve2D.Double` och `CubicCurve2D.Float` definieras ett segment av en kubisk kurva (och därmed en kubisk kurva). Ett sådant segment definieras genom att segmentets startpunkt, två kontrollpunkter och dess slutpunkt anges. Kontrollpunkterna bestämmer ett kurvsegments utseende. Linjen från den första kontrollpunkten genom ett kurvsegments startpunkt är tangenten till kurvsegmentet i denna startpunkt. Linjen från den andra kontrollpunkten genom ett kurvsegments slutpunkt är tangenten till detta kurvsegment i den slutpunkten. Genom att ändra kontrollpunkternas positioner, ändrar man ett kurvsegments utseende (se bilden nedan).



Man kan skapa ett kubiskt kurvsegment genom att ange startpunkten, två kontrollpunkter och slutpunkten som argument till motsvarande konstruktorer. Punkterna anges via deras koordinater, till exempel så här:

```
CubicCurve2D    kurva = new CubicCurve2D.Double (0.0, 0.0,
                                                 1.0, 4.0,
                                                 2.0, -4.0,
                                                 3.0, 0.0);
```


© FADIL GALJIĆ

Här skapas ett kubiskt kurvsegment mellan punkterna `(0.0, 0.0)` och `(3.0, 0.0)`, med kontrollpunkterna `(1.0, 4.0)` och `(2.0, -4.0)`. Koordinaterna för alla dessa punkter lagras som publika medlemmar med namnen `x1`, `y1`, `ctrlx1`, `ctrly1`, `ctrlx2`, `ctrly2`, `x2` och `y2`. Dessa koordinater kan kommas åt direkt, eller via metoderna `getX1`, `getY1`, `getCtrlX1`, `getCtrlY1`, `getCtrlX2` och `getCtrlY2`, `getX2` och `getY2`. Om en superklassreferens (av typen `CubicCurve2D`) används, kan endast motsvarande metoder användas. Motsvarande uppgifter kan även erhållas via de metoder som returnerar startpunkten, kontrollpunkterna och slutpunkten som objekt av typen `Point2D`. Dessa metoder är `getP1`, `getCtrlP1`, `getCtrlP2` och `getP2`. Ett kurvsegment kan ändras med metoden `setCurve` (nya punkter anges som argument – punkterna anges antingen som objekt av typen `Point2D`, eller via deras koordinater).

Klassen `CubicCurve2D` definierar den statiska metoden `solveCubic`, som hittar reella lösningar för en kubisk ekvation med reella koefficienter. Metoden tar emot en flyttalsvektor som representerar ekvationen (den innehåller ekvationens koefficienter – ekvationen $ax^3 + bx^2 + cx + d = 0$ lagras som en vektor som innehåller d, c, b och a), och en flyttalsvektor för ekvationens rötter (om endast koefficientvektorn anges som argument, lagras rötterna där). Metoden löser ekvationen, lagrar de reella (icke-komplexa) rötterna i motsvarande vektor, och returnerar antalet reella rötter. Om ekvationen är omöjlig, eller om den alltid är uppfylld, returnerar metoden `-1`.

En sammansatt linje

Klasserna `Line2D`, `QuadCurve2D`, `CubicCurve2D` och dess subklasser definierar ett linjesegment, ett kvadratiskt kurvsegment och ett kubiskt kurvsegment. Dessa segment kan kombineras till en sammansatt linje. En sådan linje representeras med ett objekt av typen `GeneralPath`. Med ett sådant objekt kan man representera en triangel, en polygon, en kombination av olika kurvsegment, en kombination av olika linjesegment och kurvsegment, och så vidare (se bilderna nedan).

En triangel kan skapas så här:

© FADIL GALJIĆ

```
GeneralPath    triangel = new GeneralPath ();
triangel.moveTo (0.0f, 0.0f);
triangel.lineTo (2.0f, 2.0f);
triangel.lineTo (4.0f, 0.0f);
triangel.closePath ();
```

Triangeln påbörjas i punkten `(0.0, 0.0)`. Startpunkten anges med metoden `moveTo`, som tar emot två argument av typen `float`. Därefter läggs ett linjesegment från startpunkten till punkten `(2.0, 2.0)`. Detta sker med metoden `lineTo` (även denna metod har parametrar av typen `float`). Detta följs av ett linjesegment från den aktuella punkten (punkten `(2.0, 2.0)`) till punkten `(4.0, 0.0)`, och sedan stängs linjen (ett linjesegment läggs till från den aktuella punkten till startpunkten med metoden `closePath`). På så vis skapas en triangel.

Ett kvadratiskt kurvsegment läggs till i en sammansatt linje med metoden `quadTo`. Denna metod tar emot koordinater för kontrollpunkten och koordinater för slutpunkten som sina argument (av typen `float`). Ett kubiskt kurvsegment läggs till i en sammansatt linje med metoden `curveTo` (inte `cubicTo`). Denna metod tar emot koordinaterna för kontrollpunkterna och slutpunkten som argument (av typen `float`). En sammansatt linje som består av ett kvadratiskt kurvsegment, ett linjesegment och ett kubiskt kurvsegment kan skapas så här:

```
GeneralPath    path = new GeneralPath ();
path.moveTo (0.0f, 0.0f);
path.quadTo (1.0f, 2.0f, 2.0f, 0.0f);
path.lineTo (4.0f, 0.0f);
path.curveTo (5.0f, 2.0f, 5.5f, -2.0f, 6.0f, 0.0f);
```

Först skapas ett kvadratiskt kurvsegment mellan punkterna `(0.0, 0.0)` och `(2.0, 0.0)`, med kontrollpunkten `(1.0, 2.0)`. Sedan placeras ett linjesegment mellan punkterna `(2.0, 0.0)` och `(4.0, 0.0)`. Slutligen placeras ett kubiskt kurvsegment mellan punkterna `(4.0, 0.0)` och `(6.0, 0.0)`, med kontrollpunkterna `(5.0, 2.0)` och `(5.5, -2.0)`. Linjen stängs inte. En sammansatt linje kan vara öppen.

En sammansatt linje behöver inte vara sammanhängande. Den kan bestå av flera separata delar. Linjen kan när som helst avbrytas med metoden `moveTo`. Sedan placeras ett segment från den punkt som anges som argument till metoden `moveTo`. Man kan till exempel göra så här:

```
GeneralPath    path = new GeneralPath ();
path.moveTo (0.0f, 0.0f);
path.lineTo (2.0f, 0.0f);
path.moveTo (0.0f, 1.0f);
path.lineTo (2.0f, 3.0f);
```


© FADIL GALJIĆ

På detta vis skapas en sammansatt linje som består av två separata linjesegment. Det ena linjesegmentet går från punkten (0.0, 0.0) till punkten (2.0, 0.0), och det andra linjesegmentet går från punkten (0.0, 1.0) till punkten (2.0, 3.0).

Den aktuella punkten (punkten för den aktuella positionen då ett antal segment placerats) erhålls med metoden getCurrentPoint (metoden returnerar en punkt som ett objekt av typen Point2D). Ett objekt av typen GeneralPath kan rensas från alla komponenter (och göras till en tom linje) med metoden reset.

Rektanglar, ellipser och bågar

Rektanglar

En rektangel definieras i klassen Rectangle2D och i dess subklasser Rectangle2D.Double och Rectangle2D.Float. Man definierar en rektangel genom att ange dess vänstra övre hörn och dess bredd och höjd (se bilden nedan).

En rektangel kan skapas så här:

```
Rectangle2D    rek = new Rectangle2D.Double (
                              10.0, 20.0, 60.0, 40.0);
```

Här skapas en rektangel med det övre vänstra hörnet i punkten (10.0, 20.0), vars bredd är 60.0 och vars höjd är 40.0 enheter. Dessa uppgifter lagras som publika instansvariabler, och man kan komma åt dessa värden via deras namn. Dessa namn är x och y (för koordinaterna), width (för bredden) och height (för höjden). Om en superklassreferens (av typen Rectangle2D) används, kan dessa värden endast kommas åt via motsvarande metoder. Dessa metoder är getX, getY, getWidth och getHeight. En rektangels position och storlek kan ändras med metoden setRect. Rektangelns nya hörn (dess koordinater), bredd och höjd anges som argument till metoden.

Man kan undersöka om en rektangels inre del har gemensamma punkter med ett linjesegment. Till detta används metoden intersectsLine, och koordinaterna för linjesegmentets startpunkt och slutpunkt tillförs som

argument till metoden. Det går även att tillföra ett objekt av typen `Line2D` som argument.

Man kan skapa en kvadrat genom att skapa en rektangel vars bredd och höjd är lika.

Även en rektangel med rundade hörn kan skapas. Sådana rektanglar definieras i klassen `RoundRectangle2D` och dess subklasser `RoundRectangle2D.Double` och `RoundRectangle2D.Float`. När man skapar en rektangel med runda hörn, utgår man ifrån den rektangel som omger den runda rektangeln (se bilden nedan). Rektangelns vänstra övre hörn, och dess bredd och höjd anges.

En rektangel med rundade hörn kan skapas så här:

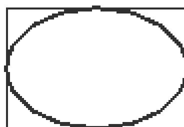
```
RoundRectangle2D    rrek = new RoundRectangle2D.Double (
                           10.0, 20.0, 60.0, 40.0, 6.0, 4.0);
```

Här anges koordinaterna för motsvarande rektangels vänstra övre hörn, rektangelns bredd och dess höjd. Även bredden och höjden för den runda delen anges (bredden och höjden för den imaginära rektangel som finns i ett hörn, och som omsluter den ellips varav en del utgör rundningen). Dessa uppgifter lagras som publika instansvariabler med namnen `x`, `y`, `width`, `height`, `arcwidth` och `archeight`. Om en superklassreferens (av typen `RoundRectangle2D`) används kan dessa värden endast kommas åt via motsvarande metoder. Dessa metoder är `getX`, `getY`, `getWidth`, `getHeight`, `getArcWidth` och `getArcHeight`. Värdena kan ändras med metoden `setRoundRect`. Nya värden anges som argument till denna metod.

Ellipser

En ellips definieras i klassen `Ellipse2D` och dess subklasser `Ellipse2D.Double` och `Ellipse2D.Float`. Man definierar en ellips genom att definiera ellipsens omslutande rektangel (se bilden nedan). Rektangelns vänstra övre hörn (dess koordinater), bredd och höjd anges. Ellipsens två diametrar motsvarar denna rektangels bredd och höjd.

© FADIL GALJIĆ



En ellips kan skapas så här:

```
Ellipse2D    el = new Ellipse2D.Double (10.0, 20.0, 60.0, 40.0);
```

Här anges (koordinater för) vänstra övre hörnet, bredden och höjden för den rektangel som omger ellipsen. Dessa uppgifter lagras som publika instansvariabler, som kan kommas åt via deras namn. Namnen är `x` och `y` (för koordinaterna), `width` (för bredden) och `height` (för höjden). Om en superklassreferens (av typen `Ellipse2D`) används, kan dessa uppgifter endast kommas åt via motsvarande metoder. Dessa metoder är `getX`, `getY`, `getWidth` och `getHeight`.

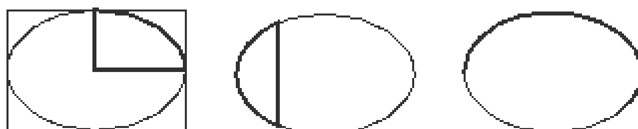
En ellips position och storlek kan ändras med metoden `setFrame`. En ny omslutande rektangel (via dess hörn, bredd och höjd) anges som argument till metoden. Den rektangel som omsluter en ellips erhålls med metoden `getFrame`, eller med metoden `getBounds2D`. Dessa metoder returnerar den omslutande rektangeln som ett objekt av typen `Rectangle2D`.

Man kan skapa en cirkel genom att skapa en ellips vars två diametrar är likadana.

Bågar

En båge definieras i klassen `Arc2D` och i dess subklasser `Arc2D.Double` och `Arc2D.Float`. Man definierar en båge genom att definiera motsvarande ellips (den ellips på vilken bågen ligger), och startpunkten och slutpunkten på ellipsen (se figurerna nedan). Man definierar denna ellips genom att definiera dess omslutande rektangel. Startpunkten kan preciseras genom den vinkel som linjen genom ellipsens mittpunkt och denna punkt bildar mot den positiva delen av x-axeln. Bågens slutpunkt kan preciseras genom bågens vinkel (den vinkel som uppstår när bågens ändpunkter förbinds med ellipsens mittpunkt). En båges ändpunkter kan vara bundna eller inte. De kan vara direkt bundna, eller indirekt, via ellipsens mittpunkt. Hur dessa två punkter är bundna (och om de är bundna) preciseras med en lämplig konstant från klassen `Arc2D`. En av följande konstanter kan väljas: `PIE` (punkterna bundna till ellipsens mittpunkt), `CHORD` (punkterna direkt bundna) och `OPEN` (punkterna inte bundna).

© FADIL GALJIĆ



En båge kan skapas så här:

```
Arc2D    b = new Arc2D.Double (10.0, 20.0, 60.0, 40.0,
                               0, 90, Arc2D.PIE);
```

Här preciseras den omgivande rektangeln (och därmed den ellips på vilken bågen ligger) genom koordinater för dess vänstra över hörn (`10.0 och 20.0`), dess bredd (`60.0`) och dess höjd (`40.0`). Sedan anges bågens startvinkel (`0` grader moturs) och bågens vinkel (`90` grader moturs). Slutligen preciseras att bågens ändpunkter ska bindas med rektangelns (ellipsens) mittpunkt (`Arc2D.PIE`). Den omgivande rektangeln kan även preciseras med ett objekt av typen `Rectangle2D` som (första) argument.

Uppgifter om en båge lagras som publika instansvariabler. Namnen på de variabler som beskriver den omgivande rektangeln är `x`, `y`, `width` och `height`. Namnet på en båges startvinkel är `start` och namnet på en båges vinkel är `extent`. Om en superklassreferens (av typen `Arc2D`) används, kan dessa uppgifter endast erhållas via motsvarande metoder. Dessa metoder är `getX`, `getY`, `getWidth`, `getHeight`, `getAngleStart` och `getAngleExtent`. Metoden `getArcType` ger en båges typ (egentligen den konstant som beskriver hur bågens ändpunkter är bundna).

En båges position, storlek, vinklar och typ kan ändras med metoden `setArc`. Nya uppgifter anges som argument till metoden (som när en båge skapas). En båges startvinkel kan ändras med metoden `setAngleStart`, en båges vinkel med metoden `setAngleExtent`, och en båges typ med metoden `setArcType`.

Metoderna `getBounds2D` och `getFrame` ger en båges omgivande rektangel (som ett objekt av typen `Rectangle2D`). En båges omgivande rektangel kan anges med metoden `setFrame` (koordinaterna för rektangelns hörn, bredd och höjd anges som argument). Metoden `getStartPoint` ger en båges startpunkt (som ett objekt av typen `Point2D`), och metoden `getEndPoint` ger en båges slutpunkt. En båges startpunkt och slutpunkt kan anges med metoden `setAngles` (genom att ange en båges ändpunkter preciserar man dess vinklar). Dessa två punkter anges som argument till metoden (antingen via deras koordinater eller som två objekt av typen `Point2D`). Med metoden `containsAngle` kan man kontrollera om en given vinkel hamnar

 © FADIL GALJIĆ

i en båges vinkelområde. Den givna vinkeln tillförs metoden som argument.

En båge som ligger på en cirkel kan skapas. Cirkelns mittpunkt (dess koordinater) och radie anges, samt bågens startvinkel, vinkel och typ. Dessa uppgifter tillförs som argument till metoden `setArcByCenter`:

```
Arc2D    b = new Arc2D.Double ();
b.setArcByCenter (60.0, 40.0, 20.0, 0, 90, Arc2D.PIE);
```

Rektangulära figurer

Klasserna `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D` och `Arc2D` har en gemensam superklass. Det är den abstrakta klassen `RectangularShape`. Metoder från den klassen kan appliceras i samband med såväl rektanglar och runda rektanglar, som ellipser och bågar. Alla dessa figurer har en omgivande rektangel (eller är en rektangel), och metoder från klassen `RectangularShape` hanterar denna rektangel på olika sätt.

Metoden `getFrame` ger en figurs omgivande rektangel (som ett objekt av typen `Rectangle2D`). En figurs omgivande rektangel kan anges med metoden `setFrame`, som tar emot en rektangel som argument. Denna rektangel specificeras antingen som ett objekt av typen `Rectangle2D` eller via koordinaterna för dess vänstra övre hörn, dess bredd och höjd. En figurs omgivande rektangel kan även specificeras utifrån rektangelns mittpunkt och vänstra övre hörn. Dessa två punkter anges (antingen via deras koordinater eller som två objekt av typen `Point2D`) som argument till metoden `setFrameFromCenter`. Metoden `setFrameFromDiagonal` anger en figurs omgivande rektangel utifrån rektangelns vänstra övre hörn och högra nedre hörn.

Metoderna `getWidth` och `getHeight` ger bredden och höjden av en figurs omgivande rektangel. Metoderna `getX`, `getY`, `getCenterX` och `getCenterY` ger koordinaterna för rektangelns vänstra övre hörn och dess mittpunkt. Metoderna `getMaxX`, `getMaxY`, `getMinX` och `getMinY` ger de största och minsta värdena för koordinaterna `x` och `y` på den omgivande rektangeln.

Areor

Objekt av typen `Area` kan användas för att representera figurer av olika former. Ett objekt av typen `Area` skapas utifrån ett objekt som implementerar gränssnittet `Shape`. Så här kan man göra:

© FADIL GALJIĆ

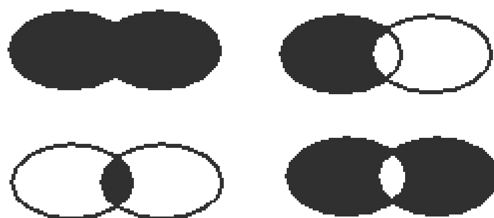
```
Ellipse2D    ellips1 = new Ellipse2D.Double (100, 50, 80, 50);
Ellipse2D    ellips2 = new Ellipse2D.Double (160, 50, 80, 50);
Area    area1 = new Area (ellips1);
Area    area2 = new Area (ellips2);
```

Två olika areor skapas utifrån två givna ellipser.

Två areor kan kombineras på olika sätt (se bilderna nedan). Metoden `add`, till exempel, kan användas för att lägga till en figur till en annan figur (för att utöka den aktiverande figuren med den figur som anges som argument). Detta kan göras så här:

```
area1.add (area2);
```

Figuren `area1` utökas med figuren `area2`. Även andra operationer kan utföras med två areor. Metoden `subtract` kan användas för att ta bort den del av en area som även tillhör en annan area. Metoden `intersect` kan användas för att ta bort den del av en area som inte tillhör en annan area (för att endast behålla det som är gemensamt). Metoden `exclusiveOr` kan användas för att lägga till en area till en annan area, och för att sedan ta bort den gemensamma delen.



Metoden `isEmpty` kontrollerar om en area är tom. Metoden `reset` gör en area till en tom area.

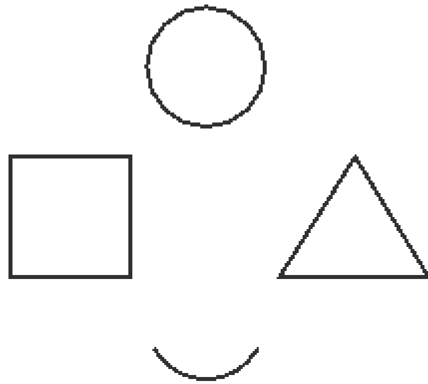
Med metoden `isRectangular` kontrollerar man om en area är rektangulär, och metoden `isPolygonal` kontrollerar om en area har formen av en polygon. Metoden `isSingular` kontrollerar om en area är sammanhängande.

Sammansatta figurer

En figur av typen `GeneralPath` kan bestå av flera linjesegment och/eller flera kurvsegment. En figur av denna typ kan även innehålla rektangulära figurer. Metoden `append` (i klassen `GeneralPath`) kan användas för att lägga till en godtycklig figur (ett objekt av en klass som implementerar gränssnittet `Shape`) till ett objekt av typen `GeneralPath`. På så sätt skapas

© FADIL GALJIĆ

en sammansatt figur, som består av flera andra figurer (`Shape-s`) (se bilden nedan).



Så här kan man göra:

```
Rectangle2D    rektangel =
                    new Rectangle2D.Double (100, 120, 120, 80);
Ellipse2D    ellips = new Ellipse2D.Double (100, 240, 120, 80);
GeneralPath    figur = new GeneralPath ();
figur.moveTo (100, 70);
figur.lineTo (220, 70);
figur.append (rektangel, false);
figur.append (ellips, false);
```

Här skapas en sammansatt figur (av typen `GeneralPath`), som innehåller ett linjesegment, en rektangel och en ellips. Metoden `append` tar emot den figur som ska läggas till och ett booleskt värde som sina argument. Normalt används `false` som andra argument. Om `true` används, kan den figur som läggs till bindas med ett linjesegment med resten av figuren. Detta kan dock även göras direkt, på ett mer uppenbart sätt.

Med metoden `reset` kan alla figurer tas bort från ett objekt av typen `GeneralPath` (objektet görs till en tom figur).

Rita i en panel

Rita figurer

Definiera en egen panel

En figur (en `Shape`) kan ritas i en godtycklig grafisk komponent (ett objekt av någon subklass till klassen `javax.swing.JComponent`). Även om figurer kan ritas i olika komponenter, väljer man vanligtvis en panel (ett objekt av typen `javax.swing.JPanel`) som kanvas. Man skapar en klass som definierar en panel, och ritar i metoden `paintComponent` i den klassen.

För att kunna rita en figur i en panel, måste figurens position i panelen preciseras. Figurens position anges relativt panelens koordinatsystem. Detta koordinatsystem har origo i panelens vänstra övre hörn. Koordinatsystemets x-axel går från origo till höger, och dess y-axel går från origo neråt. En enhet i koordinatsystemet är en pixel. Positioner och dimensioner uttrycks i antal pixlar.

En figur definieras som ett objekt av en klass som implementerar gränssnittet `java.awt.Shape`. Det kan vara en linje (ett objekt av typen `java.awt.geom.Line2D`), en rektangel (ett objekt av typen `java.awt.geom.Rectangle2D`), en ellips (ett objekt av typen `java.awt.geom.Ellipse2D`), eller någon annan figur.

För att rita en rektangel representerar man först rektangeln som ett objekt av typen `Rectangle2D`:

```
Rectangle2D    rek = new Rectangle2D.Double (90, 40, 120, 80);
```

En rektangel skapas, som har det vänstra övre hörnet i punkten `(90, 40)`, bredden `120` och höjden 80 enheter.

För att kunna rita en figur, krävs det ett objekt som kan rita figurer. Ett objekt av typen `java.awt.Graphics2D` kan göra detta. Objektets `draw`-metod används, och figuren som ska ritas anges som argument till metoden. För att fylla hela figuren med färg används metoden `fill`. Ett objekt av typen `Graphics2D` har olika ritmöjligheter. Olika inställningar i objektet justeras för att önskad grafik ska kunna skapas. En figur kan ritas i olika kontext. En figur kan ritas på en skärm, i datorns minne eller skrivas ut på en skrivare. Ett objekt av typen `Graphics2D` skapas för en sådan kontext. Därför brukar ett sådant objekt kallas för *grafisk kontext* (objektet representerar olika grafiska möjligheter i en given kontext).

© FADIL GALJIĆ

Ett objekt av typen Graphics2D skapas inte direkt (klassen Graphics2D saknar publika konstruktorer). Ett sådant objekt erhålls indirekt, när panelens paintComponent-metod omdefinieras (denna metod definieras i panelens superklass JComponent, och klassen JPanel ärver metoden). Java anropar metoden varje gång en panel visas. Metoden tar emot ett argument av typen java.awt.Graphics. Detta argument skapas automatiskt, och tillförs till metoden paintComponent. Ett objekt av typen Graphics kan användas för att skapa ett slags grafik. Men ett objekt av typen java.awt.Graphics2D erbjuder mycket större möjligheter. Klassen Graphics2D är en subklass till klassen Graphics, och metoderna draw och fill definieras i denna subklass. Därför är det bättre att använda ett objekt av typen Graphics2D istället för ett objekt av typen Graphics.

Java anropar en panels paintComponent-metod (vi anropar inte denna metod direkt) varje gång panelen visas. Ett objekt av typen Graphics2D skapas och tillförs som argument till metoden. Metoden paintComponent har en parameter av typen Graphics, men den kan även ta emot ett objekt av typen Graphics2D (eftersom klassen Graphics2D är en subklass till klassen Graphics – en referens av typen Graphics kan även referera till ett objekt av typen Graphics2D). Ett objekt av typen Graphics2D tillförs som argument, och därför kan parameterreferensen (av typen Graphics) omvandlas till en referens av typen Graphics2D. Denna referens kan sedan användas till att aktivera metoderna draw och fill (en referens av typen Graphics käner inte dessa metoder).

För att kunna rita olika figurer, skapar man en klass som definierar en typ av paneler. Denna klass skapas som en subklass till klassen javax.swing.JPanel. I subklassen omdefinieras metoden paintComponent, och olika figurer ritas i denna metod. Så här, till exempel:

```
class RitPanel extends JPanel
{
    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        this.setBackground (Color.WHITE);
        Graphics2D    g = (Graphics2D) gr;

        Rectangle2D    rek =
                new Rectangle2D.Double (90, 40, 120, 80);
        g.draw (rek);
    }
}
```

Klassen RitPanel definierar en typ av paneler (eftersom den är en subklass till klassen JPanel). Metoden paintComponent omdefinieras, och på så sätt

preciseras hur en panel av typen `RitPanel` kommer att se ut. Panelens bakgrundsfärg anges, och en rektangel ritas i panelen. Varje gång panelen visas, sätts panelens bakgrundsfärg till den angivna färgen och den angivna rektangeln ritas i panelen.

När metoden `paintComponent` omdefinieras, anropas först superklassens `paintComponent`-metod (som definieras i klassen `JComponent`, och ärvs i klassen `JPanel`). Superklassens metod utför vissa nödvändiga operationer, och därför ska anropet till denna metod placeras i början i den omdefinierade metoden.

Justera en linjes tjocklek

En figur ritas med en linje som har en viss tjocklek. Den förvalda tjockleken är `1` enhet, men kan justeras. En linjes tjocklek representeras med ett objekt av typen `java.awt.BasicStroke`. Tjockleken anges som ett argument (av typen `float`) när ett objekt av denna typ skapas. Så här, till exempel:

```
BasicStroke    stroke = new BasicStroke (2.0f);
```

En figur ritas med ett objekt av typen `Graphics2D`. Detta objekt innehåller information om den aktuella tjockleken. Tjockleken anges med metoden `setStroke`. Det går att göra så här:

```
g.setStroke (stroke);
g.draw (rek);
```

Linjes tjocklek justeras (tjockleken anges via objektet `stroke`), och sedan ritas en figur (`rek`) med denna tjocklek. Den angivna tjockleken gäller tills den ändras på nytt (med metoden `setStroke`).

Ett ritprogram

En ritning kan definieras i en subklass till klassen `JPanel`. En panel av denna subklass kan skapas, och placeras i ett fönster. Fönstret visas sedan, och på så sätt visas panelen och ritningen i den. Så här kan detta gå till:

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

class RitPanel extends JPanel
{
    public void paintComponent (Graphics gr)
```


© FADIL GALJIĆ

```
    {
        super.paintComponent (gr);
        this.setBackground (Color.WHITE);
        Graphics2D    g = (Graphics2D) gr;

        Rectangle2D    rektangel1 =
                   new Rectangle2D.Double (90, 40, 120, 80);
        g.draw (rektangel1);

        Rectangle2D    rektangel2 =
                 new Rectangle2D.Double (240, 40, 120, 80);
        BasicStroke    stroke = new BasicStroke (2.0F);
        g.setStroke (stroke);
        g.draw (rektangel2);

        Rectangle2D    rektangel3 =
                 new Rectangle2D.Double (90, 150, 120, 80);
        g.setPaint (Color.LIGHT_GRAY);
        g.fill (rektangel3);

        Rectangle2D    rektangel4 =
                 new Rectangle2D.Double (240, 150, 120, 80);
        g.fill (rektangel4);
        g.setPaint (Color.BLACK);
        g.draw (rektangel4);
    }
}

class RitaFigurer
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame (" Rita figurer");
        frame.setSize (600, 400);
        frame.setLocation (100, 100);

        RitPanel    panel = new RitPanel ();
        frame.add (panel);

        frame.setViaible (true);
    }
}
```

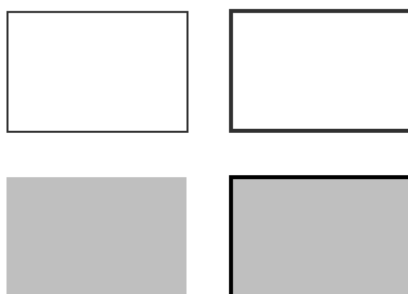
En panel (mer exakt: en typ av paneler) definieras i klassen `RitPanel`. Panelens bakgrundsfärg bestäms, och fyra rektanglar ritas i panelen. Den första rektangeln ritas med en penna vars tjocklek är `1` enhet (den förvalda tjockleken). Sedan ändras tjockleken (till `2` enheter) och en ny rektangel ritas. Därefter ritas en rektangel till, men nu fylls hela rektangelns area med färg (istället för metoden `draw` används metoden `fill`). Slutligen skapas och ritas en rektangel till. Denna rektangel målas med en färg

(med metoden `fill`), och rektangelns kontur ritas med en annan färg (med metoden `draw`).

En konkret panel (ett objekt av typen `RitPanel`) skapas i metoden `main`, i klassen `RitaFigurer`. Denna panel placeras i en ram (för att den ska kunna visas), och ramen visas. När ramen visas, visas även den panel som finns i ramen. Java skapar ett objekt av typen `Graphics2D`, och anropar metoden `paintComponent` i samband med panelen, och med objektet av typen `Graphics2D` som argument. Koden i metoden `paintComponent` utförs, och på så sätt bestäms panelens utseende (när panelen visas).

Programmets grafik definieras i klassen `RitPanel`. En panel av denna klass skapas och visas i klassen `RitaFigurer`. Om klassen `RitPanel` är tänkt för engångsanvändning (endast i samband med programmet `RitaFigurer`), kan de båda klasserna placeras i en och samma fil. Denna fil ska i så fall heta `RitaFigurer.java` (som den klass som innehåller metoden `main`). Om man har för avsikt att använda paneler av typen `RitPanel` i flera olika program, ska klasserna `RitPanel` och `RitaFigurer` placeras i två separata filer.

När programmet `RitaFigurer` exekveras, skapas en ram med en vit panel och fyra rektanglar i panelen. Panelen kan föreställas se ut så här:



Specificera en färg

Ett objekt av typen `Graphics2D` använder en viss färg. Den förvalda färgen är svart. Denna färg kan justeras med metoden `setPaint` (i klassen `Graphics2D`):

```
g.setPaint (Color.YELLOW);
```

Här sätts färgen till gul. Härefter ritas och målas allt med denna färg. Självklart kan färgen ändras på nytt.

© FADIL GALJIĆ

En färg anges med ett objekt av typen `java.awt.Color`. Man specificerar en färg genom att ange intensiteter av tre föreskrivna komponenter i färgen. Intensiteten för den röda, gröna och blå komponenten anges. En komponents intensitet anges som ett heltal mellan `0` och `255` (inklusive dessa värden). Man skapar en färg genom att kombinera tre angivna komponentfärger. Man kan göra så här:

```
Color    c = new Color (255, 100, 0);
g.setPaint (c);
```

Färgen skapas här mestadels från den röda färgen (det första argumentet), och delvis från den gröna färgen (det andra argumentet). Ett slags orange skapas. Sedan justeras den aktuella färgen i den grafiska kontexten till den skapade färgen.

Även ett fjärde argument kan anges när en färg skapas. Detta argument representerar en färgs genomskinlighet. Detta värde kallas för alfavärdet. Ett alfavärde anges med ett heltal mellan `0` (helt genomskinlig färg) och `255` (helt ogenomskinlig färg).

Metoderna `getRed`, `getGreeen` och `getBlue` ger en färgs komponenter. Metoden `getAlpha` ger en färgs alfavärde. I vissa sammanhang används även metoden `getRGB`, som returnerar en färgs komponenter som 4 byte inuti ett värde av typen `int`. Den högsta byten (bitarna 24-31) representerar alfavärdet, och tre följande byte representerar en färgs (röda, gröna och blå) komponenter.

Även flyttal (av typen `float`) kan användas för att specificera en färg. I så fall anges motsvarande komponenter och alfavärde som flyttal mellan `0.0` och `1.0` (inklusive dessa värden).

För att underlätta arbetet med färger, har flera vanliga färger definierats i klassen `Color`. Man kan välja en av dessa färger (ett fördefinierat objekt av typen `Color`) genom att välja en statisk konstant från klassen `Color`. Dessa konstanter är: `WHITE`, `LIGHT_GRAY`, `GRAY`, `DARK_GRAY`, `BLACK`, `BLUE`, `RED`, `YELLOW`, `GREEN`, `ORANGE`, `MAGENTA`, `CYAN` och `PINK`. Den vita färgen, till exempel, kan anges så här:

```
Color    farg = Color.WHITE;
```

Rita relativt panelen

En figurs position och dimensioner anges i antalet pixlar. Om dessa värden fixeras, kommer figuren att finnas på olika platser i olika paneler. En figur kan finnas i en panels hörn, eller mitt i panelen, beroende på pane-

© FADIL GALJIĆ

lens storlek. Figuren kan också vara liten eller stor, sett relativt denna panel. Det är därför bättre att uttrycka en figurs position och dimensioner relativt panelen, istället för att fixera dem. För att kunna göra detta, måste man bestämma dimensionerna för den panel (komponent) där man ritar. Dessa dimensioner erhålls med metoderna `getWidth` (returnerar den aktuella komponentens bredd) och `getHeight` (returnerar den aktuella komponentens höjd).

En cirkel kan ritas (i metoden `paintComponent`) relativt panelen så här:

```
int    w = this.getWidth ();
int    h = this.getHeight ();

int    xc = w / 2;
int    yc = h / 2;

int    d = (h < w)? h : w;
int    radie = 3 * d / 8;

Ellipse2D    cirkel = new Ellipse2D.Double ();
cirkel.setFrameFromCenter (xc, yc, xc - radie, yc - radie);

g.draw (cirkel);
```

Den aktuella panelens bredd (`w`) och höjd (`h`) bestäms. Sedan bestäms panelens mittpunkt (koordinaterna för denna punkt är `xc` och `yc`). Därefter ritas en cirkel i mitten av panelen. Cirkelns radie är `3/8` av panelens kortare dimension. Cirkeln kommer att ligga i mitten av panelen, och ha en konstant storlek relativt panelen.

Sammanfattning

Man kan rita en figur (ett objekt som implementerar gränssnittet `Shape`) i en grafisk komponent (ett objekt av någon subklass till klassen `JComponent`, till exempel i en panel). Detta gör man med ett objekt av typen `Graphics2D`. Ett sådant objekt erhålls automatiskt när metoden `paintComponent` omdefinieras. Ett objekt av typen `Graphics2D` är en slags penna, som har en viss tjocklek och färg. Tjockleken kan justeras med metoden `setStroke`, och färgen med metoden `setPaint`.

Koden som ritar olika figurer placeras i metoden `paintComponent`. Denna metod anropas automatiskt varje gång motsvarande grafiska komponent visas. Koden i metoden utförs, och motsvarande figurer ritas. I metoden `paintComponent` preciseras hur motsvarande grafiska komponent (till exempel en panel) kommer att se ut.

© FADIL GALJIĆ

En grafisk komponent har ett koordinatsystem, vars origo finns i komponentens övre vänstra hörn. Koordinatsystemets x-axel går åt höger, och dess y-axel går nedåt. En figurs position och dimensioner anges relativt detta koordinatsystem. För att få en figur med konstant position och storlek relativt den underliggande komponenten, uttrycker man figurens position och dimensioner via komponentens dimensioner.

Rita tecken

Rita en teckensträng

Man kan rita tecken i en panel, eller i en annan grafisk komponent (ett objekt av typen `javax.swing.JComponent`). Det går också att kombinera tecken och olika figurer i en och samma komponent. Metoden `paintComponent` omdefinieras, och tecken och figurer ritas i den metoden.

De tecken som ska ritas organiseras i olika teckensträngar. En teckensträng ritas med metoden `drawString` i klassen `java.awt.Graphics`. Den sträng som ska ritas anges som argument till metoden `drawString`. Även den position i komponenten där strängen ska ritas anges. Denna position preciseras med koordinaterna för den punkt där strängen ska börja (y-koordinaten för denna position bestämmer strängens baslinje – den linje på vilken strängen ligger). Med objektet `g` av typen `Graphics2D` (eller av typen `Graphics` – metoden `drawString` definieras i klassen `Graphics`) kan en teckensträng ritas så här:

```
String    s = "Morgonstund har guld i mun.";
g.drawString (s, 50, 20);
```

Strängen `s` ritas, med början i punkten `(50, 20)`.

Ett objekt av typen `Graphics` använder en viss font när det ritar en teckensträng. Om inte fonten specificeras, använder objektet en förvald font. En font kan anges med metoden `setFont` (i klassen `Graphics`):

```
Font    f = new Font ("SansSerif", Font.PLAIN, 16);
g.setFont (f);
```

En font representeras med ett objekt av typen `java.awt.Font`. Fontens namn, stil och storlek (i antalet punkter, som sedan avbildas till antalet pixlar) anges. En fonts stil anges med en konstant från klassen `Font`. Man kan välja mellan `PLAIN` (vanlig stil), `BOLD` (fet stil) och `ITALIC` (kursiv stil). Det går även att kombinera en vanlig eller fet stil med kursiv stil. Detta gör man så här:

© FADIL GALJIĆ

```
Font    f = new Font ("SansSerif", Font.BOLD + Font.ITALIC, 14);
```

Olika fonter kan väljas för olika teckensträngar. Man kan välja olika stilar och storlekar, och få teckensträngar med olika utseenden (se nedanstående bild).

Morgonstund har guld i mun.

Morgonstund har guld i mun.

Morgonstund har guld i mun.

Morgonstund har guld i mun.

Det finns även olika typer av fonter att välja mellan. Man anger en fonts typ genom att ange dess namn. En fonts logiska namn kan anges. Man kan välja mellan `SansSerif`, `Serif`, `Monospaced`, `Dialog` och `DialogInput`. Dessa namn avbildas till konkreta fonttyper, som finns i den aktuella datorn. Genom att välja olika typer av fonter, erhålls teckensträngar med olika utseenden (se nedanstående bild).

Morgonstund har guld i mun.

Morgonstund har guld i mun.

Morgonstund har guld i mun.

Morgonstund har guld i mun.

En teckensträngs dimensioner

Man kanske vill rita flera teckensträngar efter varandra, eller under varandra. Då måste man känna till platsen där en teckensträng avslutas och en annan teckensträng ska börja. Det som ska ritas kan ha följande form:

Morgonstund har **g u l d** i mun.

Tre skilda teckensträngar definieras för att skapa den här ritningen:

```
String    s1 = "Morgonstund har  ";
String    s2 = "g u l d  ";
String    s3 = "i mun.";
```


© FADIL GALJIĆ

För att den andra teckensträngen ska kunna ritas, måste positionen för slutet på den första teckensträngen beräknas. För att sedan den tredje teckensträngen ska kunna ritas, måste slutpositionen för den andra teckensträngen beräknas.

En teckensträngs bredd och höjd erhålls från teckensträngens omslutande rektangel. Teckensträngens bredd erhålls sedan som bredden av denna rektangel, och teckensträngens höjd som höjden av rektangeln. Dessa beräkningar kan utföras så här:

```
FontRenderContext    kontext = g.getFontRenderContext ();
Rectangle2D    rek1 = f1.getStringBounds (s1, kontext);
int    w1 = (int) rek1.getWidth ();
Rectangle2D    rek2 = f2.getStringBounds (s2, kontext);
int    w2 = (int) rek2.getWidth ();
```

Här skapas först ett objekt av typen `java.awt.font.FontRenderContext`, som innehåller vissa informationer som är nödvändiga för mätningar i samband med tecken (det har betydelse om ett tecken ritas på en skärm, eller skrivs ut på en skrivare). Detta objekt erhålls med metoden `getFontRenderContext` i klassen `Graphics2D`. När detta objekt bestämts, kan en teckensträngs omslutande rektangel (vid användningen av en viss font) erhållas. För detta ändamål används metoden `getStringBounds` i klassen `Font`.

Med bredden och höjden för de olika teckensträngarna går det att beräkna positionerna där teckensträngarna ska ritas:

```
g.setFont (f1);
g.drawString (s1, 50, 330);
g.setFont (f2);
g.drawString (s2, 50 + w1, 330);
g.setFont (f1);
g.drawString (s3, 50 + w1 + w2, 330);
```

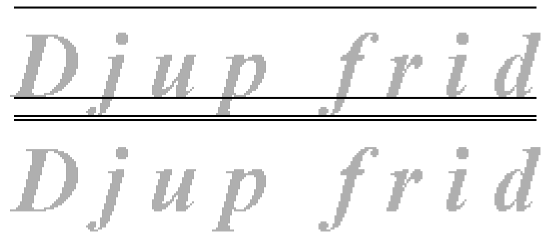
För att beräkna vertikala positioner kan man använda höjder av olika omslutande rektanglar. För en finare beräkning kan metoder i klassen `java.awt.font.LineMetrics` användas. Ett objekt av denna klass (för en given teckensträng och en given kontext) erhålls så här:

```
LineMetrics    metrik = f.getLineMetrics (s, kontext);
```

Objektets metoder `getAscent`, `getDescent`, `getLeading` och `getHeight` kan sedan användas för olika vertikala beräkningar. En rad text definieras med dess baslinje, dess övre linje och dess nedre linje (se bilden nedan). Metoden `getAscent` returnerar avståndet mellan en rads baslinje och dess övre linje. Metoden `getDescent` returnerar avståndet mellan en rads baslinje och dess nedre linje. Metoden `getLeading` returnerar avståndet mellan en

© FADIL GALJIĆ

rads nedre linje och nästa rads övre linje. Metoden `getHeight` returnerar avståndet mellan en rads övre linje och nästa rads övre linje.



Tolka en teckensträng som en figur

En teckensträng ritas med metoden `drawString` (i klassen `Graphics`), och inte med metoderna `draw` och `fill` (i klassen `Graphics2D`). Detta innebär att teckensträngar hanteras på ett sätt och andra figurer (`Shape`-s) på ett annat sätt. Man kan till exempel inte använda metoden `draw` för att rita ett teckens kontur och metoden `fill` för att fylla tecknets inre del med en given färg.

I vissa sammanhang vill man kunna hantera teckensträngar på samma sätt som alla andra figurer. För att kunna göra detta måste man skapa en figur (ett objekt av typen `Shape`) som motsvarar en given teckensträng. En teckensträng ska omvandlas till en figur. Denna omvandling kan utföras med ett objekt av klassen `java.awt.font.TextLayout`. Först skapas ett sådant objekt, och sedan erhålls motsvarande figur (`Shape`) med detta objekt.

Ett objekt av typen `TextLayout` representerar en teckensträng som ett grafiskt objekt. Hur ett sådant objekt ser ut beror på själva teckensträngen, och på den font som används när teckensträngen ritas. Objektets utseende beror även på den grafiska kontexten där objektet ritas. Ett och samma objekt ser olika ut på en skärm och i en pappersutskrift. Ett objekt av typen `TextLayout` kan skapas så här:

```
String    s = "L o v e";
Font    font = new Font ("Serif", Font.BOLD + Font.ITALIC, 60);
FontRenderContext    kontext = g.getFontRenderContext ();
TextLayout    layout = new TextLayout (s, font, kontext);
```

Klassen `TextLayout` har en metod som returnerar en figur (ett objekt av typen `Shape`) som motsvarar den givna strängen. Det är metoden `getOutline`, som tar emot ett objekt av typen `java.awt.geom.AffineTransform`

 © FADIL GALJIĆ

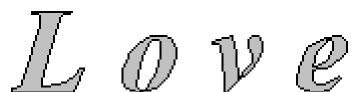
som argument. Detta argument preciserar den returnerade figurens position. Figuren kan fås så här:

```
AffineTransform    transform =
              AffineTransform.getTranslateInstance (80, 160);
Shape    figur = layout.getOutline (transform);
```

Här erhålls ett objekt av typen `Shape` som representerar den givna teckensträngen, och objektet placeras vid punkten (`80, 160`) (baslinjens vänstra ändpunkt). Detta objekt kan användas på samma sätt som alla andra figurer, till exempel så här:

```
g.setFont (font);
g.setStroke (new BasicStroke (1.0f));
g.setPaint (Color.LIGHT_GRAY);
g.fill (figur);
g.setPaint (Color.BLACK);
g.draw (figur);
```

Tecknens ytterlinjer ritas med svart färg, och tecknens inre delar fylls med grå färg (se bilden nedan). En teckensträng (omvandlad till en `Shape`) hanteras på samma sätt som alla andra figurer.



Rita figurer av olika typer

Ett objekt av typen `java.awt.Graphics2D` kan användas för att rita figurer av olika typer. Metoderna `draw` och `fill` i denna klass har en parameter av typen `java.awt.Shape`. Detta innebär att de kan ta emot objekt av alla klasser som implementerar gränssnittet `Shape`. Det är möjligt att rita linjer och sammansatta linjer, rektanglar, ellipser och bågar, samt areor och sammansatta figurer..

© FADIL GALJIĆ

Hantera bilder

Lagra en bild

Lagra och visa en bild

Man kan rita olika figurer (och/eller text) i en grafisk komponent (till exempel i en panel), genom att placera ritkoden i komponentens `paint-Component`-metod. Denna metod anropas automatiskt varje gång komponenten visas. Koden i metoden utförs, och olika figurer ritas i komponenten. En bild ritas i en grafisk komponent. Om det sker på detta sätt, lagras inte bilden som en oberoende enhet. Bilden är bunden till en viss komponent, och kan inte hanteras oberoende av denna komponent. Det går till exempel inte att spara bilden, eller använda den i samband med en annan komponent. För att kunna göra detta, måste bilden skapas som en självständig enhet i datorns minne. Bilden måste lagras.

En bild kan representeras via ett objekt av typen `java.awt.image.BufferedImage` (en subklass till klassen `java.awt.Image`). En bild kan skapas så här:

```
BufferedImage    bild =
      new BufferedImage (160, 100, BufferedImage.TYPE_INT_ARGB);
```

Här skapas en bild, vars storlek är `160 x 100` pixlar och vars typ är `TYPE_INT_ARGB`. En bilds typ definieras med en konstant från klassen `Buf-feredImage`. Man kan välja mellan `TYPE_INT_ARGB`, `TYPE_INT_RGB`, `TYPE_INT_BGR`, `TYPE_4BYTE_ABGR`, `TYPE_BYTE_GRAY`, och andra typer. Olika uppgifter kan användas för att representera en pixels färg, och dessa uppgifter kan lagras på olika sätt. Därför finns olika typer av bilder.

Man kan bestämma en bilds utseende genom att rita olika figurer (`Shape-s`) och/eller text. För detta krävs det en lämplig grafisk kontext (en penna). Denna kontext erhålls (som ett objekt av typen `Graphics2D`) med metoden `createGraphics` i klassen `BufferedImage`. Med metoderna `draw`, `fill` och `drawString` kan olika figurer och text ritas, på samma sätt som när man ritar i en panel. Med metoderna `setStroke`, `setPaint`, `setFont` och andra, kan man justera olika inställningar i den grafiska kontexten.

En bild kan ritas så här:

```
Graphics2D    gbi = bild.createGraphics ();
gbi.setStroke (new BasicStroke (2.0f));
gbi.setPaint (Color.BLACK);
```


© FADIL GALJIĆ

```
Rectangle2D    ram = new Rectangle2D.Double (0, 0, 160, 100);
gbi.setPaint (Color.WHITE);
gbi.fill (ram);
gbi.setColor (Color.BLACK);
gbi.draw(ram);

Ellipse2D    el = new Ellipse2D.Double (20, 10, 120, 80);
gbi.setPaint (Color.LIGHT_GRAY);
gbi.fill (el);

gbi.dispose ();
```

Här skapas en lämplig grafisk kontext (objektet `gbi`), och linjens tjocklek och färg anges. Först ritas en vit rektangulär area med en svart ram. Därefter ritas en grå ellips i mitten. Slutligen anropas metoden `dispose` (i klassen `Graphics`) för att frigöra de resurser som använts vid ritningen. De positioner och dimensioner som anges vid ritningen gäller relativt bilden. Bildens koordinatsystem används, och detta har origo i bildens vänstra övre hörn. Koordinatsystemets x-axel går till höger och dess y-axel går nedåt.

När man har skapat och ritat en bild, kan bilden användas på olika sätt. Den kan till exempel visas i en panel (eller i en annan grafisk komponent). Bilden kan skapas i den klass där panelen definieras (till exempel i metoden `paintComponent` i denna klass). Om bilden skapas utanför panelens definitionsklass, måste en referens som refererar till bilden tillföras till denna klass. Referensen ska tillföras antingen som argument till en konstruktor eller som argument till en metod.

En bild kan skapas och visas så här:

```
class RitPanel extends JPanel
{
    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        this.setBackground (Color.WHITE);
        Graphics2D    g = (Graphics2D) gr;

        BufferedImage    bild = new BufferedImage (
                           160, 100, BufferedImage.TYPE_INT_ARGB);

        // rita bilden här med dess grafiska kontext

        g.drawImage (bild, null, 60, 40);
    }
}
```

© FADIL GALJIĆ

En bild (av typen `BufferedImage`) ritas i en grafisk komponent med metoden `drawImage` i klassen `Graphics2D`. Bilden anges som första argument till metoden. Som andra argument anges ett eventuellt filter (ett objekt av typen `java.awt.image.BufferedImageOp`), som bearbetar bilden innan den ritas. Det tredje och fjärde argumentet till metoden `drawImage` anger den plats i den grafiska komponenten (panelen), där bilden ska visas (platsen för bildens vänstra övre hörn).

En bild kan även ritas med metoden `drawImage` i klassen `Graphics` (superklassen till klassen `Graphics2D`). Denna metod tar emot en bild (av typen `java.awt.Image`) och koordinaterna för bildens övre vänstra hörn (i den grafiska komponenten) som argument. Ytterligare ett argument, som representerar ett objekt (av typen `java.awt.image.ImageObserver`) som informeras om hur laddningen av en bild fortskrider, anges. Ett sådant objekt kan användas när bilden laddas via ett nätverk, till exempel i en webbläsare. Bilden kan i så fall ritas steg för steg. När en lokal bild visas, anges normalt `null` som sista argument till metoden `drawImage`:

```
g.drawImage (bild, 60,200, null)
```

En variant av metoden `drawImage` (i klassen `Graphics`) tar emot ett rektangulärt område via sina argument. Det är det område där bilden ska visas. Detta område preciseras via dess övre vänstra hörn, dess bredd och dess höjd. Metoden kan användas så här:

```
g.drawImage (bild, 320, 200, 120, 75, null);
```

Området som anges kan vara större eller mindre än den bild som visas. I så fall förstoras eller förminskas bilden proportionellt, så att den passar till det angivna området.

Man kan bestämma en delbild av en given bild med metoden `getSubimage` (i klassen `BufferedImage`). Det rektangulära området som omger delbilden anges som argument till metoden. Koordinaterna för områdets övre vänstra hörn, och områdets bredd och höjd anges:

```
BufferedImage    delBild = bild.getSubimage (0, 0, 80, 50);
```

En bild kan ritas inuti en annan bild (med metoden `drawImage`). En bild kan innehålla flera andra bilder som sina delar.

Definiera en typ av bilder

En klass som definierar en typ av bilder kan skapas. Man kan sedan skapa bilder av denna typ och använda dem på olika sätt.

 © FADIL GALJIĆ

En typ av bilder definieras i en subklass till klassen `BufferedImage`. I denna klass definieras ett antal konstruktorer och metoder, som beskriver hur en bild av klassern ser ut och beter sig. Genom olika parametrar till dessa konstruktorer och metoder kan en bilds olika egenskaper preciseras.

En typ av bilder kan definieras så här:

```
import java.awt.image.*;
import java.awt.*;
import java.awt.geom.*;

class FImage extends BufferedImage
{
    public FImage (int w, int h, Color ellipseColor)
    {
        super (w, h, BufferedImage.TYPE_INT_ARGB);
        Graphics2D    gbi = this.createGraphics ();
        gbi.setStroke (new BasicStroke (2.0f));

        Rectangle2D    ram = new Rectangle2D.Double (0, 0, w, h);
        gbi.setPaint (Color.WHITE);
        gbi.fill (ram);
        gbi.setColor (Color.BLACK);
        gbi.draw(ram);

        Ellipse2D    el = new Ellipse2D.Double ();
        el.setFrameFromDiagonal (w / 10, h / 10,
                                 9 * w / 10, 9 * h / 10);
        gbi.setPaint (ellipseColor);
        gbi.fill (el);

        gbi.dispose ();
    }
}
```

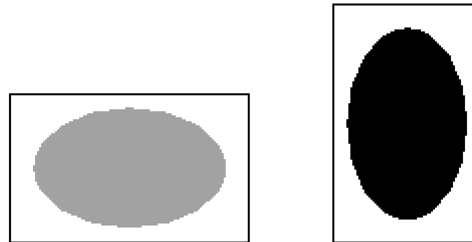
Först målas hela bilden i vitt, och en svart ram ritas. Därefter ritas en ellips i mitten. Ellipsens färg bestäms genom motsvarande argument till klassens konstruktor. Även bildens bredd (`w`) och höjd (`h`) bestäms genom motsvarande argument.

Två bilder av typen `FImage` kan skapas så här:

```
FImage    bild1 = new FImage (120, 75, Color.LIGHT_GRAY);
FImage    bild2 = new FImage (75, 120, Color.BLACK);
```

Dessa två bilder är av samma typ (`FImage`), men har olika dimensioner och färger (se nedanstående bild). Bilderna kan skapas i metoden `paintComponent` i en klass som representerar en grafisk komponent (till exempel en panel). Bilderna kan sedan ritas med metoden `drawImage`.

© FADIL GALJIĆ



På samma sätt kan man definiera mer avancerade bilder, och använda dem i olika sammanhang. Ett helt bibliotek kan skapas, med olika typer av bilder (på samma sätt som Javas standardbibliotek definierar olika typer av figurer).

Spara en bild

En bild av typen `BufferedImage` kan sparas i en fil. Bilden kan sedan läsas in och användas på olika sätt. En bild kan läsas in oavsett om den är skapad i ett Javaprogram eller på något annat sätt, till exempel i ett ritprogram eller med en kamera.

En bild sparas med metoden `write` i klassen `javax.imageio.ImageIO`. Detta är en statisk metod, som tar emot den bild som ska sparas, destinationsfilens format (som en teckensträng) och destinationsfilen (som ett objekt av typen `java.io.File`) som argument. En bild kan sparas så här:

```
File    fil = new File ("bild.png");
ImageIO.write (bild, "PNG", fil);
```

Bilden `bild` (av någon klass som implementerar gränssnittet `java.awt.image.RenderedImage`, till exempel av klassen `BufferedImage`) sparas i filen `bild.png`. Filens format är `PNG`. Även formatet `JPEG` kan användas. Om filen redan innehåller en bild, skrivs denna bild över (den gamla bilden tas bort). Om något problem uppstår under utmatningen, kastar metoden `write` ett undantag av typen `java.io.IOException`.

En bild kan läsas in från en fil med metoden `read` i klassen `javax.imageio.ImageIO`. Det är en statisk metod som tar källfilen (som ett objekt av typen `java.io.File`) som argument. Metoden returnerar den inlästa bilden som ett objekt av typen `BufferedImage`. En bild kan läsas in så här:

```
File    fil = new File ("bild.png");
```

 © FADIL GALJIĆ

```
BufferedImage    bild = ImageIO.read (fil);
```

En bild vars format är `PNG` läses in från en fil. Metoden `read` kan även läsa `JPEG`- och `GIF`-filer. Om något problem uppstår vid inläsningen, kastar metoden `read` ett undantag av typen `java.io.IOException`.

Definiera en bilds pixlar

Man kan definiera en bild genom att definiera bildens pixlar. Man kan ange färg för en godtycklig pixel i bilden. Genom att specificera färger för en bilds pixlar, definieras bildens utseende.

Bildens pixlar kontrolleras via bildens *raster*. Varje pixel i detta raster har sina koordinater. På så sätt är det möjligt att komma åt en bestämd pixel och ange dess färg. En bilds raster representeras med ett objekt av typen `java.awt.image.WritableRaster`. Objektets metod `setPixel` används för att ange en pixels färg. En pixels koordinater och färg tillförs som argument till metoden.

Man kan få en bilds raster, och sätta färg på en pixel i bilden, på följande sätt:

```
BufferedImage    bild = new BufferedImage (160, 100,
                                      BufferedI-
mage.TYPE_INT_ARGB);
WritableRaster    raster = bild.getRaster ();
raster.setPixel (0, 0, Color.BLACK);
```

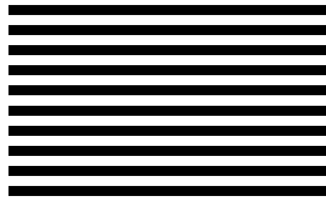
Genom att stega igenom alla bildens pixlar och ange deras färger, kan man bestämma bildens utseende.

En klass som definierar en typ av bilder genom att definiera pixlar i en bild av denna typ, kan skapas så här:

```
class PixelBild extends BufferedImage
{
    public PixelBild (int w, int h)
    {
        super (w, h, BufferedImage.TYPE_INT_ARGB);

        WritableRaster    raster = this.getRaster ();

        int[]    svart = {0, 0, 0, 255};
        int[]    vit = {255, 255, 255, 255};
        int[]    farg = vit;

        for (int i = 0; i < w; i++)
        {
```

```
                farg = vit;
                for (int j = 0; j < h; j++)
                {
                    if (j % 5 == 0)
                        if (farg == vit)
                            farg = svart;
                        else
                            farg = vit;

                    raster.setPixel (i, j, farg);
                }
            }
        }
}
```

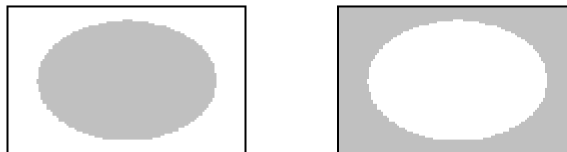
Endast två färger används, svart och vit. En pixels position (koordinaterna i bildens raster) anges via heltalen `i` och `j`. Dessa heltal varieras, och på så sätt stegar man sig igenom bildens raster. När en vertikal linje ritas (`j` ökas för ett bestämt värde på `i`), ändras färger efter varje `5` pixlar. På så sätt skapas horisontella linjer som är `5` pixlar tjocka (se bilden nedan).



Bearbeta en bild

Bearbeta en bild av typen TYPE_INT_ARGB

Det går att komma åt en bilds pixlar, och bearbeta dessa pixlar på olika sätt. En pixels aktuella färg kan erhållas via bildens raster. Man kan analysera pixelns position och färg, och bestämma sig för att eventuellt ändra denna färg. Det går också att bearbeta pixlar i ett visst område inom en bild. Nedanstående figurer visar en bild före och efter bearbetningen.



 © FADIL GALJIĆ

Man påbörjar bearbetningen av bilden genom att bestämma dess dimensioner (bredd och höjd) och raster. Med en bild (av typen `BufferedImage`) som refereras av referensen `bild`, erhålls dessa uppgifter så här:

```
int    w = bild.getWidth ();
int    h = bild.getHeight ();
WritableRaster    raster = bild.getRaster ();
```

I en bild av typen `TYPE_INT_ARGB` representeras varje pixel med fyra värden. Komponenterna röd, grön, blå och alfa i en pixel specificeras. Varje komponent anges med ett heltal mellan `0` och `255`. Dessa komponenter erhålls med metoden `getPixel` i klassen `WritableRaster`. Man kan göra så här:

```
int[]    pixel = new int[4];
raster.getPixel (x, y, pixel);
Color   farg = new Color (pixel[0], pixel[1], pixel[2], pixel[3]);
```

Först skapas en vektor, för att en pixels komponenter ska kunna lagras. Dessa komponenter (för pixeln på positionen (`x`, `y`)) erhålls med metoden `getPixel`. Denna metod lagrar komponenterna i vektorn `pixel`. Sedan skapas motsvarande färg (som ett objekt av typen `Color`) utifrån dessa komponenter.

Man kan analysera en pixels färg och/eller position, och utifrån dessa uppgifter bestämma sig för att eventuellt ändra pixelns färg. När man väl har bestämt den nya färgen (som ett objekt av typen `Color`), kan denna anges med metoden `setPixel`. Om den önskade färgen heter `farg`, kan detta göras så här:

```
pixel[0] = farg.getRed ();
pixel[1] = farg.getGreen ();
pixel[2] = farg.getBlue ();
pixel[3] = farg.getAlpha ();
raster.setPixel (x, y, pixel);
```

De nya värdena lagras i vektorn `pixel`, och sedan används denna vektor för att sätta den nya färgen.

En hel bild kan bearbetas, eller en del av en bild. Följande modell kan användas:

```
int[]    pixel = new int[4];
Color    farg = null;
for (int x = 0; x < w; x++)
    for (int y = 0; y < h; y++)
    {
        raster.getPixel (x, y, pixel);
        farg = new Color (pixel[0], pixel[1], pixel[2], pixel[3]);
```

© FADIL GALJIĆ

```
        // analysera här pixelns position och/eller färg,
        // och ändra eventuellt färgen

        pixel[0] = farg.getRed ();
        pixel[1] = farg.getGreen ();
        pixel[2] = farg.getBlue ();
        pixel[3] = farg.getAlpha ();

        raster.setPixel (x, y, pixel);
    }
```

Man kan få uppgifter om samtliga pixlar i en bild, och lagra dessa uppgifter i en tillräckligt stor vektor. Detta görs med metoden `getPixels` (i klassen `WritableRaster`). Man kan sedan modifiera vissa värden i vektorn, och på det sättet bearbeta bilden. Nya värden anges med metoden `setPixels` (i klassen `WritableRaster`). Följande modell kan användas:

```
WritableRaster    raster = bild.getRaster ();
int[]    pixels = new int[4 * w * h];
raster.getPixels (0, 0, w, h, pixels);

Color   farg = null;
for (int i = 0; i < pixels.length - 4; i = i + 4)
{
    farg = new Color (
        pixels[i], pixels[i + 1], pixels[i + 2], pixels[i + 3]);

    // ändra eventuellt färgen här

    pixels1[i] = farg.getRed ();
    pixels1[i + 1] = farg.getGreen ();
    pixels1[i + 2] = farg.getBlue ();
    pixels1[i + 3] = farg.getAlpha ();
}
raster.setPixels (0, 0, w, h, pixels);
```

I vektorn `pixels` lagras uppgifter om en bilds pixlar. Uppgifter om en pixel lagras genom att pixelns röd-, grön-, blå- och alfakomponent lagras. Därför hämtas fyra successiva värden från vektorn och motsvarande färg skapas. Pixelns färg och/eller position analyseras, och färgen ändras eventuellt. Motsvarande värden i vektorn `pixels` uppdateras. Sedan hämtas nästa fyra värden (för nästa pixel) och så vidare. Nya värden för bilden anges med metoden `setPixels` (värden för samtliga pixlar i bilden anges med ett enda metodanrop).

Man kan bearbeta en given bild, eller skapa en ny bild utifrån en given bild. Man kan skapa en ny bild och erhålla bildens raster, på följande vis:

```
BufferedImage    bild1 = new BufferedImage (bild.getWidth (),
                              bild.getHeight (), bild.getType ());
```

 © FADIL GALJIĆ

```
WritableRaster    raster1 = bild1.getRaster ();
```

Bilden `bild1` skapas med utgångspunkt i bilden `bild`. Samma storlek och typ väljs för den nya bilden. Därefter ska en pixels färg avläsas från den första bilden (dess raster), färgen ska bearbetas, och sedan ska den nya färgen anges för motsvarande pixel i den nya bilden (dess raster). Detta upprepas för samtliga pixlar i bilden.

Metoderna `getPixels` och `setPixels` tar emot ett rektangulärt område via sina argument (de första fyra argumenten). Det övre vänstra hörnet av området anges, samt dess bredd och höjd. Pixlarnas färg i detta område avläses, eller anges. Detta innebär att ett område i en bild kan kopieras och placeras på en plats i en annan bild.

Bearbeta en bild av en godtycklig typ

Bilder av olika typer representerar en pixels färg på olika sätt. En bild av typen `TYPE_INT_ARGB` representerar färgen hos en pixel med fyra komponenter (alfa, röd, grön och blå). Dessa komponenter packas i ett heltalsvärde av typen `int`. Varje komponent representeras med 8 bitar i detta heltalsvärde. En bild av en annan typ representerar en pixels färg på ett annat sätt. Därför går det inte att få uppgifter om en pixel för en bild av en godtycklig typ enligt det mönster som normalt appliceras i samband med en bild av typen `TYPE_INT_ARGB`. Om bilden inte är av typen `TYPE_INT_ARGB`, måste en något mer komplicerad strategi användas.

Bilder av en viss typ använder en viss *färgmodell* för att representera uppgifter om en pixels färg. En sådan modell kan representeras med ett objekt av typen `java.awt.image.ColorModel`. En bilds färgmodell erhålls med metoden `getColorModel` (i klassen `BufferedImage`). Denna modell kan sedan användas för omvandlingar mellan den aktuella färgrepresentationen och `ARGB`-representationen (alfa-röd-grön-blå-representationen). Med en bild (av typen `BufferedImage`) som refereras av referensen `bild`, kan man göra så här:

```
ColorModel    model = bild.getColorModel ();
WritableRaster    raster = bild.getRaster ();
Object    pixel = raster.getDataElements (x, y, null);
int    argb = model.getRGB (pixel);
Color    farg = new Color (argb, true);
```

Uppgifter om en pixels färg erhålls med metoden `getDataElements` i klassen `WritableRaster`. Sedan används färgmodellen `model` för att översätta dessa uppgifter till ett heltalsvärde (`argb`) av typen `int`. Detta heltalsvärde innehåller uppgifter om den aktuella färgens alfa-, röd-, grön- och blå-

© FADIL GALJIĆ

komponent (8 bitar för varje komponent). Utifrån detta heltalsvärde skapas motsvarande färg som ett objekt av typen `Color`.

Man kan analysera en pixels färg och/eller position och bestämma sig för att ändra denna färg. Efter den ändringen måste den nya färgen representeras enligt den färgmodell som den aktuella bilden använder. För detta ändamål används metoden `getDataElements` i klassen `ColorModel`. Färgen anges sedan med metoden `setDataElements` i klassen `WritableRaster`. Så här gör man:

```
argb = farg.getRGB ();
pixel = model.getDataElements (argb, null);
raster.setDataElements (x, y, pixel);
```

Man går från en färgmodellspecifik representation till motsvarande `ARGB`-representation, och sedan till motsvarande `Color`-representation. Efter ändringen går man i motsatt riktning. Metoden `getDataElements` ger uppgifter om en pixel, och metoden `setDataElements` anger en pixels färg (på samma sätt som metoderna `getPixel` och `setPixel` används i samband med en bild av typen `TYPE_INT_ARGB`).

Skriva ut en bild

En bild kan skrivas ut i ett Javaprogram. På så vis kan man få en papperskopia av en bild. För att en bild ska kunna skrivas ut, måste den definieras i metoden `print` i en klass som implementerar gränssnittet `java.awt.print.Printable`. En sådan klass kan skapas så här:

```
class PrintableImage implements Printable
{
    public int print (Graphics gr, PageFormat pf, int page)
                                    throws PrinterException
    {
        if (page >= 1)
            return Printable.NO_SUCH_PAGE;

        Graphics2D    g = (Graphics2D) gr;

        Ellipse2D    e =
                  new Ellipse2D.Double (50, 100, 120, 80);
        g.draw (e);

        return Printable.PAGE_EXISTS;
    }
}
```

 © FADIL GALJIĆ

Klassen `PrintableImage` (ett godtyckligt namn kan väljas för denna klass) implementerar gränssnittet `Printable`, vilket innebär att klassen har metoden `print`. Metoden `print` tar ett argument av typen `Graphics`, som representerar den grafiska kontext som bilden ritas i. Metoden tar också ett objekt av typen `java.awt.print.PageFormat`. Detta argument representerar formatet (storleken, riktningen och marginalerna) för den sida där bilden ritas. Metoden `print` tar även ett heltal som argument. Detta heltal representerar index (ordningsnummer) för den sida där bilden ritas. Första sidan har index `0`. Metoden `print` kan kasta ett (kontrollerat) undantag av typen `java.awt.print.PrinterException`.

Metoden `print` returnerar ett heltal av typen `int`. Detta heltal representeras med en lämplig konstant från gränssnittet `Printable`. Något av värdena `PAGE_EXISTS` eller `NO_SUCH_PAGE` kan returneras. Om konstanten `PAGE_EXISTS` returneras, innebär detta att metoden `print` måste anropas (minst) en gång till. Det finns ytterligare sidor att skriva ut. Om konstanten `NO_SUCH_PAGE` returneras, innebär detta att metoden `print` inte längre ska anropas. Det avsedda antalet sidor har skrivits ut. Utifrån det returnerade värdet avgörs det om nästa sida ska skrivas ut eller inte. Metoden `print` ritar olika figurer (och/eller text) å den ena sidan, och signalerar å andra sidan när (vid vilket index) utskriften ska upphöra. Om endast en sida ska skrivas ut (index `0`), placeras följande kod i början av metoden `print`:

```
if (page >= 1)
    return Printable.NO_SUCH_PAGE;
```

För att kunna skriva ut en bild behövs det ett objekt som kan beskriva en sida (i skrivartermer), och skicka denna beskrivning till en skrivare. Detta kan göras med ett objekt av typen `java.awt.print.PrinterJob`. Ett sådant objekt erhålls med (den statiska) metoden `getPrinterJob` i klassen `PrinterJob`. Så här kan man göra:

```
PrinterJob    skrivare = PrinterJob.getPrinterJob ();
```

Ett objekt av typen `PrinterJob` behöver få informationer av två olika typer. Å ena sidan behöver det veta vad det ska skriva ut. Å den andra sidan behöver det känna till olika utskriftsinställningar. Utskriftsinnehållet preciseras med ett objekt av typen `Printable`. Objektets `print`-metod innehåller ritningskoden, och algoritmen som bestämmer det antal sidor som ska skrivas ut. Ett sådant objekt tillförs till ett objekt av typen `PrinterJob` med metoden `setPrintable`, till exempel så här:

```
Printable    bild = new PrintableImage ();
skrivare.setPrintable (bild);
```

När något ska skrivas ut, måste man kunna justera olika utskriftsinställningar. Detta görs normalt via en dialogruta, där det går att välja skrivare, utskriftsformat och annat. En sådan ruta aktiveras med metoden printDialog i klassen PrinterJob. Denna metod returnerar true om användaren väljer OK (Ja, skriv ut), annars false. Metoden tar emot ett objekt som implementerar gränssnittet javax.print.attribute.PrintRequestAttributeSet (till exempel ett objekt av klassen javax.print.attribute.HashPrintRequestAttributeSet) som argument. De olika inställningarna i dialogrutan tolkas och lagras i detta objekt.

Ett objekt som kan lagra olika utskriftsinställninar kan skapas, och en dialogruta (för att fylla objektet med olika informationer) kan aktiveras, så här:

```
PrintRequestAttributeSet    attribut =
                         new HashPrintRequestAttributeSet ();
boolean    okAttSkrivaUt = skrivare.printDialog (attribut);
```

Med metoderna setPrintable och printDialog förser sig ett objekt av typen PrinterJob med de informationer som det behöver för att kunna skriva ut en eller flera sidor. Efter anropet till dessa metoder vet objektet vad det ska skriva ut, och hur utskriften ska ske.

Ett objekt av typen PrinterJob skriver ut en eller flera sidor med metoden print (i klassen PrinterJob). Metoden kan aktiveras så här:

```
if (okAttSkrivaUt)
    skrivare.print (attribut);
```

Metoden print i klassen PrinterJob anropar metoden print i samband med det aktuella Printable-objektet, och på så sätt skapas sidor. Denna metod kan kasta ett kontrollerat undantag av typen java.awt.print.PrinterException.

Metoden print i klassen PrinterJob vet inte hur många sidor den behöver skriva ut. Därför anropar den metoden print i samband med det aktuella Printable-objektet. Den slutar anropa denna metod när metoden returnerar NO_SUCH_PAGE. Metoden print som definieras i en klass som implementerar gränssnittet Printable, analyserar aktuella sidindex och avgör (genom att returnera en lämplig konstant) om utskriftsjobbet ska avslutas eller inte. I samband med vissa skrivare anropas metoden print (i samband med det aktuella Printable-objektet) flera gånger för varje sida. En sida skrivs ut stegvis. Men i varje sådant anrop skickas ett och samma index (sidans ordningsnummerav). Detta index används för att avgöra när ett utskriftsjobb ska avslutas.

© FADIL GALJIĆ

I det aktuella `Printable`-objektets `print`-metod preciseras positioner och dimensioner för olika figurer och tecken. När en bild ritas med en skrivares grafiska kontext, tolkas angivna dimensioner och positioner som antalet punkter. Även ett pappersarks dimensioner mäts i punkter. En punkt är `1/72` av en inch (1 inch = 2,54 cm). Ett A4-ark har dimensionerna `595 x 842` punkter.

Ett papperssidas koordinatsystem har origo i det övre vänstra hörnet. Dess x-axel går åt höger och dess y-axel går nedåt. Trots att origo ligger i det övre vänstra hörnet, går det inte att rita utanför papperssidans marginaler. Det är därför förnuftigt att flytta origo till det övre vänstra hörnet av den area där man kan rita. Detta kan göras i metoden `print` i den klass som implementerar gränssnittet `Printable`, på följande vis:

```
double    x = pf.getImageableX ();
double    y = pf.getImageableY ();
g.translate (x, y);
```

Objektet `pf` representerar det aktuella utskriftsformatet (ett objekt av typen `PageFormat` som är tillgängligt i metoden `print` – ett objekt av typen `PrinterJob` får denna information via motsvarande dialogruta, och använder den när den anropar metoden `print` i samband med ett `Printable`-objekt). Detta objekt har metoderna `getImageableX` och `getImageableY`, som returnerar koordinaterna (relativt hela sidan) för det övre vänstra hörnet av den area som man kan rita i. Origo flyttas till den punkten med metoden `translate` i klassen `Graphics2D`.

Med metoderna `getImageWidth` och `getImageHeight` (i klassen `PageFormat`) kan man även få dimensionerna för den del av ett ark där man kan rita. Dessa uppgifter kan sedan användas för att positionera och dimensionera figurer och tecken. Detta görs i metoden `print` i den klass som implementerar gränssnittet `Printable`.

© FADIL GALJIĆ

Olika rittekniker

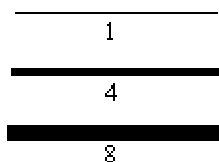
Olika typer av linjer

En linjes tjocklek

När man ritar en figur, kan man välja en linjes tjocklek. En linjes tjocklek anges genom ett objekt av typen `java.awt.BasicStroke`. Detta objekt tillförs till den aktuella grafiska kontexten (ett objekt av typen `Graphics2D`) med metoden `setStroke`. Kontexten används sedan för att rita olika figurer. Om den grafiska kontexten refereras med referensen `g`, kan man göra så här:

```
BasicStroke    stroke = new BasicStroke (4.0F);
g.setStroke (stroke);

Line2D    linje = new Line2D.Double (100, 80, 200, 80);
g.draw (linje);
```

Här väljs tjockleken `4` enheter, och ett linjesegment med denna tjocklek ritas. Tjockleken kan varieras, vilket ger olika tjocka linjer (se bilden nedan).



Dekorera en linjes ändpunkter

Man kan välja olika dekorationer för ändpunkterna på en öppen linje. Olika dekorationer kan även appliceras på de punkter där två segment i en öppen linje möts. Det går också att rita streckade linjer med olika streckmönster. Alla dessa egenskaper definieras genom ett objekt av typen `BasicStroke`.

Man väljer den dekoration som ska användas på en öppen linjes ändpunkter genom att välja en lämplig konstant i klassen `BasicStroke`. En av följande konstanter kan väljas: `CAP_BUTT` (ingen dekoration), `CAP_ROUND` (halvcirkel) och `CAP_SQUARE` (halvkvadrat). Klassen `BasicStroke` innehåller inte någon konstruktor eller metod som gör det möjligt att endast

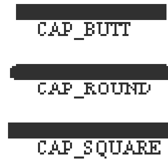
© FADIL GALJIĆ

ange dekorationsstilen för en linjes ändpunkter. Även linjens tjocklek måste anges, och den stil som ska användas för att dekorera de punkter där två segment möts. Man kan välja den dekoration som ska appliceras på en punkt där två segment i en öppen linje möts, genom att välja en av följande konstanter: `JOIN_MITER` (en spetsig dekoration), `JOIN_ROUND` (en rund dekoration) och `JOIN_BEVEL` (en rak dekoration).

Ett linjesegment kan ritas så här:

```
Line2D    linje = new Line2D.Double (100, 200, 200, 200);
BasicStroke    stroke = new BasicStroke (10.0F,
                                        BasicStroke.CAP_BUTT,
                                        BasicStroke.JOIN_MITER);
g.setStroke (stroke);
g.draw (linje);
```

Tjockleken sätts här till `10` enheter. Linjens ändpunkter ska inte dekoreras på något vis (`CAP_BUTT`), och de punkter där två segment möts ska dekoreras med spetsiga dekorationer (`JOIN_MITER`). Genom att variera dekorationsstilen för en linjes ändpunkter, kan man skapa linjer med olika utseenden (se bilden nedan).



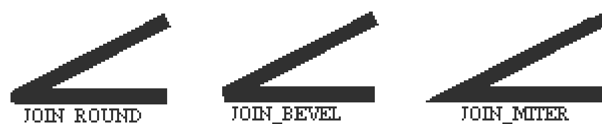
Dekorera de punkter där två segment möts

En öppen linje, som består av flera bundna segment, kan skapas. Det går då att välja den dekorationsstil som ska appliceras på de punkter där två segment möts. Så här kan detta göras:

```
GeneralPath    path = new GeneralPath ();
path.moveTo (350, 250);
path.lineTo (250, 300);
path.lineTo (350, 300);

BasicStroke    stroke = new BasicStroke (10.0F,
                                         BasicStroke.CAP_BUTT,
                                         BasicStroke.JOIN_MITER);
g.setStroke (stroke);
g.draw (path);
```

© FADIL GALJIĆ

Den stil som används för att dekorera de punkter där två segment möts kan varieras, och på så vis kan linjer med olika utseenden skapas (se bilden nedan).



Streckade linjer

För att rita streckade linjer, definierar man först ett streckmönster. Ett sådant mönster definieras med en flyttalsvektor (av typen `float[]`), där ett strecks längd och längden på mellanrummet mellan strecken anges. Ett streckmönster kan definieras så här:

```
float[]    monster = {10, 10};
```

Den här vektorn definierar ett streckmönster, där varje streck är `10` enheter långt och varje mellanrum också är `10` enheter långt. En vektor som definierar ett streckmönster kan ha fler än två element. Om fyra element anges i en vektor, representerar det första elementet det första streckets längd, det andra elementet det första mellanrummets längd, det tredje elementet det andra streckets längd, och det fjärde elementet anger det andra mellanrummets längd. Detta mönster upprepas sedan längs en linje.

När man ritar en streckad linje, definierar man även en fas som preciserar förskjutningen (åt vänster) av det första strecket relativt en linjes början. Den förskjutna delen av strecket ritas inte.

En streckad linje kan ritas så här:

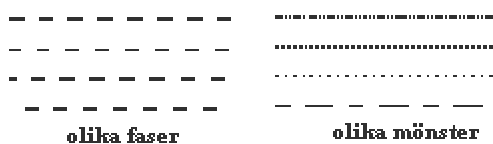
```
float[]    monster = {10, 10};
BasicStroke    stroke = new BasicStroke (2.0F,
                                BasicStroke.CAP_BUTT,
                                BasicStroke.JOIN_MITER, 10.0F,
                                monster, 0);

Line2D    linje = new Line2D.Double (400, 50, 550, 50);
g.setStroke (stroke);
g.draw (linje);
```

Klassen `BasicStroke` innehåller inte någon konstruktor eller metod som gör det möjligt att endast ange ett streckmönster. Även en linjes tjocklek måste anges, och dekorationsstilarna för olika ändpunkter och de punkter

 © FADIL GALJIĆ

där två segment möts. De dekorationsstilar som anges, appliceras på varje enskilt streck. Förutom dekorationsstilarna anges (som fjärde argument till konstruktorn) även ett flyttal, som bestämmer den gräns där `MITER`-stilen automatiskt ersätts av `BEVEL`-stilen. När två segment bildar en alltför liten vinkel, är det inte lämpligt att använda en spetsig dekorationsstil. Istället använder man då en rak stil (`BEVEL`), även om en spetsig stil (`MITER`) har valts i konstruktorn.

Som femte argument till konstruktorn anges ett streckmönster, och som sjätte argument anges en fas (förskjutningen mellan det första strecket och linjens början). Genom att variera streckmönstret och förskjutningen, kan man skapa streckade linjer med olika utseenden (se bilden nedan).

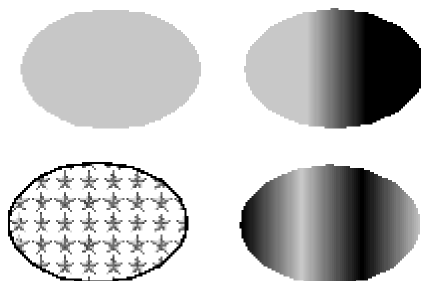


Om inte något objekt av typen `BasicStroke` anges för den grafiska kontexten, använder kontexten följande (förvalda) objekt:

```
BasicStroke    stroke = new BasicStroke (1.0F,
                                BasicStroke.CAP_SQUARE,
                                BasicStroke.JOIN_MITER, 10.0F);
```

Olika fyllnadsmönster

Olika fyllnadsmönster kan användas för att fylla en figur (se bilden nedan). En figur kan fyllas med en enhetlig färg, eller med en färggradering. Man kan också fylla en figur med en bild som mönster. Även ritade tecken kan fyllas på olika sätt.



© FADIL GALJIĆ

Det fyllnadsmönster som ska användas när olika figurer och text ritas, bestäms med metoden `setPaint` i klassen `Graphics2D`. Metoden `setPaint` tar emot ett objekt som implementerar gränssnittet `java.awt.Paint` som argument. Ett sådant objekt representerar ett fyllnadsmönster, och det är detta mönster som ska användas för att fylla olika figurer och tecken.

Klassen `java.awt.Color` implementerar gränssnittet `Paint`, och ett objekt av denna klass kan användas som argument till metoden `setPaint`. Detta innebär att det som ritas (en figur eller tecken i en teckensträng) fylls med en enhetlig färg. På följande sätt kan man skapa en ellips, och fylla denna med en given färg:

```
Ellipse2D    ellips = new Ellipse2D.Double (100, 50, 120, 80);
Color    c = Color.LIGHT_GRAY;
g.setPaint (c);  // g - ett objekt av typen Graphics2D
g.fill (ellips);
```

Den angivna färgen gäller oavsett om metoden `draw` eller metoden `fill` används. Om man vill rita ellipsens kontur med en annan färg, måste man byta färgen i den grafiska kontexten. Detta kan göras så här:

```
g.setPaint (Color.BLACK);
g.draw (ellips);
```

Även klassen `java.awt.GradientPaint` implementerar gränssnittet `Paint`. Ett objekt av denna typ kan skapas och användas så här:

```
GradientPaint    gp = new GradientPaint (
                                140, 90, Color.WHITE,
                                180, 90, Color.LIGHT_GRAY);
g.setPaint (gp);
g.fill (ellips);
```

Här preciseras två punkter i ellipsen (dessa punkter kan även anges som objekt av typen `Point2D`), och färgerna för dessa två punkter. På så sätt definieras en färggradering från den första punkten till den andra punkten. Den första färgen övergår så småningom i den andra färgen, på vägen från den första punkten till den andra punkten. En vertikal linje i figuren har en enhetlig färg. Området till vänster om den första punkten fylls med den första färgen, och området till höger om den andra punkten fylls med den andra färgen. För att färggraderingen ska appliceras även på dessa två områden (bortom punkterna), måste ytterligare ett argument anges när motsvarande objekt av typen `GradientPaint` skapas. Det sista argumentet måste vara `true`:

```
GradientPaint    gp = new GradientPaint (
                                140, 90, Color.WHITE,
                                180, 90, Color.LIGHT_GRAY,
                                true);
```


© FADIL GALJIĆ

Förutom klasserna `Color` och `GradientPaint`, finns det en klass till i standardbiblioteket som implementerar gränssnittet `Paint`. Det är klassen `java.awt.TexturePaint`. Ett objekt av denna typ skapas utifrån en bild (av typen `BufferedImage`) och en rektangel (av typen `Rectangle2D`). Den angivna rektangeln definierar ett område som fylls med den angivna bilden (bildens storlek anpassas till detta område). På så sätt skapas en fyllnadsenhet. Denna enhet upprepas horisontellt och vertikalt inuti en angiven figur (eller inuti en strängs tecken), och på så vis fås ett fyllnadsmönster.

Ett objekt av typen `TexturePaint` kan skapas och användas så här:

```
BufferedImage    bild = new BufferedImage (10, 10,
                                   BufferedImage.TYPE_INT_ARGB);
Graphics2D    gb = bild.createGraphics ();
Ellipse2D     cirkel = new Ellipse2D.Double (0, 0, 8, 8);
gb.setPaint (Color.LIGHT_GRAY);
gb.fill (cirkel);

Rectangle2D    ankare = new Rectangle2D.Double (0, 0,
               0.9 * bild.getWidth (), 0.9 * bild.getHeight ());

TexturePaint    tp = new TexturePaint (bild, ankare);
g.setPaint (tp);
g.fill (ellips);
```

Först skapas en bild som representerar en grå cirkel. Därefter skapas rektangeln som ska innehålla den skapade bilden, och som ska upprepas som en fyllnadsenhet inuti ellipsen. Utifrån denna bild och rektangeln skapas ett objekt av typen `TexturePaint`. Detta objekt representerar ett fyllnadsmönster, och detta fyllnadsmönster används för att fylla ellipsen. En ellips med gråa cirklar skapas.

Rita inuti en figur

Arean som man ritar i kan begränsas till en viss figur (en `Shape`). Allt som ritas därefter begränsas till denna figur. Det som ritas utanför figuren syns inte. Detta kan användas för att fylla en figur med ett visst mönster.

Den area som ritandet utförs i är vanligtvis en hel panel (eller en annan grafisk komponent), eller en papperssida utan marginaler. Denna area erhålls med metoden `getClip` i klassen `Graphics`. När man har memorerat arean (för att kunna återvända till den, om så behövs), kan man med metoden `clip` i klassen `Graphics2D` begränsa ritningsområdet till en given figur. Denna metod tar emot en figur (en `Shape`), och begränsar ritnings-

© FADIL GALJIĆ

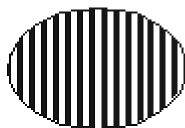
området till den area som är gemensam för det aktuella ritningsområdet och figuren. Man kan göra så här:

```
Ellipse2D    ellips = new Ellipse2D.Double (100, 50, 120, 80);
g.draw (ellips);
Shape    ritArea = g.getClip ();
g.clip (ellips);
```

En ellips ritas, och därefter begränsas ritningsområdet till denna ellips. Bara det som hamnar inuti ellipsen kommer att synas. Så här kan man göra:

```
g.setStroke (new BasicStroke (4));
int    w = 100;
while (w <= 220)
{
    g.draw (new Line2D.Double (w, 50, w, 130));
    w = w + 8;
}
```

Linjer ritas inuti ellipsens omslutande rektangel. Men eftersom ritningsområdet begränsats till ellipsen, syns bara de delar av linjerna som hamnar inuti ellipsen. Det skapas en ellips med vertikala linjer (se bilden nedan).



När man ritat färdigt i ellipsen, kan man återgå till den ursprungliga ritningsarean (till exempel hela panelen). Detta görs med metoden `setClip` i klassen `Graphics`:

```
g.setClip (ritArea);
```

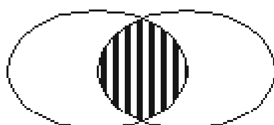
Metoden `setClip` fastställer ritningsarean till en given figur (en `Shape`). Metoden `clip` kombinerar den figur som anges som argument med den aktuella ritningsarean. Deras gemensamma del blir det nya ritningsområdet. Metoden `clip` kan appliceras flera gånger, och på så vis kan den önskade ritningsarean erhållas. Det går att göra så här:

```
Ellipse2D    ellips1 = new Ellipse2D.Double (100, 60, 120, 80);
g.setStroke (new BasicStroke (1));
g.draw (ellips1);

Ellipse2D    ellips2 = new Ellipse2D.Double (160, 60, 120, 80);
g.draw (ellips2);
```


© FADIL GALJIĆ

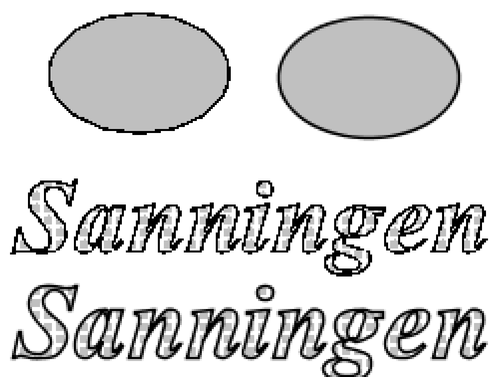
```
g.clip (ellips1);
g.clip (ellips2);

g.setStroke (new BasicStroke (4));
int    w = 100;
while (w <= 280)
{
    g.draw (new Line2D.Double (w, 60, w, 140));
    w = w + 8;
}
```

Ritningsområdet begränsas till den area som är gemensam för två givna ellipser. Därefter ritas vertikala linjer i ett rektangulärt område som omfattar dessa ellipser. Det är dock endast de delar av dessa linjer som hamnar i ellipsernas snitt som blir synliga (se bilden nedan).



Förbättra en bilds kvalitet

En bilds kvalitet kan förbättras, på bekostnad av rithastigheten (se bilden nedan). Det går även att göra tvärtom: rithastigheten kan ökas på bekostnad av bildens kvalitet. Man gör detta genom att justera vissa inställningar i den grafiska kontexten. Dessa justeringar utförs med hjälp av klassen `java.awt.RenderingsHints`.



För att precisera den egenskap som ska justeras, anger man en nyckel som representerar denna egenskap. En nyckel anges med en lämplig konstant

© FADIL GALJIĆ

från klassen RenderingHints. Man kan välja mellan konstanterna KEY_ANTIALIASING, KEY_TEXT_ANTIALIASING, KEY_COLOR_RENDERING och andra. När en nyckel valts, väljer man nyckelns värde. Ett värde anges med en lämplig konstant från klassen RenderingHints. Om till exempel KEY_ANTIALIASING valts som nyckel, kan ett av tre möjliga värden väljas för denna nyckel. Dessa värden är VALUE_ANTIALIAS_ON (förbättrar kvaliteten på bekostnad av rithastigheten), VALUE_ANTIALIAS_OFF (ökar rithastigheten på bekostnad av bildens kvalitet) och VALUE_ANTIALIAS_DEFAULT (förvalt värde).

En nyckel och dess värde anges med metoden setRenderingHint i klassen Graphics2D. Dessa uppgifter tillförs som argument till metoden. Om g är den aktuella grafiska kontexten, kan metoden användas så här:

```
g.setRenderingHint (RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

På det här sättet går det att förbättra kvaliteten för en ellips eller en båge, och för en kvadratisk eller kubisk kurva. En texts kvalitet kan förbättras på ett liknande sätt. Man väljer nyckeln KEY_TEXT_ANTIALIASING, och anger VALUE_TEXT_ANTIALIASING som dess värde. Även de andra nycklarnas värden kan justeras, och på så vis kan en bilds kvalitet (eller rithastigheten) ökas. Dessa förbättringar kan vara mer eller mindre synliga, beroende på vad som justeras och på den dator som programmet exekveras i.

Ett objekt av typen RenderingHints representerar en hashtabell, eftersom det implementerar gränssnittet java.util.Map. Därför kan ett sådant objekt användas som behållare för flera nyckel-värde-par. Ett sådant par placeras i tabellen med metoden put, och en nyckels värde erhålls med metoden get. Med metoden setRenderingHints (s på slutet, värden för alla nycklar i tabellen anges) kan inställningarna i en grafisk kontext justeras utifrån ett sådant objekt. En tom hashtabell av typen RenderingHints skapas genom att null tillförs som argument till dess konstruktor.

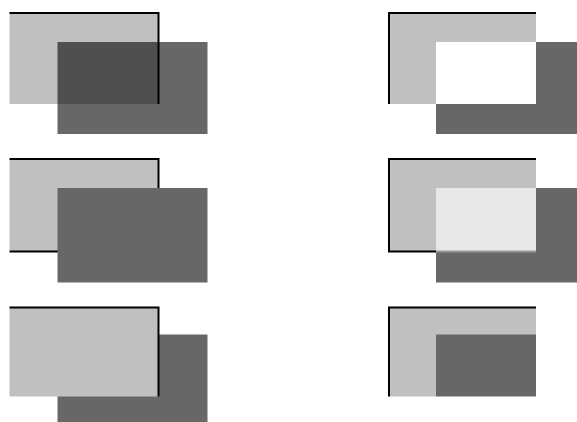
Komposition av två bilder

En bild kan ritas över en annan bild, eller över en del av en annan bild. På så sätt bildas en *komposition* av två bilder (som omfattar de båda bilderna). Den bild som ritas först kan kallas för *destinationsbilden*, och den bild som ritas över kan kallas för *källbilden*.

När en komposition av två bilder ritas, kan olika strategier, så kallade *regler*, användas. Man väljer en av dessa kompositionsregler genom att välja en konstant från klassen java.awt.AlphaComposite. Följande kon-

 © FADIL GALJIĆ

stanter (kompositionsregler) definieras i denna klass: `SRC`, `SRC_OVER`, `SRC_IN`, `SRC_OUT`, `SRC_ATOP`, `DST`, `DST_OVER`, `DST_IN`, `DST_OUT`, `DST_ATOP`, `XOR`, `CLEAR` (`SRC` står för source (källbilden) och `DST` för destination (destinationsbilden)). Bilder med olika utseenden kan skapas genom att olika regler appliceras (se bilden nedan). Om till exempel regeln `SRC` väljs, placeras källbilden över destinationsbilden. Den del av destinationsbilden som ligger under källbilden syns inte. För att göra även denna del synlig, ska regeln `SRC_OVER` appliceras. I så fall blir källbilden mer eller mindre genomskinlig. För att placera destinationsbilden överst (så att motsvarande del av källbilden inte syns), ska regeln `DST_OVER` användas.



För att en kompositionsstrategi ska vara fullständigt definierad, måste även motsvarande genomskinlighetsgrad för källbilden anges. Den faktorn anges med motsvarande alfavärde för källbildens pixlar. Detta värde anges som ett flyttal (av typen `float`) mellan `0.0` (helt genomskinlig) och `1.0` (helt ogenomskinlig).

En kompositionsregel och ett alfavärde kombineras inuti ett objekt av typen `java.awt.AlphaComposite`. Ett sådant objekt erhålls med (den statiska) metoden `getInstance` i klassen `AlphaComposite`:

```
AlphaComposite    composite =
                    AlphaComposite.getInstance (SRC_OVER, 0.6f);
```

Ett objekt av typen `AlphaComposite` beskriver ett sätt att kombinera två bilder. För att applicera detta sätt, måste motsvarande inställning i den aktuella grafiska kontexten justeras. Detta görs med metoden `setComposite` i klassen `Graphics2D`. Om `g` är den aktuella grafiska kontexten, gör man så här:

```
g.setComposite (composite);
```

© FADIL GALJIĆ

Man kan ha flera objekt av typen `AlphaComposite`, och använda olika objekt vid olika tillfällen. Den förvalda regeln i klassen `Graphics2D` är `SRC_OVER`, och det förvalda alfavärdet är `1.0` (källbilden helt ogenomskinlig).

Det finns inte några garantier för att alla regler kan appliceras korrekt vid ritning på en skärm (olika skärmar kan hantera alfavärdet på olika sätt). Därför skapar man först en komposition av två bilder som en bild av typen `BufferedImage`. Bilderna kombineras i datorns minne. Därefter ritas kompositionsbilden på skärmen. En kompositionsbild kan definieras enligt följande mönster:

```
class CompositeImage extends BufferedImage
{
    public CompositeImage ()
    {
        super (600, 400, BufferedImage.TYPE_INT_ARGB);
        Graphics2D    g = this.createGraphics ();

        // rita destinationsbilden här

        // ange kompositionsinställningarna
        AlphaComposite    composite =
                    AlphaComposite.getInstance (SRC_OVER, 0.6f);
        g.setComposite (composite);

        // rita källbilden här
    }
}
```

En bild av klassen `CompositeImage` kan skapas (till exempel i metoden `paintComponent`, när en grafisk komponent definieras), och ritas med metoden `drawImage` (precis som alla andra bilder av typen `BufferedImage`).

© FADIL GALJIĆ

Rörliga figurer

Flytta en figur

En panel (eller en annan grafisk komponent) kan användas som kanvas när man ritar olika figurer. En klass som definierar en typ av paneler skapas, och i denna klass omdefinieras metoden `paintComponent`. Ritkoden placeras i denna metod. Koden utförs i samband med en panel av klassen varje gång panelen visas. Panelen ritar sig själv med sin metod `paintComponent`. Den är både en kanvas och ett ritverktyg.

En figur som ritas i en panel kan definieras med ett antal variabler. Dessa variabler kan uppdateras efter ett pass genom metoden `paintComponent`. På så sätt förbereds en ny situation för nästa pass genom metoden. Det skapas en möjlighet att ändra en panels utseende under programmets gång.

En typ av paneler kan definieras så här:

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

class Rum extends JPanel
{
    private Shape    figure;
    private int    x = 0;
    private int    y = 100;

    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        figur = new Ellipse2D.Double (x, y, 12, 8);
        g.fill (figur);

        x += 100;
    }
}
```

En figur (en ellips) ritas i metoden `paintComponent`. Figurens position i panelen representeras med variablerna `x` och `y`. Först ritas figuren i punkten (`0, 100)` (det vänstra övre hörnet av ellipsens omslutande rektangel hamnar där). Därefter ändras variabeln `x`. En ny situation förbereds i pa-

© FADIL GALJIĆ

nelen. När panelen visas nästa gång, kommer den att se annorlunda ut. Figuren kommer att finnas på en annan plats.

Metoden `paintComponent` anropas inte direkt i våra program (den anropas automatiskt när en grafisk komponent visas). Men metoden kan anropas indirekt, via metoden `repaint` (i klassen `java.awt.Component`, som är panelens indirekta superklass). Med ett sådant anrop tvingas panelen att rita om sig. Om omständigheterna i panelen ändrats vid första passet genom metoden `paintComponent`, kommer panelen att se annorlunda ut efter omritningen.

Man kan rita om en panel, och på så sätt flytta figuren i panelen, på följande vis:

```
import java.awt.*;
import javax.swing.*;

class FlyttaEnFigur
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame (" Flytta en figur");
        frame.setSize (600, 400);
        frame.setLocation (100, 100);
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        Rum    rum = new Rum ();
        frame.add (rum);

        frame.setVisible (true);


        for (int i = 0; i < 10; i++)
        {
            try
            {
                Thread.sleep (2000);
            }
            catch (InterruptedException e)
            {}

            rum.repaint ();
        }
    }
}
```

Ett rum (en panel) av typen `Rum` skapas och visas. Därefter anropas metoden `repaint` i samband med detta rum, och på så sätt framtvingas en omritning av rummet. Rummet ritas om flera gånger, med en tids för-

 © FADIL GALJIĆ

dröjning. Rummet ritas om genom att dess `paintComponent`-metod utförs. Metoden ritar en figur, och bestämmer figurens position för nästa omritning. Figuren flyttas ett antal enheter åt höger. Som resultat av flera successiva anrop till metoden `repaint`, skapas en figur som hoppar i rummet. Figuren startar vid rummets vänstra kant och lämnar rummet vid dess högra kant.

Varje anrop till metoden `repaint` är en begäran att panelen ska ritas om. Om flera successiva anrop utförs, kan det hända att ett anrop inträffar innan det föregående anropet bearbetats. I så fall resulterar dessa två anrop i en enda omritning. Man kan därför inte räkna med att man har så många anrop till metoden `paintComponent` som till metoden `repaint`. Det går inte att dra några slutsatser om en figurs position i panelen genom att räkna antalet anrop till metoden `repaint`.

En figur som rör sig

En figur kan flyttas till en ny position i en panel. Detta kan användas för att skapa en figur som rör sig. En och samma figur ska ritas på flera successiva, närliggande positioner, med en kort tids fördröjning. På så sätt skapas intrycket av en figur som rör sig.

En figurs position kan uppdateras i metoden `paintComponent`, eller i någon annan metod i panelens definitionsklass. Uppdateringskoden kan organiseras i en särskild metod, som bestämmer en figurs bana. Denna metod kan sedan anropas före varje anrop till metoden `repaint`. Uppdateringsmetoden vet en figurs aktuella position, och kan avgöra när figuren behöver avsluta sin rörelse. Det kan till exempel vara när figuren lämnar panelen. I så fall måste metoden meddela detta, så att man vet att anropen till denna metod och metoden `repaint` måste avslutas. Metoden kan meddela om en figur nått sin destination eller inte, genom att returnera ett booleskt värde. Man kan välja att returnera `false` när figuren når sin destination.

En klass med en särskild uppdateringsmetod kan skapas så här:

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

class Rum extends JPanel
{
    private Shape    figur;
    private int    x;
```

© FADIL GALJIĆ

```
    private int    y;
    private int    dx;
    private int    dy;

    public Rum ()
    {
        this.setBackground (Color.PINK);

        this.x = -12;
        this.y = 200;

        dx = 1;
        dy = 1;
    }


    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        figur = new Ellipse2D.Double (x, y, 12, 8);
        g.setPaint (Color.BLUE);
        g.fill (figur);
    }

    public boolean move ()
    {
        x += dx;
        y += dy;

        int    w = this.getWidth ();
        int    h = this.getHeight ();

        if (y == h - 8  ||  y == 0)
            dy = -dy;

        boolean    inutiRummet = (x <= w)? true : false;

        return inutiRummet;
    }
}
```

En figur som representerar en ellips ritas. Figurens position bestäms med koordinaterna `x` och `y`. Figurens steg är `dx` i x-riktningen och `dy` i y-riktningen. Rummets (panelens) dimensioner är `w` och `h`.

Figurens position uppdateras i metoden `move`. Denna metod ökar ständigt figurens `x`-position, och tvingar på det sättet figuren att röra sig mot rummets högra kant. Metoden ökar figurens `y`-position tills figuren når

 © FADIL GALJIĆ

(med sin nedre kant – därför `h - 8` (`8` är ellipsens höjd)) rummets nedre kant. Då ändras riktningen genom att ett negativt steg väljs för `y`-riktningen. Figurens riktning ändras även vid rummets övre kant. Metoden `move` analyserar figurens `x`-position, och avgör om figuren fortfarande befinner sig i rummet eller inte. Om figuren lämnat rummet (vid rummets högra kant), returnerar metoden `false`.

Man kan skapa en klass, och anropa metoderna `move` och `repaint` i metoden `main` i denna klass. Det går också att skapa en särskild klass som kontrollerar omritningen av ett rum. Detta kan göras så här:

```
class Mover
{
    private Rum    rum;

    public Mover (Rum rum)
    {
        this.rum = rum;
    }

    public void move ()
    {
        boolean    inteNattDestination = rum.move ();
        while (inteNattDestination)
        {
            rum.repaint ();

            try
            {
                Thread.sleep (4);
            }
            catch (InterruptedException e)
            {}

            inteNattDestination = rum.move ();
        }
    }
}
```

Ett objekt av typen `Mover` har tillgång till ett rum (av typen `Rum`), och anropar rummets metoder `move` och `repaint`. På så sätt rör sig motsvarande figur i rummet. Dessa metoder anropas så länge figuren befinner sig i rummet. Ett objekt av typen `Mover` har en viss anropsfrekvens, som bestäms genom argument till metoden `sleep`. Genom att ändra denna frekvens, kan man ändra figurens hastighet.

Man kan skapa ett rum, och ett objekt av typen `Mover` utifrån detta rum. Detta kan göras i metoden `main` i en särskild klass. Sedan kan figurens

© FADIL GALJIĆ

rörelse i rummet startas genom att metoden `move` anropas i samband med `Mover`-objektet.

Definiera en rörlig figur

Man kan skapa en klass som definierar en typ av paneler, och rita en figur i klassens `paintComponent`-metod. Figurens position kan antingen uppdateras i denna metod, eller i någon annan metod i samma klass. Denna strategi kan dock bara användas i vissa enkla fall. Det kan finnas behov av flera figurer som rör sig i en och samma panel. I detta fall måste positionerna för var och en av figurerna uppdateras. Ju fler figurerna är, desto svårare blir det att kontrollera situationen. Det behövs därför en bättre strategi.

En panels funktion är att rita figurer, inte att uppdatera deras positioner. Uppdateringen av en figurs position gäller själva figuren, inte panelen. Att ändra sin position är en egenskap hos en figur. Därför ska uppdateringsmetoden placeras i den klass där en figurtyp definieras.

Man kan definiera en typ av figurer som kan visa (rita) sig i ett rum (ett objekt av typen `JComponent`, till exempel en panel), och som kan röra sig i rummet. Det behövs en metod som ritar en figur av denna typ, och en metod som ändrar figurens position. Dessa metoder kan kallas för `draw` (ritmetoden) och `move` (förflyttningsmetoden). Metoderna kan definieras i två separata gränssnitt. Då kan figurer av olika typer implementera dessa gränssnitt. På så sätt skapas möjlighet att rita och flytta figurer av olika typer på ett enhetligt sätt. Gränssnitten kan heta `Drawable` och `Movable`, och definieras (i separata filer) på följande vis:

```
// Drawable.java

import java.awt.*;

public interface Drawable
{
    void draw (Graphics2D g);
}


// Movable.java

public interface Movable
{
    boolean move ();
}
```


© FADIL GALJIĆ

Metoden `draw` får en grafisk kontext, och ritar en figur (av en klass som implementerar gränssnittet `Drawable`) med denna kontext. Metoden `move` flyttar en figur (av en klass som implementerar gränssnittet `Movable`) till en ny position i rummet, och returnerar ett booleskt värde. Detta värde signalerar om figuren nått sin destination eller inte. När figuren når sin destination returnerar metoden `false` (annars `true`).

Man kan skapa en klass som definierar en typ av figurer, och som implementerar gränssnitten `Drawable` och `Moveable`. En figur av denna typ kan visa (rita) sig i ett givet rum, och kan flytta sig i detta rum. Så här kan detta göras:

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

class Partikel implements Drawable, Movable
{
    private int    w;
    private int    h;

    private Paint    fyllnadsmonster;
    private int    xsize;
    private int    ysize;

    private int    x;
    private int    y;

    private int    dx;
    private int    dy;


    public Partikel (JComponent rum, Paint fyllnadsmonster)
    {
        this.w = rum.getWidth ();
        this.h = rum.getHeight ();

        this.fyllnadsmonster = fyllnadsmonster;
        this.xsize = 12;
        this.ysize = 8;

        this.x = -xsize;
        this.y = (int) (9 * h * Math.random () / 10) + 1;

        this.dx = 1;
        this.dy = 1;
    }

    public void setSize (int xsize, int ysize)
```

```
    {
        this.xsize = xsize;
        this.ysize = ysize;
    }

    public void draw (Graphics2D g)
    {
        Ellipse2D    figur = new Ellipse2D.Double (
                                      x, y, xsize, ysize);
        g.setPaint (fyllnadsmonster);
        g.fill (figur);
    }

    public boolean move ()
    {
        x += dx;
        y += dy;

        if (y == h - ysize  ||  y == 0)
            dy = -dy;

        boolean    inutiRummet = (x <= w)? true : false;

        return inutiRummet;
    }
}
```

Här definieras en partikel (en figur med en elliptisk form), som kan visa sig och röra sig i ett givet rum. Rummets dimensioner är `w` och `h`. Dessa dimensioner fås utifrån (via ett argument till konstruktorn) när en partikel binds till ett visst rum (en partikel kan bara röra sig i ett rum). Partikelns dimensioner är `xsize` och `ysize`. Partikelns position i rummet beskrivs med variablerna `x` och `y` (koordinaterna för det övre vänstra hörnet av ellipsens omslutande rektangel). Dessa variabler initieras i klassens konstruktor (partikelns startposition är vid rummets vänstra kant). Partikelns steg i `x`-riktningen är `dx`, och partikelns steg i `y`-riktningen är `dy`. En partikel kan fyllas med olika fyllnadsmönster (ett fyllnadsmönster tillförs som argument till klassens konstruktor). Metoden `setSize` bestämmer partikelns storlek. Metoden `draw` ritar en partikel på en viss position i rummet. Metoden `move` flyttar partikeln till en ny position i rummet. När partikeln lämnar rummet, returnerar denna metod `false`.

På följande vis kan man definiera en typ av paneler, som kan användas som rum för en partikel:

```
import java.awt.*;
import javax.swing.*;

class Rum extends JPanel
```


© FADIL GALJIĆ

```
{
    private Drawable    figur;

    public Rum ()
    {
        this.setBackground (Color.PINK);
        figur = null;
    }

    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        if (figur != null)
            figur.draw (g);
    }

    public void setDrawable (Drawable figur)
    {
        this.figur = figur;
    }
}
```

En panel av typen `Rum` har en figur, och det är denna figur som ritas varje gång panelen ritas. En figur placeras i ett rum med metoden `setDrawable`. Denna metod kan ta emot vilken figur som helst som kan ritas (med metoden `draw` som är definierad i gränssnittet `Drawable`). Den kan till exempel ta emot en partikel (ett objekt av typen `Partikel`).

Man kan skapa en partikel (ett objekt av typen `Partikel`) och ett rum (ett objekt av typen `Rum`), och låta partikeln röra sig i rummet. Partikeln flyttas med metoden `move`, och rummet uppdateras med metoden `repaint`. En särskild klass, som definierar uppgiften, kan skapas. Så här kan man göra:

```
class Mover
{
    private Rum    rum;

    public Mover (Rum rum)
    {
        this.rum = rum;
    }

    public void move (Movable figur)
    {
        boolean    inteNattDestination = figur.move ();
        while (inteNattDestination)
        {
            rum.repaint ();
```

```
            try
            {
                Thread.sleep (4);
            }
            catch (InterruptedException e)
            {}

            inteNattDestination = figur.move ();
        }
    }
}
```

En applikation med ett rum och en partikel som rör sig i rummet kan skapas på följande vis:

```
import java.awt.*;
import javax.swing.*;

class EnRorligFigur
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame (" En rörlig figur");
        frame.setSize (600, 400);
        frame.setLocation (100, 100);
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        Rum    rum = new Rum ();
        frame.add (rum);

        frame.setVisible (true);

        Partikel    partikel = new Partikel (rum, Color.BLUE);
        rum.setDrawable (partikel);
        Mover    mover = new Mover (rum);
        mover.move (partikel);
    }
}
```

Här skapas en partikel som rör sig i ett rum, från rummets vänstra kant mot dess högra kant. Partikeln lämnar rummet vid rummets högra kant.

Flera rörliga figurer

Det kan finnas flera figurer som rör sig i samma rum. Dessa figurer kan vara av samma typ, eller av olika typer. Var och en av dessa figurer måste kunna ritas (det räcker med att de är `Drawable`), och måste kunna röra sig

© FADIL GALJIĆ

(det räcker med att de är `Movable`). En figur kan komma in i ett rum när som helst, och lämna rummet när som helst.

Ett rum för flera figurer kan definieras så här:

```
import java.util.*;
import java.awt.*;
import javax.swing.*;

class Rum extends JPanel
{
    private ArrayList<Drawable>    figurer;

    public Rum ()
    {
        this.setBackground (Color.PINK);
        figurer = new ArrayList<Drawable> ();
    }

    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        synchronized (figurer)
        {
            int    antalFigurer = figurer.size ();
            Drawable    figur = null;
            for (int i = 0; i < antalFigurer; i++)
            {
                figur = figurer.get (i);
                figur.draw (g);
            }
        }
    }

    public synchronized void addDrawable (Drawable figur)
    {
        figurer.add (figur);
    }

    public synchronized void removeDrawable (Drawable figur)
    {
        figurer.remove (figur);
    }
}
```

Ett rum är en panel, som har ett antal figurer (av typen `Drawable`) som kan ritas. Dessa figurer lagras i en behållare av typen `java.util.ArrayList`. Varje gång panelen ritas, ritas alla figurerna. En

© FADIL GALJIĆ

figur placeras i rummet med metoden `addDrawable`, och en figur tas bort från rummet med metoden `removeDrawable`. Figurerna ritas i ett synkroniserat block. Man ska inte lägga till eller ta bort en figur medan figurerna ritas. Metoderna `addDrawable` och `removeDrawable` är synkroniserade, och därför kan det inte inträffa. Inte heller kan det inträffa att en figur läggs till samtidigt som en annan figur tas bort.

För att flera oberoende figurer ska kunna röra sig och ritas samtidigt, behövs det flera trådar. En tråd ska hantera en figur. En tråd kan startas när som helst, och den kan avslutas när som helst. En sådan tråd kan definieras så här:

```
class Mover<T extends Drawable & Movable> implements Runnable
{
    private Rum    rum;
    private T    figur;

    public Mover (Rum rum, T figur)
    {
        this.rum = rum;
        this.figur = figur;
    }

    public void move ()
    {
        boolean    inteNattDestination = figur.move ();
        while (inteNattDestination)
        {
            rum.repaint ();

            try
            {
                Thread.sleep (4);
            }
            catch (InterruptedException e)
            {}

            inteNattDestination = figur.move ();
        }

        rum.removeDrawable (figur);
    }

    public void run ()
    {
        this.move ();
    }
}
```

 © FADIL GALJIĆ

Ett objekt av typen `Mover` har ett rum och en figur (som är `Drawable` och `Movable` – `Drawable` för att kunna tas bort från rummet med metoden `removeDrawable`, och `Movable` för att kunna anropa metoden `move` i samband med den). Figuren rör sig i detta rum. När figuren når sin destination, tas den bort ifrån rummet och avslutar sin aktivitet. Klassen `Mover` implementerar gränssnittet `Runnable`, vilket innebär att koden i `run`-metoden kan exekveras parallellt med exekveringen av andra kodsekvenser.

I metoden `main` i en applikation kan ett rum skapas, och olika figurer som rör sig i rummet. Detta kan göras så här:

```
import java.awt.*;
import javax.swing.*;

class RorligaFigurer
{
    public static void main (String[] args)
    {
        JFrame    frame = new JFrame (" Rörliga figurer");
        frame.setSize (600, 400);
        frame.setLocation (100, 100);
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        Rum    rum = new Rum ();
        frame.add (rum);

        frame.setVisible (true);

        while (true)
        {
            Partikel    partikel = new Partikel (rum, Color.BLUE);
            rum.addDrawable (partikel);

            Mover<Partikel>    mover =
                              new Mover<Partikel> (rum, partikel);
            new Thread (mover).start ();

            try
            {
                Thread.sleep (2000);
            }
            catch (InterruptedException e)
            {}
        }
    }
}
```

© FADIL GALJIĆ

Här skapas partiklar som rör sig i rummet. Partiklarna kommer in i rummet med regelbundna tidsfördröjningar. En partikel rör sig enligt den algoritm som byggts in i klassen `Partikel` (en partikel vet på vilken plats den startar, den känner till sin bana och vet när den avslutar sin rörelse i rummet).

© FADIL GALJIĆ

Koordinatbyte

Komponentkoordinater och användarkoordinater

När man ritar en figur i en grafisk komponent, anger man på något sätt figurens position i komponenten, och figurens storlek. När till exempel en rektangel ritas, anges koordinaterna för rektangelns övre vänstra hörn och rektangelns bredd och höjd. En rektangel kan definieras så här:

```
Rectangle2D    rekt = new Rectangle2D.Double (100, 50, 120, 80);
```

Här anges att rektangelns vänstra övre hörn finns i punkten (`100, 50`), och att rektangelns dimensioner är `120` och `80`. Man använder ett eget koordinatsystem, och anger alla uppgifter i egna enheter. Dessa enheter kan vara pixlar, millimeter, centimeter, meter och så vidare.

En grafisk komponent (som det ritas i) har ett eget koordinatsystem. Detta koordinatsystem har origo i komponentens övre vänstra hörn, dess x-axel är en horisontell linje riktad åt höger, och dess y-axel är en vertikal linje riktad nedåt. Alla uppgifter i koordinatsystemet anges i antal pixlar.

När en figur ritas i en komponent, måste de egna koordinaterna relateras till de koordinater som gäller för komponenten. Det egna koordinatsystemet måste relateras till komponentens koordinatsystem. En *koordinattransformation* måste definieras.

Om man inte definierar en koordinattransformation, använder den aktuella grafiska kontexten en *standardtransformation*. Denna utgår i så fall ifrån att användarkoordinatsystemet sammanfaller med det koordinatsystem som används i den aktuella komponenten. Alla uppgifter som anges tolkas som om de var angivna i antalet pixlar. Skillnaden mellan komponentens koordinatsystem och användarkoordinatsystemet försummas. Man definierar sina figurer direkt i komponentkoordinater.

Man kan definiera en egen koordinattransformation, och använda denna för att översätta sina koordinater (angivna i egna enheter) till komponentkoordinater (angivna i pixlar). En sådan transformation definierar var i en komponent som man placerar sitt koordinatsystem, hur koordinatsystemets axlar orienteras, och hur användarenheter (till exempel centimeter) översätts till pixlar.

Den grafiska kontexten som används för att rita i en komponent (ett objekt av typen `java.awt.Graphics2D`) har ett objekt av typen

`java.awt.geom.AffineTransform`, som representerar den aktuella koordinattransformationen. Det är denna transformation som används för att översätta användarkoordinater till komponentkoordinater vid ritningen. *Transformationsobjektet* kan påverkas, och därmed kan en figurs position, dimensioner, orientering och utseende påverkas. Ett transformationsobjekt har en uppsättning *transformationsparametrar* som beskriver en koordinattransformation. Alla, eller bara vissa av dessa parametrar, kan påverkas. När transformationsobjektet väl har ändrats, används de nya inställningarna i objektet vid efterföljande koordinatbyten. De används så länge objektet inte modifieras på nytt.

Ett objekt av typen `AffineTransform` representerar en linjär transformation. Denna transformation lämnar raka linjer raka och parallella linjer parallella. Den gällande transformationen (det aktuella transformationsobjektet) för en grafisk kontext erhålls med metoden `getTransform`. Om `g` är den aktuella grafiska kontexten, kan man göra så här:

```
AffineTransform    at0 = g.getTransform ();
```

På detta vis sparas den aktuella uppsättningen transformationsparametrar i transformationsobjektet. Objektet kan sedan modifieras med metoden `transform` i klassen `Graphics2D`. Denna metod tar ett objekt av typen `AffineTransform` som argument, och bildar en ny transformation utifrån den aktuella transformationen och objektet. En *komposition av transformationer* bildas, och därefter används den nya transformationen. Om man vid något tillfälle vill gå tillbaka till den gamla transformationen, kan man göra detta med metoden `setTransform`, på följande vis:

```
g.setTransform (at0);
```

Metoden `transform` kombinerar det aktuella transformationsobjektet i den grafiska kontexten med ett givet transformationsobjekt (argumentobjektet). Det aktuella transformationsobjektet modifieras. Metoden `setTransform` sätter den grafiska kontextens transformationsobjekt till ett nytt transformationsobjekt.

En egen skala

Man kan använda egna enheter när man definierar en figur. Man kan använda sin egen skala i ett eget koordinatsystem. Alla koordinater och längder kan till exempel uttryckas i centimeter. En rektangel, där alla uppgifter anges i centimeter, kan definieras på följande vis:

```
Rectangle2D    rekt = new Rectangle2D.Double (10, 5, 12, 8);
```


© FADIL GALJIĆ

En rektangel definieras, vars övre vänstra hörn finns i punkten med koordinaterna `10` cm och `5` cm, och vars dimensioner är `12` cm och `8` cm.

Innan den angivna rektangeln ritas i en grafisk komponent (till exempel en panel), måste man ange hur de egna enheterna (centimeter) ska översättas till pixlar (hur koordinaterna byts ut). Detta kan göras med metoden `scale` i klassen `Graphics2D`. Om `g` är den aktuella grafiska kontexten, kan man göra så här:

```
g.scale (10, 10);
```

Det här metodanropet ändrar inställningarna i det aktuella transformationsobjektet i den grafiska kontexten. Efter denna ändring översätts användarkoordinaterna till komponentkoordinater genom att användarkoordinaterna multipliceras med `10`. Den angivna rektangeln ritas som en rektangel vars övre vänstra hörn finns i punkten med koordinaterna `10 x 10` pixlar och `10 x 5` pixlar, och vars dimensioner är `10 x 12` pixlar och `10 x 8` pixlar. *Skalfaktorn* `10` används både i `x`-riktningen (det första argumentet till metoden `scale`) och i `y`-riktningen (det andra argumentet). På så sätt behålls proportionerna mellan rektangelns sidor. Det går också att använda olika skalfaktorer för olika riktningar (två olika argument kan tillföras till metoden `scale`).

Ändringen av transformationsobjektet medför även att en linjes tjocklek ändras. Den nya tjockleken beror på skalfaktorerna (en horisontell linjes tjocklek, till exempel, multipliceras med den skalfaktor som gäller i `y`-riktningen). Eftersom den förvalda tjockleken är `1` pixel, blir linjens tjocklek efter anropet till metoden `scale` `10 x 1` pixlar. För att undvika så tjocka linjer, måste man definiera en egen tjocklek (i sina egna enheter) innan metoden `scale` anropas. Man kan bestämma sig för att linjens tjocklek ska vara `0.2` cm (efter transformationen blir tjockleken `10 x 0.2` pixlar). Den angivna rektangeln kan i så fall ritas så här:

```
BasicStroke    stroke = new BasicStroke (0.2f);
g.setStroke (stroke);
g.scale (10, 10);
g.draw (rekt);
```

Efter anropet till metoden `scale` ritas alla figurer på så sätt att användarkoordinaterna och användarlängderna multipliceras med de angivna skalfaktorerna. För att återgå till de ursprungliga inställningarna måste man ändra transformationsobjektet en gång till. Objektet måste ändras så att effekterna av den föregående ändringen raderas. I det angivna fallet måste användarkoordinaterna och användarlängderna multipliceras med `1/10`. Multiplikationen med `10` och multiplikationen med `1/10` tar ut

© FADIL GALJIĆ

varandra, och man återgår till den ursprungliga transformationen. Metoden `scale` ska anropas så här:

```
g.scale (1.0 / 10, 1.0 / 10);
```

Metoden `scale` kan användas för att ändra skalfaktorerna i det aktuella transformationsobjektet. Men detta kan även göras på ett annat sätt. Man kan skapa ett transformationsobjekt av typen `AffineTransform` som definierar de nya skalfaktorerna, och sedan kombinera detta objekt med det aktuella transformationsobjektet. Ett transformationsobjekt som innehåller två skalfaktorer kan skapas genom anrop till metoden `getScaleInstance` i klassen `AffineTransform`. Detta är en statisk metod, som tar emot skalfaktorerna som argument. Så här gör man:

```
AffineTransform   at1 = AffineTransform.getScaleInstance (10, 10);
```

Detta transformationsobjekt kombineras med det aktuella transformationsobjektet (en komposition av transformationer bildas) med metoden `transform` i klassen `Graphics2D`. Detta görs så här:

```
g.transform (at1);
```

Om det här sättet används, kan den angivna rektangeln ritas så här:

```
BasicStroke    stroke = new BasicStroke (0.2f);
g.setStroke (stroke);
AffineTransform   at0 = g.getTransform ();
AffineTransform   at1 = AffineTransform.getScaleInstance (10, 10);
g.transform (at1);
g.draw (rekt);
g.setTransform (at0);
```

De ursprungliga transformationsinställningarna sparas (som objektet `at0`), och sedan återgår man till dessa inställningar (med metoden `setTransform`) när rektangeln har ritats.

Förflytta en figur

Origo för det koordinatsystem som används kan placeras var som helst i den komponent som man ritar i. På så sätt kan koordinatsystemets axlar förflyttas relativt komponenten (och dess koordinatsystem). Tillsammans med koordinatsystemet förflyttas även de figurer som definierats i detta koordinatsystem. Dessa figurer förflyttas inte relativt användarkoordinatsystemet, utan relativt den aktuella komponenten.

En rektangel (i användarkoordinatsystemet) kan definieras så här:

```
Rectangle2D    rekt2 = new Rectangle2D.Double (10, 20, 120, 80);
```


© FADIL GALJIĆ

Rektangelns övre vänstra hörn finns i punkten med koordinaterna `10` och `20`, och dess dimensioner är `120` och `80` användarenheter. Innan rektangeln ritas, kan användarkoordinatsystemet förflyttas i en godtycklig punkt i komponenten. För detta ändamål används metoden `translate` i klassen `Graphics2D`. Om `g` är den aktuella grafiska kontexten, kan förflyttningen utföras så här:

```
g.translate (100, 150);
```

Metoden `translate` tar emot två argument, som definierar förflyttningen. Om den ursprungliga skalan används (där en användarenhet avbildas till en pixel), förflyttas användarkoordinatsystemet `100` pixlar åt höger i den horisontella riktningen och `150` pixlar nedåt i den vertikala riktningen (om man använder egna enheter, blir förflyttningen `100` och `150` sådana enheter). Koordinatsystemets origo hamnar i punkten `(100, 150)`.

Tillsammans med koordinatsystemet förflyttas även den rektangeln som definierats i koordinatsystemet. Rektangeln behåller sina dimensioner (`120` pixlar och `80` pixlar), men dess övre vänstra hörn förflyttas till den punkt vars koordinater är `10 + 100` pixlar och `20 + 150` pixlar (relativt den komponent som man ritar i).

Anropet till metoden `translate` ändrar inställningarna i det aktuella transformationsobjektet. Som en konsekvens av detta förflyttas alla figurer som ritas efter detta anrop. För att återgå till de gamla inställningarna måste metoden `translate` anropas en gång till, för att radera effekten av den föregående ändringen. I detta fall ska metoden `translate` anropas så här:

```
g.translate (-100, -150);
```

Förflyttningen kan även utföras med hjälp av ett objekt av typen `AffineTransform` som beskriver förflyttningen. Detta objekt kombineras med det aktuella transformationsobjektet med metoden `transform` (det bildas en komposition av transformationer). Ett objekt som representerar en förflyttning kan erhållas med metoden `getTranslateInstance` i klassen `AffineTransform`. Det är en statisk metod som tar förflyttningsparametrar som argument. Så här kan man göra:

```
AffineTransform    at0 = g.getTransform ();
AffineTransform    at1 =
              AffineTransform.getTranslateInstance (100, 150);
g.transform (at1);
g.draw (rekt);
g.setTransform (at0);
```

© FADIL GALJIĆ

De ursprungliga inställningarna sparas, och när rektangeln har ritats återgår man till dem.

Rotera en figur

Användarkoordinatsystemet kan roteras med en given vinkel runt sitt origo, i den komponent som man ritar i. På så sätt kan även de figurer som man definierar i detta koordinatsystem roteras. Dessa figurer roteras inte relativt användarkoordinatsystemet, utan relativt den komponent som de ritas i (relativt den komponentens koordinatsystem).

En rektangel kan definieras (i användarkoordinatsystemet) så här:

```
Rectangle2D    rekt = new Rectangle2D.Double (100, 50, 120, 80);
```

Om den här rektangeln ritas, blir dess kanter parallella med den aktuella komponentens kanter. Rektangeln kan roteras relativt komponenten genom att användarkoordinatsystemet roteras. Detta utförs med metoden `rotate` i klassen `Graphics2D`. Om `g` är den aktuella grafiska kontexten, kan man göra så här:

```
g.rotate (Math.PI / 12);
```

Användarkoordinatsystemet roteras med en given vinkel kring sitt origo. Rotationsvinkeln anges i radianer (`Math.PI/12` radianer är `15` grader). Om en positiv vinkel anges, utförs rotationen medsols. Om rektangeln ritas efter anropet till metoden `rotate`, kommer den att vara roterad relativt den komponent som den ritas i. Rektangelns kanter är parallella till användarkoordinatsystemets axlar, vilket innebär att rektangeln är roterad relativt det ursprungliga koordinatsystemet.

Metoden `rotate` ändrar den grafiska kontextens transformationsobjekt, och alla figurer som ritas efter anropet till metoden blir roterade. För att återgå till de ursprungliga inställningarna, måste metoden `rotate` anropas med en vinkel som raderar den föregående ändringen. Man gör så här:

```
g.rotate (-Math.PI / 12);
```

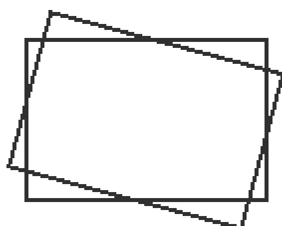
Ofta vill man rotera en figur kring en viss punkt (till exempel kring ett av dess hörn, eller kring dess mittpunkt). Man kan göra detta genom att förflytta figuren så att punkten sammanfaller med komponentens övre vänstra hörn, rotera användarkoordinatsystemet (och därmed figuren), och förflytta figuren tillbaka så att punkten återkommer till sin gamla plats. Alla dessa transformationer kan utföras med en variant av metoden `rotate` som tar emot tre argument. Det första argumentet är rotationsvin-

© FADIL GALJIĆ

keln, och de andra och tredje argumenten är rotationspunktens koordinater. Metoden kan användas så här:

```
g.rotate (Math.PI / 12, 160, 90);
```

Den angivna rektangeln kan ritas efter den här transformationen. Rektangeln kommer då att ha roterats med `15` grader medsols (se nedanstående bild) kring sin mittpunkt (rektangelns mittpunkt har koordinaterna `100 + 120/2` och `50 + 80/2`).



En rotation kan även utföras med hjälp av ett transformationsobjekt av typen `AffineTransform`, som representerar en rotation. Detta objekt kombineras med det aktuella transformationsobjektet med metoden `transform` (det bildas en komposition av transformationer). Ett objekt som representerar en rotation erhålls med metoden `getRotateInstance` i klassen `AffineTransform`. Detta är en statisk metod som tar rotationsparametrar som argument. Så här kan man göra:

```
AffineTransform    at0 = g.getTransform ();
AffineTransform    at1 = AffineTransform.getRotateInstance (
                                   Math.PI / 12, 160, 90);
g.transform (at1);
g.draw (rekt);
g.setTransform (at0);
```

De ursprungliga inställningarna sparas, och när rektangeln har ritats återgår man till dem.

Skjuva en figur

En figur kan skjuvas. Man kan skapa en rektangel, och därefter utföra en skjuvningstransformation av användarkoordinaterna. När rektangeln sedan ritas, skapas en figur som har följande form:

© FADIL GALJIĆ

En skjuvning utförs med metoden `shear` i klassen `Graphics2D`. Om g är den aktuella grafiska kontexten, kan detta göras så här:

```
g.shear (-0.2, 0);
```

Efter den här transformationen avbildas en x-koordinat i användarkoordinatsystemet till `x - 0.2 *` y, och en y-koordinat till `y + 0 * x` pixlar. En rektangel i användarkoordinatsystemet ritas som en romboid i den aktuella grafiska komponenten. Skjuvningen regleras via två argument till metoden `shear`. En figur kan skjuvas endast längs x-axeln (det andra argumentet är 0), eller endast längs y-axeln (det första argumentet är 0), eller längs båda axlarna (det ursprungliga koordinatsystemets axlar). En figur kan skjuvas i en riktning (motsvarande argument positivt), eller i motsatt riktning (motsvarande argument negativt).

Metoden `shear` ändrar den aktuella grafiska kontextens transformationsobjekt. För att återgå till den föregående transformationen, anropas metoden `shear` med motsatta argument:

```
g.shear (0.2, 0);
```

Skjuvningen kan även regleras med ett objekt av typen `AffineTransform` som representerar en skjuvning. Om `rekt` representerar en rektangel i användarkoordinatsystemet, kan man göra så här:

```
AffineTransform    at0 = g.getTransform ();
AffineTransform    at1 =
               AffineTransform.getShearInstance (-0.2, 0);
g.transform (at1);
g.draw (rekt);
g.setTransform (at0);
```

Komposition av transformationer

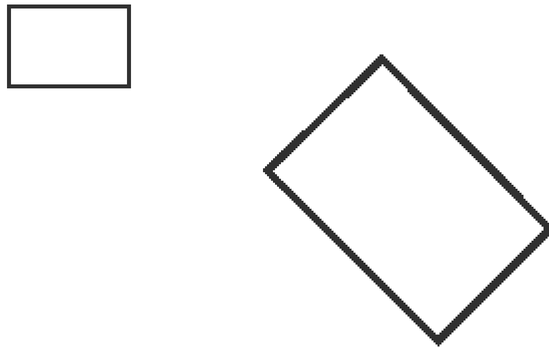
Flera koordinattransformationer kan utföras efter varandra. Varje transformation ändrar den grafiska kontextens transformationsobjekt på ett speciellt sätt. En specifik transformation kombineras med den gällande transformationen och bildar en komposition av transformationer. Om `g` är den aktuella grafiska kontexten, kan detta göras så här:

© FADIL GALJIĆ

```
Rectangle2D    rekt = new Rectangle2D.Double (50, 50, 60, 40);
g.draw (rekt);

AffineTransform    at0 = g.getTransform ();
AffineTransform    at1 =
             AffineTransform.getTranslateInstance (50, 80);
g.transform (at1);
AffineTransform    at2 = AffineTransform.getScaleInstance (2, 2);
g.transform (at2);
AffineTransform    at3 =
        AffineTransform.getRotateInstance (Math.PI / 4, 80, 70);
g.transform (at3);

g.draw (rekt);
g.setTransform (at0);
```

Här definieras tre elementära transformationer (förflyttning, skalning och rotation), och dessa kombineras med det aktuella transformationsobjektet. Det bildas en komposition av den ursprungliga transformationen, transformationen `at1`, transformationen `at2` och transformationen `at3`. Därefter ritas rektangeln `rekt`, som definierats i användarkoordinatsystemet. På så vis fås en rektangel, som är roterad, förstorad och förflyttad relativt den aktuella komponentens koordinatsystem.

När man ritar en figur, applicerar man de angivna elementära transformationerna i omvänd ordning relativt den ordning i vilken de anges. I det här fallet roteras rektangeln först (därför anges den ursprungliga rektangelns mittpunkt som rotationspunkt), därefter skalas rektangeln, och till sist förflyttas den (se nedanstående bild).



Istället för att använda tre olika transformationsobjekt (`at1`, `at2` och `at3`), kan man använda ett och samma transformationsobjekt. Ett objekt av typen `AffineTransform` kan justeras med instansmetoderna `setToScale`, `setToTranslation`, `setToRotation` och `setToShear`. På så sätt kan man

© FADIL GALJIĆ

skapa ett objekt som inkluderar flera transformationer. Man kan till exempel kombinera en förflyttning och en skalning::

```
AffineTransform    at =
               AffineTransform.getTranslateInstance (50, 80);
at.setToScale (0.75, 0.75);
```


© FADIL GALJIĆ

Kapitel 5
Grafiska användargränssnitt

Ett grafiskt användargränssnitt
Ett konsolprogram
Ett program med grafiskt användargränssnitt

Grafiska komponenter
Standardkomponenter
Komponenternas utseende
Ramar runt komponenter

Ordna komponenter i en behållare
En layouthanterare
Vanliga layoutstrategier
En flexibel layoutstrategi
Layoutkomponenter
Interna fönster

Hantera händelser
Komponenter, händelser och lyssnare
Olika sätt att implementera en lyssnarklass
Mushändelser
Tangentbordshändelser

© FADIL GALJIĆ

Ett grafiskt användargränssnitt

Ett konsolprogram

En konkret plattform har ett inbyggt fönster, som kan användas för att skriva olika kommandon och för att få olika typer av information. Ett sådant fönster kallas för *konsolfönster*, eller *terminalfönster*. Detta fönster kan även användas i samband med ett Javaprogram. Olika uppgifter kan matas in via fönstret, och olika meddelanden och resultat kan visas i fönstret. Ett sådant program, som kommunicerar med användaren via plattformens konsolfönster, kallas för ett *konsolprogram*.

Ett konsolprogram behöver ett lämpligt utmatningsverktyg. Detta verktyg används för att mata ut information av olika slag till plattformens konsolfönster. För att skriva ut till konsolfönstret kan ett objekt av typen `java.io.PrintWriter`, och dess metoder `print` och `println`, användas. Ett konsolprogram behöver även ett lämpligt inmatningsverktyg. Detta verktyg används för att mata in information av olika slag via konsolfönstret. Ett objekt av typen `java.util.Scanner` och dess metod `nextLine` kan användas för detta ändamål.

Ett utmatningsverktyg och ett inmatningsverktyg i ett konsolprogram kan skapas så här:

```
PrintWriter    out = new PrintWriter (System.out, true);
Scanner    in = new Scanner (System.in);
```

Med dessa verktyg kan man kommunicera med användaren. Olika meddelanden och resultat kan skrivas ut till användaren, och information av olika slag kan fås från användaren. Programmet kan till exempel ta emot olika heltal från användaren, och skriva ut dessa heltal och deras kvadrater. Ett särskilt program som gör detta kan skapas:

```
class HeltalsKvadraterKonsol
{
    public static void main (String[] args)
    {
        int    heltal = 0;
        out.print ("ett heltal: ");
        out.flush ();
        String    input = in.nextLine ();
        while (!input.equals (""))
        {
            heltal = Integer.parseInt (input);
            out.println ("heltalet: " + heltal);
```

 © FADIL GALJIĆ

```
            out.println ("dess kvadrat: " + heltal * heltal);
            out.println ();

            out.print ("ett heltal: ");
            out.flush ();
            input = in.nextLine ();
        }
    }
}
```

Ett heltal matas in, och sedan skrivs heltalet och dess kvadrat ut. Detta upprepas så länge användaren matar in heltal. Loopen avslutas när användaren matar in en tom rad (när användaren trycker på returtangenten utan att mata in något). Om en teckensträng matas in som inte representerar ett heltal av typen `int`, kastar programmet ett undantag av typen `java.lang.NumberFormatException`. Ett lämpligt meddelande skrivs ut till konsolfönstret, och programmet avslutas.

Ett program med grafiskt användargränssnitt

Ett fönster med olika komponenter

Istället för att använda plattformens konsolfönster, kan ett Javaprogram skapa och använda egna fönster. Ett program kan utforma ett *grafiskt användargränssnitt* (*GUI* – Graphical User Interface) mot användaren. Användaren kan sedan kommunicera med programmet via detta gränssnitt. Ett sådant program kallas för *grafiskt program* (GUI-program).

Ett grafiskt program har normalt ett huvudfönster. Man skapar ett sådant fönster, och placerar olika grafiska komponenter i det. Komponenternas antal, typ, storlek och placering beror på programmets funktion. För att kunna mata in olika heltal och visa deras kvadrater, behövs det sådana grafiska komponenter som gör detta möjligt. Det behövs en inmatningskomponent och en komponent där resultaten visas (en display). Så här kan det grafiska användargränssnittet se ut:

© FADIL GALJIĆ

Användargränssnittet har ett fönster, med en textarea mitt i fönstret. Det inmatade heltalet och dess kvadrat skrivs ut i denna textarea. I nedre delen av fönstret finns en inmatningspanel. Denna panel innehåller en etikett med ett meddelande, och ett textfält. Användaren skriver in ett heltal i textfältet och trycker på returtangenten. Programmet tar texten i textfältet och använder den på ett lämpligt sätt.

Ett fönster med en textarea och en inmatningspanel kan skapas så här:

```
class HeltalsKvadraterGUI
{
    public static void main (String[] args)
    {
        JLabel    label = new JLabel ("Ett heltal:");
        JTextField    textFalt = new JTextField (20);
        JPanel    panel = new JPanel ();
        panel.add (label);
        panel.add (textFalt);

        JTextArea    textArea = new JTextArea (40, 20);

        JFrame    frame = new JFrame (" Heltalskvadrater");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        frame.setSize (400, 300);
        frame.setLocation (120, 80);

        frame.add (panel, "South");
        frame.add (textArea, "Center");

        frame.setVisible (true);
    }
}
```

 © FADIL GALJIĆ

Här skapas en etikett (som ett objekt av typen `javax.swing.JLabel`), ett textfält (som ett objekt av typen `javax.swing.JTextField`) och en panel (som ett objekt av typen `javax.swing.JPanel`). Man anger den text som ska visas på etiketten (`Ett heltal:`) och textfältets längd (`20` tecken). Sedan placeras etiketten och textfältet i panelen. Det bildas en sammansatt komponent, som kan användas som en grafisk enhet. Därefter skapas en textarea (som ett objekt av typen `javax.swing.JTextArea`), och antalet rader (`40`) och kolumner (`20`) för denna textarea anges. Slutligen skapas ett fönster (som ett objekt av typen `javax.swing.JFrame`), och inmatningspanelen och textarean placeras i detta fönster. Fönstret visas, och på så sätt visas även de komponenter som finns i fönstret.

Hantera händelser i det grafiska användargränssnittet

Man kan skapa olika grafiska komponenter och placera dem på ett lämpligt sätt i ett fönster. På så sätt bestäms ett grafiskt användargränssnitts utseende. Detta är dock inte tillräckligt. Normalt vill man kunna kommunicera med programmet via användargränssnittet. Man vill kunna mata in olika uppgifter, välja mellan olika alternativ och navigera i programmet. För att detta ska kunna ske måste de olika grafiska komponenterna ges en innebörd. Man måste bestämma vad som ska hända när användaren trycker på en knapp, skriver in text i ett textfält, väljer ett alternativ från en lista och så vidare. Man måste skapa den kod som ska utföras när en komponent används, och på något sätt knyta denna kod till komponenten. De olika komponenterna i det grafiska användargränssnittet måste på något sätt "levandegöras".

Ett textfält kan vara en del av ett grafiskt användargränssnitt. Användaren kan skriva en text i textfältet, och trycka på returtangenten för att signalera till programmet att texten ska tas emot. Vad ska hända i så fall? Om man inte preciserat vad som ska hända, händer ingenting. Det går inte att mata in något via textfältet. Det finns en komponent, men inte motsvarande funktion. Man måste skriva motsvarande kod och knyta koden till textfältet.

När användaren trycker på returtangenten inom ett textfält, skapas en *händelse*. När användaren trycker på en knapp, skapas en annan händelse. När användaren väljer ett alternativ från en lista, skapas åter en händelse. Händelser av olika typer kan skapas i ett grafiskt användargränssnitt. För att något verkligen ska hända när användaren skapar en händelse, måste man precisera vad som ska hända. Händelsen måste *hanteras* på något

sätt. Koden som ska utföras när händelsen inträffar måste skrivas. Det måste finnas en *lyssnare* bakom en komponent, som automatiskt reagerar varje gång en händelse inträffar i komponenten. Man måste skapa ett lämpligt objekt (en lyssnare), och installera detta (*registrera det*) hos en viss komponent i ett grafiskt användargränssnitt. Objektet måste ha en lämplig metod, som ska anropas automatiskt varje gång användaren skapar en händelse.

För att kunna hantera händelser i samband men komponenter av en viss typ, skapas en klass med en eller flera lämpliga metoder. Man skapar sedan ett objekt (en lyssnare) av denna klass, och registrerar objektet hos en konkret komponent i ett grafiskt användargränssnitt. När en händelse i komponenten inträffar, anropas automatiskt motsvarande metod i samband med det registrerade objektet (lyssnaren). Eftersom metoden anropas automatiskt, måste den ha ett i förväg känd prototyp. Därför har dessa prototyper definierats i olika gränssnitt i Javas standardbibliotek. De flesta av dessa gränssnitt finns i paketet `java.awt.event`. Några av gränssnitten finns i paketet `javax.swing.event`. För att kunna hantera olika händelser, måste man känna till dessa gränssnitt och deras metoder.

Varje händelse i ett grafiskt användargränssnitt är av en viss typ. Händelsen som skapas när returtangenten trycks i ett textfält, till exempel, är av typen `java.awt.event.ActionEvent`. När en händelse inträffar, skapas automatiskt ett objekt av motsvarande typ som representerar händelsen. Objektet innehåller, förutom andra uppgifter, även en referens till den komponent där händelsen genererats. Tack vare det kan en händelses källa bestämmas. Java tillför det objekt som representerar en händelse som argument till den metod som anropas automatiskt. Händelseobjektet och dess metoder kan användas i den anropade metoden.

I paketet `java.awt.event` i Javas standardbibliotek definieras ett gränssnitt som heter `ActionListener`. Detta gränssnitt definieras så här:

```
public interface ActionListener
{
    void actionPerformed (ActionEvent e);
}
```

Det här gränssnittet definierar en typ av lyssnare. Ett objekt som vill lyssna och reagera på händelser av typen `ActionEvent` måste ha metoden `actionPerformed`. Det är den metoden som anropas automatiskt när en händelse av typen `ActionEvent` inträffar. Om man vill reagera när användaren trycker på returtangenten i ett textfält, måste man skapa en klass som implementerar gränssnittet `ActionListener`. Sedan måste ett objekt av denna klass skapas och registreras som lyssnare hos textfältet. En lyss-

 © FADIL GALJIĆ

nare av typen `ActionListener` registreras hos en komponent med komponentens metod `addActionListener`.

Om ett heltal matas in via ett textfält, och heltalet och dess kvadrat skrivs ut i en textarea, kan lyssnarklassen utformas så här:

```
class HandelseHanterare implements ActionListener
{
    private JTextField    textFalt;
    private JTextArea     textArea;

    public HandelseHanterare (JTextField textFalt,
                              JTextArea textArea)
    {
        this.textFalt = textFalt;
        this.textArea = textArea;
    }

    public void actionPerformed (ActionEvent e)
    {
        String    input = textFalt.getText ();
        textFalt.setText ("");

        int    heltal = Integer.parseInt (input);

        textArea.setText ("");
        textArea.append ("heltalet: " + heltal + "\n");
        textArea.append ("dess kvadrat: " + heltal * heltal);
    }
}
```

Klassen `HandelseHanterare` (ett valfritt namn) implementerar gränssnittet `ActionListener`, och på så sätt definieras en typ av lyssnare. Ett objekt av klassen kan skapas, och registreras som lyssnare hos ett textfält. Med ett textfält som heter `textFalt` och en textarea som heter `textArea`, kan man göra så här:

```
HandelseHanterare    hanterare =
                 new HandelseHanterare (textFalt, textArea);
textFalt.addActionListener (hanterare);
```

Nu finns det ett textfält, och en lyssnare (ett objekt av klassen `HandelseHanterare` som implementerar gränssnittet `ActionListener`) bakom textfältet. När användaren skriver något i textfältet och trycker på returtangenten, skapas en händelse. Ett objekt av typen `ActionEvent`, som representerar denna händelse, skapas automatiskt. Java anropar metoden `actionPerformed` i samband med den registrerade lyssnaren, med det objekt som är av typen `ActionEvent` som argument. Koden i metoden `actionPerformed` utförs som en reaktion på tryckningen på returtangenten i

© FADIL GALJIĆ

textfältet. Denna kod utförs varje gång som returtangenten trycks i textfältet.

Ett objekt av typen `HandelseHanterare` har tillgång till ett textfält och en textarea. Objektets beteende definieras i metoden `actionPerformed`. När en händelse inträffar, avläses den text som finns i textfältet (det kan vara en tom teckensträng) och textfältet rensas (textfältet förbereds för nästa inmatning). Texten i textfältet avläses med metoden `getText`. Textfältet rensas med metoden `setText` (textfältets text sätts till en tom teckensträng). Efter inmatningen omvandlas den inmatade texten till motsvarande heltal. Detta heltal och dess kvadrat skrivs sedan ut till motsvarande textarea. På så sätt avslutas reaktionen på den händelsen som inträffat i textfältet. Ett nytt heltal kan matas in, och även dess kvadrat kan visas. Detta kan upprepas hur många gånger som helst, när som helst.

Om en teckensträng som inte representerar ett heltal av typen `int` matas in, kastas ett undantag av typen `java.lang.NumberFormatException`. Ett lämpligt felmeddelande skrivs ut till standardutmatningsenheten (man kan hantera undantaget och visa felmeddelandet via en lämplig dialog), men programmet avslutas inte. Det går att fortsätta mata in heltal och visa deras kvadrater. I detta avseende skiljer sig ett grafiskt program från ett konsolprogram.

Uppbyggnad av ett grafiskt program

Ett grafiskt program har normalt ett huvudfönster och komponenter av olika typer i detta fönster. Man analyserar ett programs funktion, och väljer sådana grafiska komponenter som kan uppfylla denna funktion. Komponenternas storlek och placering bestäms. På så sätt preciseras utseendet för programets grafiska användargränssnitt.

Förutom utseendet, måste även det grafiska användargränssnittets beteende definieras. Man skapar olika klasser (lyssnarklasser – dessa klasser kan definieras i samma fil som huvudprogrammet, eller i separata filer) som implementerar lämpliga gränssnitt (lyssnargränssnitt). Objekt av dessa klasser (lyssnare) skapas och registreras hos motsvarande komponenter i det grafiska användargränssnittet. När en händelse inträffar i en komponent, aktiveras motsvarande lyssnare (det kan finnas flera registrerade lyssnare) och reagerar på händelsen. Koden bakom komponenten anropas manuellt (genom att motsvarande händelse skapas), istället för att motsvarande anrop byggs in i programmet. Det är användaren som bestämmer om någon kod ska anropas, när den ska anropas och hur

 © FADIL GALJIĆ

många gånger den ska anropas. Genom att skapa olika händelser vid olika tillfällen, bestämmer användaren en bana genom programmet.

Ett grafiskt program används via dess grafiska användargränssnitt. Användaren matar in olika uppgifter och väljer olika alternativ via detta användargränssnitt. Olika resultat och meddelanden erhålls via användargränssnittet. Användaren har en intelligent yta framför sig, och kommunicerar med programmet via denna yta.

© FADIL GALJIĆ

Grafiska komponenter

Standardkomponenter

AWT-komponenter

För att skapa ett grafiskt användargränssnitt används olika grafiska komponenter. Dessa komponenter är byggstenar i ett grafiskt användargränssnitt. Bland komponenterna finns olika knappar, menyer, textkomponenter, tabeller och så vidare. Javas standardbibliotek innehåller ett stort antal klasser som representerar olika grafiska komponenter. Man kan använda dessa klasser som de är, eller skapa egna klasser utifrån dessa standardklasser.

Ursprungligen kom Java med en uppsättning grafiska komponenter som var definierade i paketet `java.awt`. Den här uppsättningen grafiska komponenter kallas för *AWT* (Abstract Window Toolkit). Komponenterna definieras via ett antal klasser, ordnade i en klasshierarki. I roten av denna klasshierarki finns klassen `Component` (en abstrakt klass, direkt subklass till klassen `java.lang.Object`). Utifrån denna klass skapas ett antal klasser som representerar konkreta komponenter. Dessa klasser är: `Button`, `Label`, `TextComponent` (med subklasserna `TextField` och `TextArea`), `List` och så vidare. En viktig subklass till klassen `Component` är `Container`. Denna klass representerar en behållare, en komponent som kan användas för att lagra andra komponenter.

En viktig komponent i ett grafiskt användargränssnitt är ett fönster. Ett fönster används för att lagra och visa alla andra komponenter. I AWT definieras ett fönster i klassen `Window`. För att andra komponenter ska kunna lagras i ett fönster, definieras denna klass som en subklass till klassen `Container`. Klassen `Window` representerar ett grundläggande fönster. För att få mer användbara fönster har två subklasser till klassen `Window` skapats. Dessa klasser är `Frame` och `Dialog`.

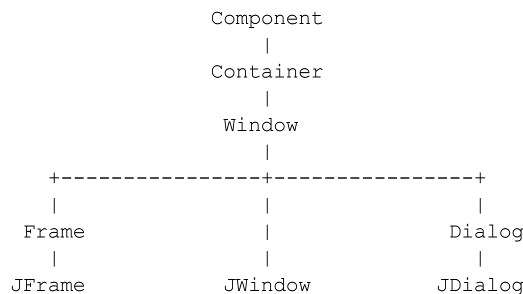
Tillsammans med paketet `java.awt` skapades även paketet `java.awt.event`. I detta paket definieras allt det som är nödvändigt för hanteringen av olika händelser i ett grafiskt användargränssnitt (uppbyggt av olika AWT-komponenter). Olika typer av händelser definieras via klasserna `ActionEvent`, `WindowEvent`, `MouseEvent` och andra klasser. Det definieras också ett antal gränssnitt, som är nödvändiga för att kunna fånga och hantera olika händelser. Bland dessa kan nämnas `ActionListener`, `WindowListener` och `MouseListener`. Paketen `java.awt` och

© FADIL GALJIĆ

`java.awt.event` bildar en enhet, som är tillräcklig för att kunna skapa och hantera ett grafiskt användargränssnitt.

Swing-komponenter

I början användes AWT-komponenter för att skapa grafiska användargränssnitt i olika Javaprogram. Under tiden utvecklades dock en bättre uppsättning grafiska komponenter. Klasserna som representerar dessa komponenter finns i paketet `javax.swing`. Den här uppsättningen grafiska komponenter kallas för *Swing*, och det är den uppsättningen som normalt används idag. Swing-paketet utgår från AWT-paketet, och använder ett antal klasser från detta paket. Olika fönster definieras genom att lämpliga subklasser till klasserna `Window`, `Frame` och `Dialog` skapas. Dessa nya klasser kallas för `JWindow`, `JFrame` och `JDialog`. Den motsvarande klasshierarkin kan representeras så här:



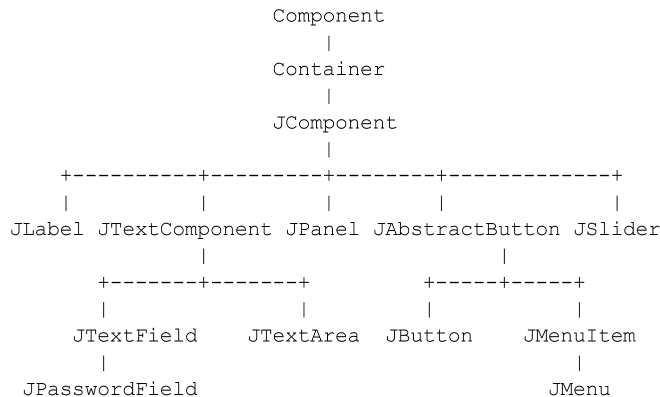
Klasserna `JWindow`, `JFrame` och `JDialog` representerar självständiga fönster. De är så kallade toppnivåkomponenter. Dessa fönster används för att visa andra komponenter, och de kan inte ingå som komponenter i ett annat fönster. Dessa komponenter brukar kallas för *tungviktiga* komponenter.

Förutom olika fönster, definieras i paketet `javax.swing` en rik uppsättning andra komponenter. Knappar, menyer, textkomponenter, listor och andra komponenter definieras. Dessa komponenter kallas för *lättviktiga* komponenter. Komponenterna representeras via klasserna `JButton`, `JLabel`, `JTextComponent` (med subklasserna `JTextField`, `JTextArea` och andra), `JList` och så vidare. För att dessa komponenter inte ska förväxlas med motsvarande AWT-komponenter, har de ett `J` i början av motsvarande namn (`JButton` istället för `Button`, `JLabel` istället för `Label`, och så vidare). Förutom de komponenter som finns i AWT, definieras i paketet `ja-`

© FADIL GALJIĆ

`vax.swing` även ett antal nya komponenter. Swing-paketet är ett bättre och rikare paket.

De komponenter som inte representerar självständiga fönster (alltså lättviktiga komponenter), definieras som subklasser till klassen `javax.swing.JComponent`. Klassen `JComponent` definieras som subklass till klassen `java.awt.Container`. En del av den motsvarande klasshierarkin kan representeras så här:



Av bilden framgår att ett antal AWT-klasser används som (direkta eller indirekta) superklasser till klasserna i paketet `javax.swing`. Förutom detta används även de klasser och gränssnitt som definieras i paketet `java.awt.event`, för att hantera olika händelser i samband med Swing-komponenter. Ett antal nya händelseklasser och lyssnargränssnitt definieras i paketet `javax.swing.event`.

Klasserna `Component` och `Container` är superklasser till alla Swing-klasser. Man måste känna till dessa klasser för att kunna använda deras metoder. Klassen `Component` definierar de metoder som gäller en komponents färg, storlek, font och så vidare. Här finns metoderna `getBackground` och `setBackground` (för en komponents färg), `getForeground` och `setForeground` (den färg som används för det som skrivs eller ritas i en komponent), `getFont` och `setFont` (för en komponents font), `getLocation` och `setLocation` (för en komponents position), `getSize` och `setSize` (för en komponents storlek), `getWidth` och `getHeight` (för en komponents dimensioner), och så vidare.

Klassen `Container` representerar en behållare för andra grafiska komponenter (som inte representerar självständiga fönster). Huvudmetoder i

 © FADIL GALJIĆ

denna klass är metoderna `add` och `remove`, som gör det möjligt att lägga till en komponent i behållaren eller ta bort en komponent från behållaren. Denna klass definierar även metoderna `getLayout` och `setLayout` (för att kunna ordna olika komponenter i en behållare), `getComponentCount` (ger antalet komponent i en behållare), `getComponents` (ger referenser till komponenterna i en behållare) och så vidare.

Alla Swing-komponenter är behållare (eftersom alla dessa komponenter har klassen `Container` som sin superklass). Det betyder att det går att lagra olika komponenter i en godtycklig Swing-komponent, till exempel i en knapp. Men det finns en särskild komponent som är tänkt att användas just som behållare för andra lättviktiga komponenter. Det är en panel (ett objekt av typen `JPanel`). En panel används för att lagra och gruppera andra grafiska komponenter. En panel är en typisk behållare i Java. Man brukar gruppera ett antal relaterade komponenter i en särskild panel, och placera denna panel på en särskild plats. Det går också att lagra alla komponenter i en panel, och placera denna panel i ett fönster. Vanligtvis skapar man en egen klass, som definierar en typ av paneler. Klassen definieras som en subklass till klassen `JPanel`, och i denna klass specificeras de komponenter som ska ingå i en panel av denna typ. Man specificerar också hur komponenterna ska ordnas i panelen. En panel används i Java som en behållare, och som en kanvas när olika figurer ritas.

Klassen `JComponent` är en gemensam superklass till de Swing-komponenter som inte representerar självständiga fönster. Klassen definierar ett antal Swing-specifika metoder, bland annat metoderna `getBorder` och `setBorder`, som gör det möjligt att placera ramar runt olika grafiska komponenter. Klassen definierar även metoderna `getToolTipText` och `setToolTipText`, som gör det möjligt att knyta en förklarande text till en grafisk komponent. Texten visas automatiskt när muspekaren placeras över komponenten. Med metoden `setOpaque` i klassen `JComponent` kan en grafisk komponent göras genomskinlig.

Grafiska komponenter av en viss typ har ett antal metoder som är specifika just för denna typ. Förutom dessa metoder finns ett stort antal metoder som är gemensamma för alla grafiska komponenter. Dessa metoder definieras i klasserna `Component` och `Container`. Metoderna som definieras i klassen `JComponent` är gemensamma för alla lättviktiga Swing-komponenter. Man måste därför känna till klasserna `Component`, `Container` och `JComponent`.

© FADIL GALJIĆ

Komponenternas utseende

Swing-komponenter kan ritas på olika sätt i ett grafiskt användargränssnitt. En och samma komponent kan få olika utseenden. Man kan välja olika *Look & Feel* för grafiska komponenter i ett grafiskt användargränssnitt.

Olika plattformar ritar grafiska komponenter på olika sätt. Ett fönster eller en knapp ser inte ut på samma sätt i Windows som i Unix. För att få ett plattformsoberoende utseende har Java valt att rita olika Swing-komponenter själv, istället för att överlämna denna uppgift till det underliggande operativsystemet. Java tillåter bara det underliggande systemet att rita självständiga fönster. Därför ser ett fönster i ett Javaprogram olika ut på olika plattformar. Ett fönster har en lokal Look & Feel.

De komponenter som inte representerar självständiga fönster ritas normalt enligt en förvald Look & Feel. Komponenterna har ett speciellt utseende, ett Javautseende, som inte beror på det underliggande systemet. Detta innebär att det förvalda beteendet i Java är att självständiga fönster har det utseende som är normalt på den aktuella plattformen, och att de andra Swing-komponenterna har ett plattformsoberoende utseende (se bilden nedan).

En plattformsberoende Look & Feel kan väljas även för de komponenter som inte representerar självständiga fönster (se bilden nedan). En Look & Feel anges med (den statiska) metoden `setLookAndFeel` i klassen `javax.swing.UIManager`. Metoden anropas innan någon Swing-komponent skapas:

 © FADIL GALJIĆ

```
try
{
    String    utseende =
                  UIManager.getSystemLookAndFeelClassName ();
    UIManager.setLookAndFeel (utseende);
}
catch (Exception e)
{
    e.printStackTrace ();
}
```

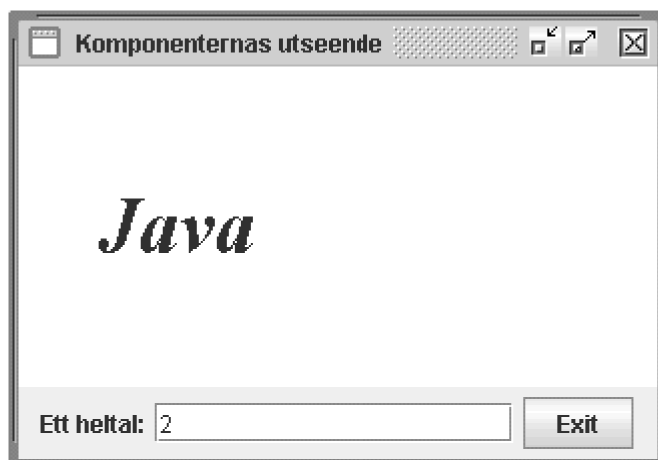
Här erhålls namnet på den Look & Feel som används på den aktuella plattformen. Sedan anges denna Look & Feel med metoden `setLookAndFeel`. Eftersom denna metod kan kasta olika kontrollerade undantag, placeras anropet till metoden i ett `try`-block.

En plattformsberoende Look & Feel kan väljas för alla komponenter i ett grafiskt användargränssnitt. Det går även att göra tvärtom: en plattformsoberoende Look & Feel (Java Look & Feel) kan väljas för alla komponenter i ett grafiskt användargränssnitt, inklusive olika ramar (se bilden nedan) och dialoger. Ett Javautseende för de olika ramarna i ett program väljs så här:

```
JFrame.setDefaultLookAndFeelDecorated (true);
```

Det här anropet placeras före den kod som skapar en ram i programmet. Ett liknande sätt används för att skapa plattformsoberoende dialoger i ett program (man skriver bara `JDialog` istället för `JFrame`).

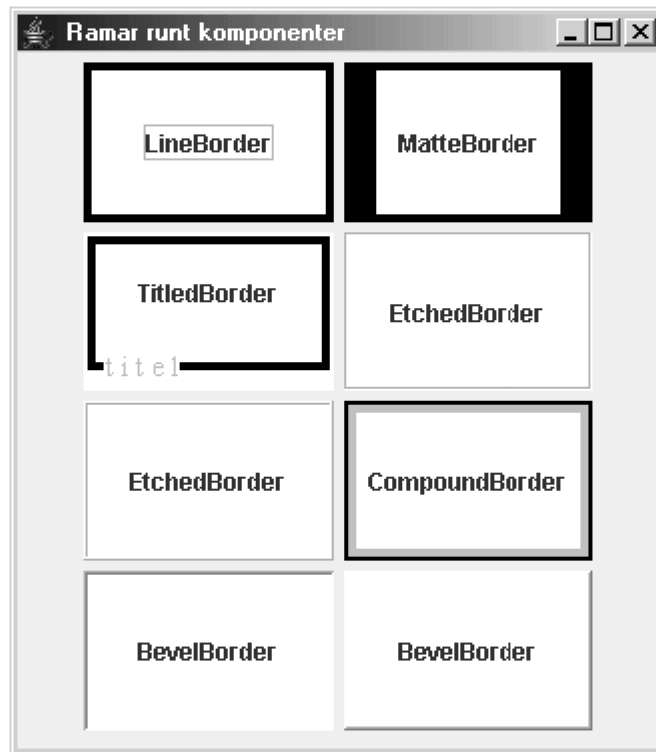
© FADIL GALJIĆ



Ramar runt komponenter

Klassen `JComponent` definierar metoden `setBorder`, som gör det möjligt att rama in en lättviktig Swing-komponent. Metoden tar ett argument av typen `javax.swing.border.Border`, som representerar en konkret ram. `Border` är ett gränssnitt, som först implementeras av den abstrakta klassen `javax.swing.border.AbstractBorder`. Klassen `AbstractBorder` är en gemensam superklass för flera andra klasser som representerar olika typer av ramar. Bland subklasserna till klassen `AbstractBorder` kan nämnas klasserna `EmptyBorder`, `LineBorder`, `MatteBorder`, `TitledBorder`, `EtchedBorder`, `CompoundBorder` och `BevelBorder` (se bilden nedan). Alla dessa klasser finns i paketet `javax.swing.border`.

 © FADIL GALJIĆ



Man kan skapa en knapp och en ram, och placera ramen runt knappen, så här:

```
JButton    knapp = new JButton ();
knapp.setText ("LineBorder");
Border    ram = new LineBorder (Color.BLACK, 4);
knapp.setBorder (ram);
```

Här skapas först en knapp med texten `LineBorder`. Sedan skapas en ram som består av en svart linje som är `4` pixlar tjock. Slutligen placeras ramen runt knappen.

För att skapa en ram med olika tjocka sidor, använder man en ram av typen `MatteBorder`. En sådan ram kan skapas så här:

```
Border    ram = new MatteBorder (4, 16, 4, 16, Color.BLACK);
```

Tjockleken för varje sida anges, och sidornas gemensamma färg. En ikon kan anges istället för färgen. Ramen dekoreras i så fall med denna ikon.

© FADIL GALJIĆ

En ram kan ha en text. En sådan ram representeras med ett objekt av typen `TitledBorder`. En ram med texten `titel` kan skapas så här:

```
Border    ram = new TitledBorder (
                      new LineBorder (Color.BLACK, 4),
                      "t i t e l",
                      TitledBorder.LEFT,
                      TitledBorder.BOTTOM,
                      new Font ("Serif", Font.PLAIN, 14),
                      Color.LIGHT_GRAY);
```

Ramens typ anges (`LineBorder`), liksom ramens text (`titel`), textens justering (`LEFT`), textens position (`BOTTOM`), font för texten och textens färg. Man behöver inte ange alla dessa argument. För att precisera textens justering i ramen används en av konstanterna `LEFT`, `CENTER` eller `RIGHT` (i klassen `TitledBorder`). Textens vertikala position preciseras med en av de följande konstanterna (i klassen `TitledBorder`): `ABOVE_TOP`, `TOP`, `BELOW_TOP`, `ABOVE_BOTTOM`, `BOTTOM` eller `BELLOW_BOTTOM`.

Det går att skapa en ram med nedsänkta sidor, eller en ram med upphöjda sidor. En sådan ram representeras med ett objekt av typen `EtchedBorder`. Konstanten `RAISED` eller konstanten `LOWERED` (båda konstanterna tillhör klassen `EtchedBorder`) anges som argument till klassens konstruktor.

Två ramar kan kombineras till en sammansatt ram. En sammansatt ram representeras med ett objekt av typen `CompoundBorder`. Den inre ramen anges som första argument, och den yttre ramen som andra argument när ett objekt av typen `CompoundBorder` skapas.

Med en ram av typen `BevelBorder` kan hela komponenten sänkas ned eller höjas upp. Man väljer en av dessa två möjligheter genom att tillföra en lämplig konstant (i klassen `BevelBorder`) som argument till klassens konstruktor. Någon av konstanterna `LOWERED` eller `RAISED` kan väljas.

Istället för att använda konstruktorer för att skapa olika ramar, kan man använda klassen `javax.swing.BorderFactory` och dess statiska metoder. Denna klass innehåller metoderna `createLineBorder`, `createMatteBorder`, `createEtchedBorder` och så vidare. Med hjälp av denna klass kan en ram skapas på följande vis:

```
Border    ram = BorderFactory.createLineBorder (Color.BLACK, 4);
```


© FADIL GALJIĆ

Ordna komponenter i en behållare

En layouthanterare

När man skapar ett grafiskt användargränssnitt, placerar man olika grafiska komponenter i en behållare. Det finns olika komponenter som kan användas som behållare, men vanligtvis använder man en panel. Komponenterna placeras i en panel, och sedan placeras panelen i ett fönster (som också är en behållare). För att ett passande användargränssnitt ska kunna skapas, måste de olika komponenterna kunna ordnas på ett vettigt sätt i behållaren. Komponenternas placering och storlek måste kunna påverkas. En lämplig *layout* måste på något sätt kunna åstadkommas.

För att underlätta layoutspecifikationen har det i Java utvecklats ett antal klasser som sköter denna uppgift. Alla dessa klasser implementerar gränssnittet `java.awt.LayoutManager`. Därför kallas objekt av dessa klasser för *layouthanterare*. De klasser som används mest är `FlowLayout`, `BorderLayout`, `GridLayout` och `GridBagLayout`. Vissa layoutklasser implementerar ett mer speciellt gränssnitt, som heter `java.awt.LayoutManager2`. Detta gränssnitt är ett subgränssnitt till gränssnittet `LayoutManager`.

En layouthanterare är ett objekt som ordnar olika komponenter i en behållare på ett speciellt sätt. En layouthanterare av en viss typ har en speciell strategi, som den använder för att ordna olika komponenter. En layouthanterare av typen `FlowLayout`, till exempel, ordnar komponenterna i en följd, en layouthanterare av typen `GridLayout` ordnar komponenterna i en matris, och så vidare. Förutom en fördefinierad strategi, kan en layouthanterare även behöva olika sorters information om grafiska komponenter i användargränssnittet. Denna information erhålls genom att olika metoder anropas i samband med komponenterna. Alla komponenter har till exempel metoderna `getMinimumSize`, `getMaximumSize` och `getPreferredSize`, samt metoderna `setMinimumSize`, `setMaximumSize` och `setPreferredSize`. Dessa metoder definieras i klassen `java.awt.Component`, som är en superklass till alla grafiska komponenter. Metoderna hanterar en komponents minsta, största och önskade storlek. En layouthanterare kan anropa de angivna `get`-metoderna för att få en bättre uppfattning om en komponents storlek. På samma sätt kan en layouthanterare anropa någon av metoderna `getAlignmentX` och `getAlignmentY`, som definieras i klassen `Component`. Dessa metoder och motsvarande `set`-metoder hanterar en komponents justering på den plats som tilldelats komponenten. En kom-

© FADIL GALJIĆ

ponent kan justeras uppåt, nedåt, i mitten, till vänster, till höger, och på andra sätt.

En behållare har en fördefinierad layouthanterare. En panel, till exempel, har en fördefinierad layouthanterare av typen `FlowLayout`. Därför placeras komponenterna i en panel i en följd. Men detta förvalda beteende kan ändras, och en godtycklig layouthanterare kan installeras. En layouthanterare installeras i en behållare med metoden `setLayout` i klassen `java.awt.Container`. Om `panel` är en referens till en panel, kan detta göras så här:

```
BorderLayout    layout = new BorderLayout ();
panel.setLayout (layout);
```

Panelens fördefinierade layouthanterare sätts ur spel, och istället väljs en layouthanterare av typen `BorderLayout`. Denna layouthanterare placerar komponenterna enligt väderstreck.

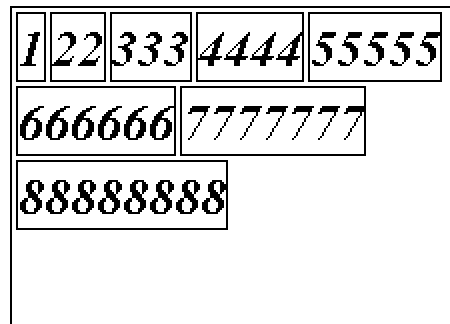
När en behållares layouthanterare har angivits, behöver man inte aktivera layouthanteraren på något sätt. Den gör sitt jobb i bakgrunden varje gång behållaren visas. En layouthanterare ordnar olika komponenter i en behållare enligt den strategi som är inbyggd i den.

Vanliga layoutstrategier

Ordna komponenter i en följd

En layouthanterare av typen `FlowLayout` ordnar komponenter i en behållare i en följd (se bilden nedan). Komponenterna placeras ut från vänster till höger. När en rad är full, fortsätter detta på nästa rad. Komponenter i en rad kan centreras, vänsterjusteras eller högerjusteras. Man kan också ange avståndet mellan två intilliggande komponenter i en rad, och avståndet mellan två rader.

© FADIL GALJIĆ



En panel, dess komponenter och layouthanterare kan definieras så här:

```
class FPanel extends JPanel
{
    public FPanel ()
    {
        JButton[]    knappar = new JButton[8];
        for (int i = 0; i < knappar.length; i++)
            knappar[i] = new JButton ();

        FlowLayout    layout =
                       new FlowLayout (FlowLayout.LEFT, 2, 2);
        this.setLayout (layout);

        for (int i = 0; i < knappar.length; i++)
            this.add (knappar[i]);
    }
}
```

Först skapas ett antal knappar. Därefter skapas en layouthanterare av typen `FlowLayout`. I koden specificeras att knapparna ska justeras till vänster, och det horisontella och vertikala avståndet mellan komponenterna anges (till `2` pixlar). Layouthanteraren installeras i panelen, och därefter placeras knapparna i panelen. Layouthanteraren bestämmer knapparnas positioner och storlekar.

Klassen `FlowLayout` har flera konstruktorer, så att ett objekt av klassen kan skapas på olika sätt. Man kan till exempel utelämna de argument som representerar avstånden mellan komponenterna. Detta medför att komponenterna placeras intill varandra. Det första argumentet anger komponenternas justering i en rad. Justeringen bestäms med en lämplig konstant i klassen `FlowLayout`. Man kan välja mellan konstanterna `LEFT`, `CENTER` och `RIGHT`. Om en layouthanterare av typen `FlowLayout` skapas

utan att något argument anges, kommer komponenterna att centreras i raden.

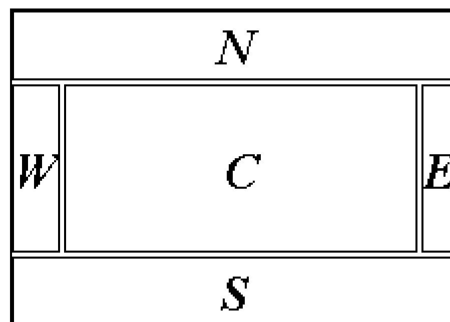
En layouthanterare av typen `FlowLayout` kontaktar de enskilda komponenterna för att bestämma deras storlekar. När det gäller en knapp, hämtas information om knappens text och den använda fonten. Utifrån dessaa uppgifter bestäms knappens storlek. Man kan påverka komponenternas dimensioner i behållaren genom att ange önskade dimensioner för dem (med metoden `setPreferredSize`).

Om en behållares storlek ändras (genom att till exempel storleken på det motsvarande fönstret ökas eller minskas), omplacerar behållarens layouthanterare de olika komponenterna. Antalet rader med komponenter kan i så fall öka eller minska. Ändringen påverkar inte de enskilda komponenternas storlekar.

En panel har en fördefinierad layouthanterare av typen `FlowLayout`.

Ordna komponenter efter väderstreck

En layouthanterare av typen `BorderLayout` placerar komponenterna i en behållare efter väderstreck. Komponenterna placeras längs behållarens fyra kanter och i dess mittersta del. Dessa positioner definieras som nord, öst, syd, väst och center (se bilden nedan). Alla fem positionerna behöver inte användas. Om en av positionerna inte används, upptas motsvarande utrymme av komponenterna på andra positioner.



En layouthanterare av typen `BorderLayout` kan skapas så här:

```
BorderLayout    layout = new BorderLayout (2, 2);
```


© FADIL GALJIĆ

Det horisontella och vertikala avståndet mellan komponenterna anges som argument till konstruktorn. Om dessa argument utelämnas, kommer komponenterna att ligga intill varandra.

En layouthanterare av typen `BorderLayout` kräver att man anger på vilken position en enskild komponent ska placeras. En av konstanterna (i klassen `BorderLayout`) `NORTH`, `EAST`, `SOUTH`, `WEST` eller `CENTER` anges som andra argument till metoden `add`. Så här kan en knapp (som refereras av referensen `knapp`) placeras i en panels (panelen refereras av referensen `panel`) övre (norra) del:

```
panel.add (knapp, JBorderLayout.NORTH);
```

Istället för konstanterna `NORTH`, `EAST`, `SOUTH`, `WEST` eller `CENTER`, kan strängarna `North`, `East`, `South`, `West` eller `Center` användas. Man kan till exempel skriva så här:

```
panel.add (knapp, "North");
```

En layouthanterare av typen `BorderLayout` kontaktar de enskilda komponenterna för att bestämma deras storlekar. När det gäller en knapp, erhålls information om knappens text och den font som används. Utifrån detta bestäms knappens storlek. Den norra komponenten tar upp hela utrymmet från behållarens vänstra sida till dess högra sida. Det är bara komponentens höjd som behöver anpassas. Det samma gäller för den södra komponenten. Komponenterna i behållarens vänstra och högra delar tar vertikalt upp den plats som återstår när den övre och den nedre komponenten har placerats ut. Det är bara komponenternas bredd som kan anpassas. Det utrymme som återstår när kantkomponenterna placerats ut, tas upp av den mittersta komponenten.

När en behållares storlek ändras, ändras även de enskilda komponenternas storlekar. Komponenterna anpassas så att hela behållaren åter fylls. Bredden på den övre, den mittersta och den nedre komponenten ändras. Höjden för den vänstra, den mittersta och den högra komponenten ändras också.

Ett fönster (till exempel en ram av typen `JFrame`) har en fördefinierad layouthanterare av typen `BorderLayout`.

Ordna komponenter i en matris

En layouthanterare av typen `GridLayout` ordnar olika komponenter i en behållare i en matris. Behållarens yta delas upp i ett antal rader och kolumner. På så sätt skapas ett antal lika stora fält. Om en behållare till ex-

empel delas i två rader och tre kolumner, så skapas sex (2 x 3) fält. Den första komponenten placeras i det första fältet, nästa komponent i det andra fältet i samma rad, och så vidare. När en rad är full, fortsätter placeringen i nästa rad (se bilden nedan). Alla fält behöver inte användas. En komponent tar upp ett fälts hela utrymme. Alla komponenter är därför lika stora. Om en behållares storlek ändras, ändras också de enskilda fältens storlek och därmed komponenternas storlek.

11	12	13	14
21	22	23	24
31	32	33	34

En layouthanterare av typen `GridLayout` kan skapas så här:

```
GridLayout    layout = new GridLayout (3, 4, 2, 2);
```

Behållaren delas i `3` rader och `4` kolumner. Avståndet mellan kolumner anges med det tredje argumentet, och avståndet mellan rader med det fjärde argumentet. Om dessa argument utelämnas, blir motsvarande avstånd `0`.

Kombinera olika layoutstrategier

För att åstadkomma en passande layout, kombinerar man olika strategier. Man kan skapa flera behållare, och kombinera dessa behållare i en större enhet. Varje behållare har sin egen layouthanterare. De olika behållarna kan ha olika typer av layouthanterare.

En panel med flera andra komponenter kan definieras så här:

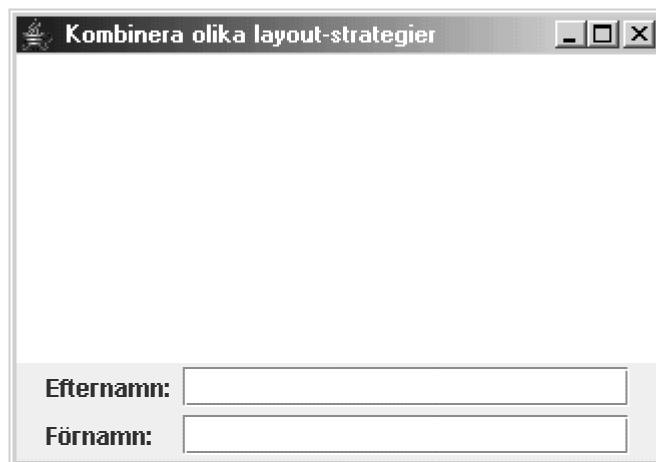
```
class FPanel extends JPanel
{
    public FPanel ()
    {
        this.setBackground (Color.WHITE);

        JLabel   etikett1 = new JLabel ("Efternamn:");
        JTextField    falt1 = new JTextField (20);
```


© FADIL GALJIĆ

```
        JPanel    panel1 = new JPanel ();
        panel1.setLayout (
                   new FlowLayout (FlowLayout.CENTER, 6, 2));
        panel1.add (etikett1);
        panel1.add (falt1);

        JLabel   etikett2 = new JLabel ("Förnamn:");
        JTextField    falt2 = new JTextField (20);
        JPanel    panel2 = new JPanel ();
        panel2.setLayout (
                  new FlowLayout (FlowLayout.CENTER, 15, 2));
        panel2.add (etikett2);
        panel2.add (falt2);

        JPanel    panel = new JPanel ();
        panel.setLayout (new GridLayout (2, 1));
        panel.add (panel1);
        panel.add (panel2);

        JTextArea    textArea = new JTextArea (10, 20);

        this.setLayout (new BorderLayout ());
        this.add (textArea, "Center");
        this.add (panel, "South");
    }
}
```

Här skapas först en etikett och ett textfält, och dessa komponenter lagras i en panel. I panelen används en layouthanterare av typen `FlowLayout`. Sedan skapas en etikett och ett textfält till, och dessa lagras i en ny panel. Även i denna panel används en layouthanterare av typen `FlowLayout`. De två panelerna grupperas därefter i en ny panel. I den nya panelen används en layouthanterare av typen `GridLayout` (`2` rader och 1 kolumn). Sedan skapas en textarea, som tillsammans med panelen (som innehåller två delpaneler) placeras i huvudpanelen (`this`-panelen, den panel som definieras i klassen `FPanel`). Huvudpanelen har en layouthanterare av typen `BorderLayout`. Den panel som innehåller de två delpanelerna lagras i den nedre delen av huvudpanelen, och textarean i mitten (se bilden nedan). Genom att gruppera olika komponenter via flera delpaneler, och genom att använda layouthanterare av olika typer, lyckas man placera ut olika komponenter på önskat sätt.

© FADIL GALJIĆ

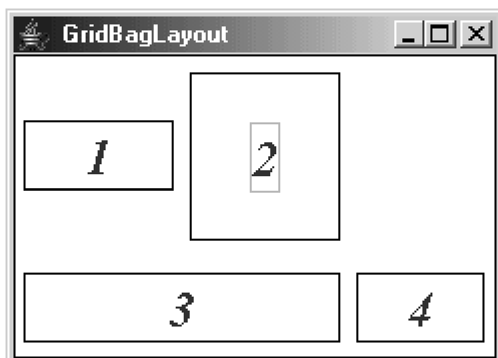


En flexibel layoutstrategi

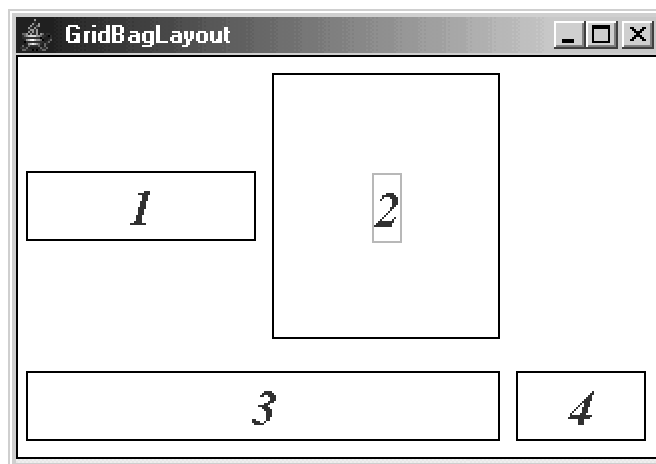
En layouthanterare av typen `GridBagLayout` använder en flexibel layoutstrategi, som är öppen för påverkan utifrån. Genom att använda en layouthanterare av denna typ, har man möjlighet att i stor utsträckning påverka komponenternas beteende vid ändring av behållarens storlek. För varje komponent som placeras i behållaren, anger man ett objekt som specificerar komponentens position, storlek och beteende vid en ändring av behållarens storlek. Man använder ett objekt av typen `java.awt.GridBagConstraints`, och justerar de olika (publika) instansvariablerna i detta objekt. Dessa variabler beskriver en komponents olika layoutegenskaper. En layouthanterare av typen `GridBagLayout` analyserar olika objekt av typen `GridBagConstraints`, och bestämmer utifrån denna analys hur de enskilda komponenterna ska placeras.

Den nedanstående bilden visar en panel med ett antal knappar utplacerade.

 © FADIL GALJIĆ



Om panelens storlek ökas i horisontell riktning, ökas storleken på samtliga knappar utom knapp 4 proportionellt. Storleken på knapp 4 ändras inte. Om panelens storlek ökas i vertikal riktning, ökas storleken på knapparna 1 och 2 proportionellt, men knapparna 3 och 4 förändras inte. Den nedanstående bilden visar panelen efter ökningen.



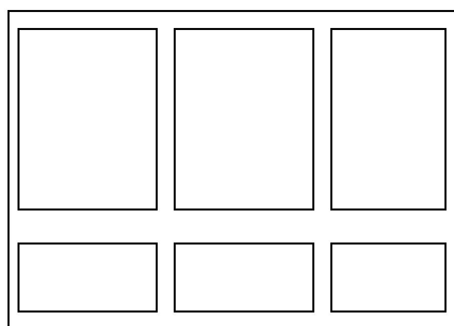
För att implementera denna layout används en layouthanterare av typen `GridBagLayout`. Så här kan detta göras:

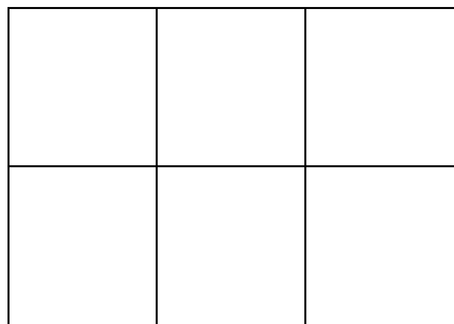
```
GridBagLayout    layout = new GridBagLayout ();
this.setLayout (layout);
```

Man skapar också knapparna och bestämmer deras utseenden. Knapparna kan refereras av referenserna `knapp1`, `knapp2`, `knapp3` och `knapp4`. Därefter bestämmer man knapparnas positioner, och placerar dem i panelen.

© FADIL GALJIĆ

För att kunna placera knapparna, observerar man den önskade layouten och försöker föreställa sig en matrisstruktur (en *prototypmatris*) i denna layout. Man försöker föreställa sig ett antal rader och kolumner. Olika rader kan ha olika höjder, och olika kolumner kan ha olika bredder. Man kan komma fram till den strukturen som visas på de följande bilderna. Strukturen är i grunden en matris med två rader och tre kolumner. De enskilda knapparnas position och storlek måste specificeras med en sådan matris som utgångspunkt.





För att specificera position, storlek och beteende för den första knappen, skapar man ett objekt av typen `GridBagConstraints` och justerar olika instansvariabler i detta objekt. Variablerna har förvalda värden, som kan accepteras. Endast de instansvariabler som inte ska behålla sina förvalda värden justeras.

De olika layoutegenskaperna för den första knappen kan beskrivas så här:

```
GridBagConstraints    con = new GridBagConstraints ();

con.gridx = 0;
con.gridy = 0;
con.gridwidth = 1;
```

 © FADIL GALJIĆ

```
con.gridheight = 1;

con.insets = new Insets (8, 4, 8, 4);
con.fill = GridBagConstraints.HORIZONTAL;
con.anchor = GridBagConstraints.CENTER;

con.ipadx = 0;
con.ipady = 0;

con.weightx = 1;
con.weighty = 1;
```

Den första knappen relateras till den matris som representerar layoutens struktur. Med variablerna `gridx` och `gridy` anges var i matrisen som knappen ska börja. Knappens övre vänstra hörn ska ligga i första raden (rad nummer `0`, rader och kolumner räknas från 0) och första kolumnen (kolumn nummer `0`). Variablerna `gridwidth` och `gridheight` bestämmer knappens storlek. En komponent kan sträcka sig över flera rader och kolumner. Den första knappen ska bara ta upp ett fält i prototypmatrisen. Den ska sträcka sig över en kolumn (`gridwidth = 1`, som förvalt värde) och över en rad (`gridheight = 1`, som förvalt värde).

Med variablerna `gridx`, `gridy`, `gridwidth` och `gridheight` definieras det område i prototypmatrisen där en komponent ska placeras. Detta område kan omfatta ett eller flera fält. Men även komponentens position i området måste preciseras. För detta ändamål används instansvariablerna `insets`, `fill` och `anchor`.

En komponent placeras i ett område som består av ett eller flera fält. Variabeln `insets` bestämmer komponentens avstånd från detta områdes gränser. Man anger hur stor utfyllnaden ska vara ovanför komponenten, till vänster om komponenten, under komponenten och till höger om komponenten. Dessa värden representeras (i pixlar) via ett objekt av klassen `java.awt.Insets`. Så här kan värdena anges:

```
con.insets = new Insets (8, 4, 8, 4);
```

Genom att placera utfyllnad kring en komponent, säkerställer man att det finns ett avstånd mellan komponenten och dess grannar. Om inte variabeln `insets` anges, blir alla dessa avstånd `0` (de förvalda värdena är `0`).

Med instansvariabeln `fill` kan man precisera om en komponent ska fylla hela (det tilldelade) området. Variabeln kan tilldelas en av de följande konstanterna (i klassen `GridBagConstraints`): `NONE` (förvalt värde), `HORIZONTAL`, `VERTICAL` och `BOTH`. Om `NONE` väljs, så ritas komponenten med sin förvalda storlek. En knapp, till exempel, ritas så stor att motsvarande text får plats. Om `HORIZONTAL` väljs, sträcker sig komponenten längs det tillde-

© FADIL GALJIĆ

lade området i horisontell riktning. Det förvalda värdet för komponentens höjd används. Om `VERTICAL` väljs som värde för instansvariabeln `fill`, sträcker sig komponenten vertikalt (men inte horisontellt) över det tilldelade området. Om `BOTH` väljs, sträcker sig komponenten över hela området (minskat med de värden som anges med variabeln `insets`).

Om en komponent inte fyller hela området, måste man precisera var i området komponenten ska placeras. Om man till exempel väljer värdet `HORIZONTAL` för instansvariabeln `fill`, kan den motsvarande komponenten placeras i den övre, mittersta eller nedre delen i det tilldelade området. Placeringen anges med variabeln `anchor`. Möjliga värden för denna variabel är: `NORTH`, `EAST`, `SOUTH`, `WEST`, `CENTER` (förvalt värde), `NORTHEAST`, `SOUTHEAST` och `SOUTHWEST` (konstanterna finns i klassen `GridBagConstraints`).

Med variablerna `gridx`, `gridy`, `gridwidth` och `gridheight` definieras det område i prototypmatrisen där en komponent ska placeras. Med variablerna `insets`, `fill` och `anchor` definieras hur komponenten ska placeras i området. Om komponenten blir för liten, kan man utöka dess storlek med ett antal pixlar. För detta ändamål används variablerna `ipadx` och `ipady`:

```
con.ipadx = 12;
con.ipady = 5;
```

Komponentens storlek ökas med `12` pixlar i horisontell riktning och `5` pixlar i vertikal riktning.

Instansvariablerna `gridx`, `gridy`, `gridwidth`, `gridheight`, `insets`, `fill`, `anchor`, `ipadx` och `ipady` bestämmer en komponents position och storlek. Men det räcker inte. Komponentens beteende vid en ändring av behållarens storlek måste också definieras. Man måste precisera om komponenten ska ändras i detta fall, och i vilken utsträckning. För detta ändamål används instansvariablerna `weightx` och `weighty`. Med dessa variabler definieras en komponents vikt. Det förvalda värdet för dessa variabler är `0`, vilket innebär att motsvarande komponent inte ska ändras vid en ändring av behållarens storlek. Värden mellan `0` och `1` (eller från ett annat intervall) kan anges för olika komponenter, och på så sätt kan komponenternas beteende vid en ändring av behållarens storlek definieras.

Det är inte en enskild komponent som ändras vid ändringen av behållarens storlek. Bredden för en hel kolumn och höjden för en hel rad ändras. Radens vikt bestäms utifrån de komponenter som finns i raden. Vikten är lika med den största komponentvikt i raden. På samma sätt bestäms en kolumns vikt. Genom att ge alla vikter i en rad eller kolumn värdet `0`, säkerställer man att radens höjd eller kolumnens bredd inte ändras vid en

 © FADIL GALJIĆ

ändring av behållarens storlek. Ju större vikt en rad eller kolumn har, desto mer ändras raden eller kolumnen.

Ett objekt av typen `GridBagConstraints` beskriver en komponents layoutegenskaper. När objektet väl har skapats och justerats, kan den motsvarande komponenten placeras i behållaren. För detta används metoden `add`, och som argument anges den komponent som ska placeras och det objekt som beskriver hur komponenten ska placeras. Knappen `knapp1` kan placeras i den panel som definieras (`this`-panelen) på följande vis:

```
this.add (knapp1, con);
```

När den första knappen har placerats, kan de olika layoutegenskaperna för den andra knappen definieras, och även den placeras i panelen. Detta upprepas för samtliga komponenter. Ett nytt objekt av typen `GridBagConstraints` kan skapas för varje komponent. På så sätt utgår man ifrån de förvalda värdena, och ändrar bara det som man vill ändra.

Knappen `knapp4` kan placeras i panelen så här:

```
con = new GridBagConstraints ();

con.gridx = 2;
con.gridy = 0;
con.gridwidth = 1;
con.gridheight = 2;

con.insets = new Insets (8, 4, 8, 4);
con.fill = GridBagConstraints.NONE;
con.anchor = GridBagConstraints.SOUTH;

con.ipadx = 50;
con.ipady = 0;

con.weightx = 0;
con.weighty = 0;

this.add (knapp4, con);
```

Knappen `knapp4` tilldelas det området som omfattar hela den tredje kolumnen (båda fälten). Knappen placeras i områdets nedre del. Eftersom knappen ritas i den förvalda storleken (värdet `NONE` väljs för variabeln `fill`), blir knappen för liten. Därför utökas knappens storlek med `50` pixlar i horisontell riktning.

Det intressanta med `knapp4` är att dess vikter sätts till `0` (förvalda värden, de motsvarande satserna kan utelämnas). Eftersom det inte finns några andra komponenter i den tredje kolumnen, blir vikten av denna kolumn `0`. Det innebär att kolumnens bredd inte ändras vid en ändring av behål-

larens storlek. I andra raden finns knappen 3 och knappen 4. Om `y`-vikten sätts till `0` även för knappen 3, får den andra raden vikten 0. I så fall ändras inte radens höjd när behållarens storlek ändras.

Information av olika slag tillförs till en layouthanterare av typen `GridBagLayout` via motsvarande objekt av typen `GridBagConstraints`. Först ritar man på ett papper den önskade layouten. Därefter identifieras (och ritas) en prototypmatris, som beskriver layoutens grundläggande struktur. Denna består av ett antal rader och kolumner, som definierar ett antal fält. Raderna kan ha olika höjder och kolumnerna olika bredder. Prototypmatrisen används för att beskriva de enskilda komponenternas layoutegenskaper. Man skapar aldrig prototypmatrisen i sitt program. En layouthanterare av typen `GridBagLayout` förstår hur prototypmatrisen ser ut utifrån de beskrivningar som tillförs via olika objekt av typen `GridBagConstraints`. Layouthanteraren analyserar de uppgifter som den får via dessa objekt, identifierar prototypmatrisen och placerar komponenterna på deras rätta platser.

Layoutkomponenter

Behållare som ordnar sina komponenter på ett speciellt sätt

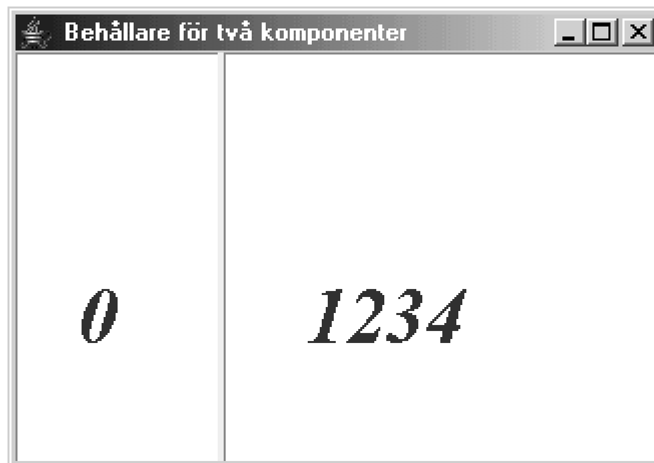
Komponenter kan ordnas i en behållare på olika sätt. För det ändamålet skapas en lämplig layouthanterare som installeras i behållaren. Den layouthanteraren ordnar sedan komponenterna enligt en inbyggd strategi. Men det finns situationer när en annan strategi kan användas. En speciell behållare, som automatiskt ordnar sina komponenter på ett förvalt sätt, kan användas. Det finns till exempel sådana behållare i standardbiblioteket som ordnar sina komponenter bakom varandra. Man kan bläddra genom dessa komponenter och granska en komponent i taget. Det finns även andra typer av behållare som kan användas för att åstadkomma vissa typiska layouter.

Behållare för två komponenter

En behållare av typen `javax.swing.JSplitPane` kan användas för att placera två komponenter bredvid varandra, med en avgränsningslinje emellan (se bilden nedan). En vertikal avgränsningslinje (komponenterna placeras i en rad) eller en horisontell avgränsningslinje (komponenterna

 © FADIL GALJIĆ

placeras i en kolumn) kan användas. Den avgränsande linjen kan dras, och på så sätt kan man välja hur behållarens area ska fördelas mellan de två komponenterna.



På följande vis kan man skapa två komponenter och en behållare av typen `JSplitPane`, och placera komponenterna i behållaren:

```
JTextArea    ta1 = new JTextArea (12, 10);
ta1.append ("\n\n   0");
JTextArea    ta2 = new JTextArea (12, 10);
ta2.append ("\n\n    1234");

JSplitPane    splitPane =
   new JSplitPane (JSplitPane.HORIZONTAL_SPLIT, ta1, ta2);
splitPane.setContinuousLayout (true);
```

Två textareor skapas, och en behållare av typen `JSplitPane` för dessa textareor. En vertikal avgränsningslinje används, och komponenterna placeras bredvid varandra i en rad (`HORIZONTAL_SPLIT`, komponenterna ordnas i horisontell riktning). Sedan anropas metoden `setContinuousLayout`, för att få en ständig uppdatering vid avgränsningslinjens förflyttning.

Avgränsningslinjens tjocklek kan anges med metoden `setDividerSize` (linjens tjocklek anges som argument i antal pixlar). Avgränsningslinjens placering kan anges med metoden `setDividerLocation` (linjens placering anges som argument i antal pixlar).

En behållare av typen `JSplitPane` innehåller exakt två komponenter. Dessa komponenter kan vara behållare som innehåller ett antal andra komponenter. Man kan till exempel gruppera flera komponenter i en

© FADIL GALJIĆ

panel, och placera denna panel som en komponent i en behållare av typen `JSplitPane`.

För att visa en behållare av typen `JSplitPane`, placerar man behållaren i ett fönster. Den kan placeras direkt i ett fönster, eller via en annan behållare (till exempel via en panel).

Visa en för stor komponent

En komponent (eller en uppsättning komponenter i en behållare) kan vara för stor för det utrymme som den tilldelas. I detta fall syns bara en del av komponenten. För att även resten av komponenten ska kunna granskas, placeras komponenten i en behållare av typen `javax.swing.JScrollPane`. En sådan behållare har en rullningslist, som gör det möjligt att rulla fram komponenten och granska en del av den i taget (se bilden nedan).

Man anger den komponent som ska finnas i en behållare av typen `JScrollPane` när behållaren skapas (komponenten kan även anges i efterhand, med metoden `setViewportView`). Så här kan detta ske:

```
JTextArea    textArea = new JTextArea (12, 10);
JScrollPane    scrollPane = new JScrollPane (textArea);
```

Här skapas en textarea och en behållare av typen `JScrollPane`, och textarean placeras i behållaren. Om textarean är större än det tilldelade utrymmet, skapas automatiskt motsvarande rullningslist. Behållaren kan ha en rullningslist (vertikal eller horisontell), eller två rullningslister (både

© FADIL GALJIĆ

vertikal och horisontell). Om komponenten i en behållare av typen `JScrollPane` är synlig i sin helhet, skapas inga rullningslister (även i detta fall kan man framtvinga att rullningslister skapas genom att använda en speciell konstruktor).

Endast en komponent placeras i en behållare av typen `JScrollPane`. Om man vill ha flera komponenter i en sådan behållare, placerar man komponenterna i en panel (eller en annan behållare) som sedan placeras i behållaren.

För att visa en behållare av typen `JScrollPane`, placerar man behållaren i ett fönster (till exempel en ram). Behållaren kan placeras direkt i ett fönster, eller via en annan behållare (till exempel via en panel).

Bläddra mellan olika komponenter.

Flera komponenter kan lagras bakom varandra i en behållare. För detta ändamål används en behållare av typen `javax.swing.JTabbedPane`. En komponent lagras tillsammans med sitt namn. Komponenternas namn visas på en eller flera rader överst i behållaren (namnen kan även placeras på någon annan plats). När ett namn väljs, visas motsvarande komponent i behållaren. På så sätt kan man bläddra mellan olika komponenter (se bilden nedan).

Man kan skapa ett antal komponenter och en behållare av typen `JTabbedPane`, och placera komponenterna i behållaren, på följande vis:

```
String[]    arsTider = {"v i n t e r n", "v å r e n",
```

```
                        "s o m m a r e n", "h ö s t e n"};
JTextArea[]    ta = new JTextArea [4];
for (int i = 0; i < ta.length; i++)
{
    ta[i] = new JTextArea (12, 10);
    ta[i].append ("\n  " + arsTider[i]);
}

String[]    namn = {"ett", "två", "tre", "fyra"};

JTabbedPane    tabbedPane = new JTabbedPane ();
for (int i = 0; i < ta.length; i++)
    tabbedPane.addTab (namn[i], ta[i]);
```

Här skapas fyra textareor som innehåller namnen på årstiderna. Därefter skapas en vektor med namn på textareorna. Slutligen skapas en behållare av typen `JTabbedPane`, och textareorna placeras i denna behållare. Till detta används metoden `addTab`, med komponentens namn och komponenten som argument. Komponenternas namn visas överst i behållaren, och komponenterna placeras bakom varandra. Man väljer den komponent som ska visas i behållaren genom att välja motsvarande namn.

En behållare av typen `JTabbedPane` visar en komponent i taget. Denna komponent kan vara en annan behållare (till exempel en panel), som innehåller ett antal andra komponenter.

Ett argument kan anges (till motsvarande konstruktor) när en behållare av typen `JTabbedPane` skapas. En konstant i klassen `JTabbedPane` tillförs, som preciserar var i behållaren namnen på de olika komponenterna ska visas. En av de följande konstanterna kan väljas: `TOP`, `BOTTOM`, `LEFT` och `RIGHT`. Om inte något argument anges, visas motsvarande namn överst i behållaren.

Tillsammans med ett namn, kan även en ikon som symboliserar motsvarande komponent användas. Ikonen anges som argument till metoden `addTab`:

```
tabbedPane.addTab (namn, new ImageIcon ("bild.gif"), komponent))
```

Om man bara vill använda en ikon, sätts motsvarande namn till en tom sträng. Även ett fjärde argument kan anges till metoden `addTab`, och detta anger den hjälptext som visas när musen placeras över motsvarande flik.

Det kan hända att det finns så många komponenter i en behållare av typen `JTabbedPane` att komponenternas namn tar upp för mycket plats. I detta fall kan komponenternas namn ordnas i en rad, med knappar med pilar i. Via dessa knappar kan man bläddra mellan de olika namnen. Layouten med knappar väljs så här:

© FADIL GALJIĆ

```
tabbedPane.setTabLayoutPolicy (JTabbedPane.SCROLL_TAB_LAYOUT);
```

För att återgå till layouten som visar alla namn (utan knappar) gör man så här:

```
tabbedPane.setTabLayoutPolicy (JTabbedPane.WRAP_TAB_LAYOUT);
```

En behållare av typen `JTabbedPane` använder index för att komma åt sina komponenter. Index räknas som vanligt från och med `0`. Dessa index kan användas för att manipulera behållarens komponenter, och deras namn och ikoner. Det går att göra så här:

```
tabbedPane.setComponentAt (index, komponent);
tabbedPane.setTitleAt (index, namn);
tabbedPane.setIconAt (index, ikon);
tabbedPane.setBackgroundAt (index, farg);
tabbedPane.setForegroundAt (index, farg);
tabbedPane.setSelectedIndex (index);
tabbedPane.removeTabAt (index);
```

En viss komponent placeras på ett givet index (någonting måste finnas på indexet redan när metoden `setComponentAt` anropas), komponentens namn och ikon anges, och bakgrundsfärgen och förgrundsfärgen för den motsvarande fliken (där komponentens namn och/eller ikon finns). Därefter visas komponenten med metoden `setSelectedIndex`. Slutligen tas komponenten bort från behållaren. Med dessa metoder kan en behållare av typen `JTabbedPane` ändras dynamiskt.

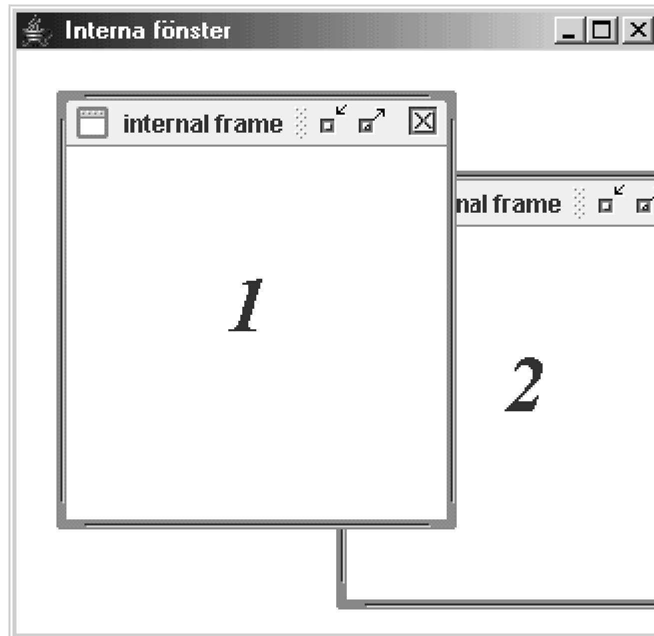
Metoden `getTabCount` ger antalet komponenter i en behållare av typen `JTabbedPane`. Man kan stega igenom alla dessa komponenter i en loop. Komponenterna kan modifieras, bytas ut mot andra komponenter, eller tas bort.

En behållare av typen `JTabbedPane` visas genom att den placeras i ett fönster. Behållaren kan placeras direkt i ett fönster, eller via en annan behållare (till exempel via en panel).

Interna fönster

En ram kan innehålla flera fönster (se bilden nedan). Dessa *interna fönster* kan flyttas med musen inuti ramen och placeras på olika ställen. Ett sådant fönster kan inte lämna sin förälderram. Ett internt fönster innehåller normalt ett antal komponenter och representerar en enhet i en applikation. Man kan till exempel fördela flera textdokument mellan olika fönster i en och samma ram.

© FADIL GALJIĆ



Ett internt fönster representeras med ett objekt av klassen `javax.swing.JInternalFrame`. Klassen `JInternalFrame` är en subklass till klassen `javax.swing.JComponent`. Det innebär att ett internt fönster är en lättviktig komponent, och inte ett fönster som kan visas oberoende av andra behållare (klassen `JInternalFrame` är inte en subklass till klassen `java.awt.Window`).

Ett internt fönster kan skapas så här:

```
JInternalFrame    iframe = new JInternalFrame (
                  " internal frame", true, true, true, true);
```

Ett internt fönster med en given titel skapas. Ett antal booleska värden anges som argument för att precisera fönstrets olika egenskaper. Det första värdet anger att fönstrets storlek ska kunna ändras. De följande värdena preciserar att fönstret ska kunna stängas, maximeras och göras till en ikon. Klassen `JInternalFrame` överlagrar konstruktorer, så att alla argumenten inte behöver anges. Dessa egenskaper kan även anges (eller ändras) med metoderna `setTitle`, `setResizable`, `setClosable`, `setMaximizable` och `setIconifiable`.

Metoderna `setSize` och `setLocation` kan användas för att ange ett fönsters storlek (den förvalda storleken är `0 x 0`) och placering (den förvalda

 © FADIL GALJIĆ

placeringen är i det övre vänstra hörnet). Det går också att använda metoden `reshape`. Som argument till denna metod anges koordinaterna för fönstrets övre vänstra hörn, fönstrets bredd och fönstrets höjd. Så här kan metoden användas:

```
iframe.reshape (160, 60, 200, 220);
```

Metoderna `getWidth` och `getHeight` ger ett fönsters dimensioner. Dessa metoder kan även användas för att bestämma dimensionerna för ett fönsters arbetsyta. Denna yta erhålls först med metoden `getContentPane`. Det går att göra så här:

```
int    arbetsYtasHojd = iframe.getContentPane ().getHeight ();
```

Ett fönsters bredd och höjd, samt höjden för motsvarande arbetsyta, kan användas för att finjustera olika fönsters placering (relativt varandra) i en och samma ram. Dessa metoder anropas bara för ett fönster när fönstret redan är på sin plats. Normalt skapas interna fönster via en meny eller en knapp. I dessa fall kan man bestämma storleken och placeringen för det fönster som skapas relativt de fönster som redan finns i ramen.

Ett internt fönster görs synligt med metoden `setVisible`:

```
iframe.setVisible (true);
```

Ett fönsters beteende vid stängningen preciseras med metoden `setDefaultCloseOperation`. Så här kan metoden användas:

```
iframe.setDefaultCloseOperation (JInternalFrame.DISPOSE_ON_CLOSE);
```

Med metoden `add` kan en komponent placeras i ett internt fönster av typen `JInternalFrame`. Man kan även ange komponentens placering i fönstret (ett internt fönster har en förvald layouthanterare av typen `java.awt.BorderLayout`). Metoden kan till exempel användas så här:

```
JButton    knapp = new JButton ("Avsluta");
iframe.add (knapp, "South");
```

Med metoden `setFrameIcon` kan ett fönsters ikon anges. Ett internt fönster kan placeras främst med metoden `moveToFront`, och längst bak med metoden `moveToBack`.

Det är någonting speciellt med interna fönster av typen `JInternalFrame`. De placeras i en behållare av typen `javax.swing.DesktopPane`. Det är en (indirekt) subklass till klassen `javax.swing.JComponent`, som är speciellt utformad för att hantera interna fönster. Man kan skapa en sådan behållare, och placera ett internt fönster i den, på följande vis:

```
JDesktopPane    desktop = new JDesktopPane ();
desktop.add (iframe);
```

Man anger sedan att behållaren av typen `JDesktopPane` ska vara en rams arbetsyta. Detta görs med metoden `setContentPane` (i klassen `JFrame`):

```
JFrame    frame = new JFrame ();
frame.setContentPane (desktop);
```

När en ram visas, visas även behållaren (av typen `JDesktopPane`) som är avsedd för interna fönster.

Klassen `javax.swing.JOptionPane` definierar speciella (statiska) metoder, som gör det möjligt att skapa en dialog i samband med (mitt i) ett internt fönster. Dessa metoder är: `showInternalConfirmDialog`, `showInternalInputDialog`, `showInternalMessageDialog` och `showInternalOptionDialog`. Dessa metoder används på samma sätt som metoderna `showConfirmDialog`, `showInputDialog`, `showMessageDialog` och `showOptionDialog`. De skapade dialogerna binds bara till interna fönster, istället för till ramar.

© FADIL GALJIĆ

Hantera händelser

Komponenter, händelser och lyssnare

Komponenter och deras händelser

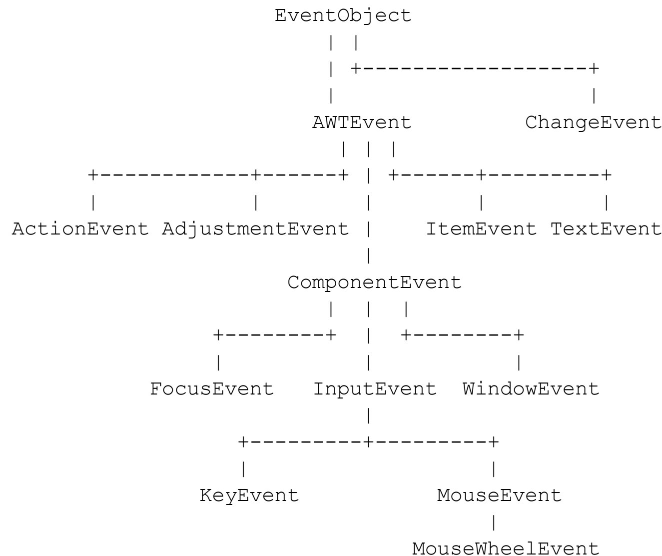
Ett grafiskt användargränssnitt skapas med olika grafiska komponenter. Paneler, knappar, textfält, textareor och andra komponenter används. Komponenterna grupperas och placeras på ett lämpligt sätt, och lagras i ett fönster. Fönstret visas, och på så sätt skapas ett grafiskt användargränssnitt mellan programmet och användaren.

Komponenter i ett grafiskt användargränssnitt kan användas på olika sätt. Användaren kan klicka på någon av knapparna i gränssnittet. En meny kan öppnas, och ett av flera möjliga alternativ kan väljas. Användaren kan skriva in en text i ett textfält och sedan trycka på returtangenten. Alla dessa handlingar kallas för *händelser*.

En händelse representeras i ett Javaprogram med ett objekt av en lämplig klass. Det finns olika typer av händelser, och därmed olika *händelseklasser*. Alla dessa klasser är organiserade i en klasshierarki, med klassen `java.util.EventObject` i roten. Klassen `EventObject` har ett antal subklasser, som beskriver konkreta typer av händelser. Klasserna `javax.swing.event.ChangeEvent` och `javax.swing.event.ListDataEvent`, till exempel, är direkta subklasser till klassen `EventObject`. En viktig subklass till klassen `EventObject` är klassen `java.awt.AWTEvent`. Det är en abstrakt klass, som är superklass till många andra händelseklasser. Klasserna `ActionEvent`, `AdjustmentEvent`, `ItemEvent`, `TextEvent` och `WindowEvent`, till exempel, är (direkta eller indirekta) subklasser till klassen `AWTEvent`. Dessa subklasser finns i paketet `java.awt.event` (superklassen `AWTEvent` finns i paketet `java.awt`).

En del av den klasshierarkin som bildas av olika händelseklasser, kan representeras så här:

© FADIL GALJIĆ



När en händelse inträffar, skapas automatiskt ett objekt (av en lämplig klass) som representerar denna händelse. När till exempel en knapp trycks, skapas automatiskt ett objekt av typen `ActionEvent`. När returtangenten trycks inuti ett textfält, skapas också ett objekt av typen `ActionEvent`. Dessa två händelser är av samma typ. En händelse av en annan typ inträffar när användaren trycker på stängningsknappen i en ram. Då skapas ett objekt av typen `WindowEvent`.

Ett objekt som representerar en händelse innehåller olika typer av information om denna händelse. Ett händelseobjekt innehåller till exempel information om källan till händelsen. Metoden `getSource` ger en referens (av typen `Object`) till denna källa. Metoden definieras i klassen `EventObject`, och ärvs av alla andra händelseklasser. Om händelseobjektet `e` skapas när användaren klickar på en knapp, kan man göra så här:

```
Object     kallan = e.getSource ();
JButton    knapp = (JButton) kallan;
String     text = knapp.getText ();
```

Referensen `kallan` av typen `Object` refererar till källknappen. Denna referens omvandlas sedan till en referens av typen `JButton`. Med denna referens får man sedan den text som finns i knappen (en referens av typen `Object` känner inte metoden `getText`, och därför utförs omvandlingen).

En referens som refererar till en händelses källa, kan användas för att avgöra vilken komponent som genererat händelsen. Detta görs i en

© FADIL GALJIĆ

lyssnarklass som hanterar händelser för flera olika komponenter. Så här kan man göra:

```
Object     kallan = e.getSource ();
if (kallan instanceof JButton)
{
    JButton    knapp = (JButton) kallan;
    String     text = knapp.getText ();
    System.out.println (text);
}
else if (kallan instanceof JTextField)
{
    JTextField    falt = (JTextField) kallan;
    falt.setText ("OK");
}
```

Den här strategin (med operatorn `instanceof`) kan användas när källkomponenterna är av olika typer. Men det går inte att använda denna strategi för att avgöra vilken av två komponenter av samma typ som genererat en händelse. I detta fall behövs det referenser till dessa komponenter för att källa ska kunna bestämmas. Så här kan detta göras:

```
Object    kallan = e.getSource ();
if (kallan == knapp1)
    panel.setBackground (Color.BLUE)
else if (kallan == knapp2)
    panel.setBackground (Color.RED);
```

Bakgrundsfärgen för en given panel sätts till blå eller röd, beroende på vilken av två givna knappar som klickats. Referenserna `knapp1` och `knapp2` refererar till dessa knappar, och dessa referenser behöver tillföras till lyssnarklassen för att källknappen ska kunna bestämmas.

Olika händelseklasser definierar olika metoder, som gör det möjligt att få ytterligare, mer specifik information om en händelse.

Komponenter och deras lyssnare

En händelse kan genereras i samband med en komponent i ett grafiskt användargränssnitt. Man skapar en händelse när man vill att någonting ska hända i programmet, när en konkret metod behöver utföras. För att det blir möjligt att anropa en metod via användargränssnittet, måste en *lyssnare* skapas och registreras hos den aktuella komponenten. En lyssnare är ett objekt av en klass som implementerar ett *lyssnargränssnitt*. Det finns olika typer av händelser, och därför innehåller Javas standardbibliotek olika typer av lyssnargränssnitt. Ett lyssnargränssnitt specificerar en eller

© FADIL GALJIĆ

flera metoder som en *lyssnarklass* måste implementera. Det är en av dessa *lyssnarmetoder* som anropas automatiskt när en händelse inträffar. Vilken av metoderna som ska anropas beror på händelsen. Var och en av metoderna tar emot ett händelseobjekt som argument. Detta objekt skapas automatiskt när en händelse inträffar, och tillförs som argument till motsvarande metod. Via en lyssnarmetod kan man *fånga* ett händelseobjekt och reagera på den aktuella händelsen.

När en händelse inträffar, skapas automatiskt ett objekt som representerar händelsen. Om en lyssnare registrerats hos källkomponenten, anropas en lämplig metod automatiskt i samband med lyssnaren. Motsvarande kod utförs, och på så sätt reagerar programmet på händelsen. Om flera lyssnare registrerats som lyssnar på en typ av händelser hos en och samma komponent, anropas motsvarande metod i samband med alla dessa lyssnare.

Ett av lyssnargränssnitten i Javas standardbibliotek är gränssnittet `java.awt.event.ActionListener`. Detta gränssnitt har en enda metod. Det är metoden `actionPerformed`, som definieras så här:

```
void actionPerformed (ActionEvent e);
```

En klass som implementerar detta gränssnitt kan skapas så här:

```
class Avslutare implements ActionListener
{
    void actionPerformed (ActionEvent e)
    {
        System.exit (0);
    }
}
```

Man kan skapa ett objekt av denna klass (ett objekt som kan lyssna på händelser av typen `ActionEvent`), och registrera det hos en knapp (när en knapp klickas på, skapas en händelse av typen `ActionEvent`) i ett grafiskt användargränssnitt. Så här kan detta göras:

```
JButton    knapp = new JButton ("Avsluta");
Avslutare    lyssnare = new Avslutare ();
knapp.addActionListener (lyssnare);
```

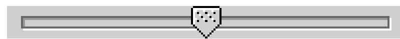
Om man gör på detta sätt, avslutas programmet så snart användaren klickar på den angivna knappen. Metoden `actionPerformed` anropas automatiskt, och det är den metoden som avslutar programmet. Man skapar en metod och binder denna på ett föreskrivet sätt till en komponent. Varje gång en händelse genereras i samband med komponenten, anropas metoden automatiskt.

 © FADIL GALJIĆ

Gränssnittet `javax.swing.event.ChangeListener` kan anges som ett annat exempel. Detta gränssnitt har bara en metod, `stateChanged`, som definieras så här:

```
void stateChanged (ChangeEvent e);
```

En händelse av typen `ChangeEvent` genereras när vissa grafiska komponenters tillstånd ändras. En av dessa komponenter representeras med ett objekt av typen `javax.swing.JSlider` (se bilden nedan). Vi kan kalla den komponenten för *vallinjal*.



En vallinjal kan användas för att justera ett värde mellan två givna värden. En vallinjal kan skapas så här:

```
JSlider     slider = new JSlider (JSlider.HORIZONTAL, 0, 10, 5);
```

En horisontell vallinjal skapas, som tillåter oss att justera ett värde mellan `0` och `10`. Det förvalda värdet är `5`. Metoden `getValue` ger en vallinjals aktuella värde. När en vallinjals värde ändras, skapas automatiskt ett objekt av typen `ChangeEvent`. Detta objekt representerar den händelse som inträffat. Om man vill att programmet ska reagera på händelsen, måste man skapa en klass som implementerar gränssnittet `ChangeListener`, och registrera ett objekt av denna klass som lyssnare hos vallinjalen. I metoden `stateChanged` i denna klass beskrivs programmets reaktion:

```
class Reaktion implement ChangeListener
{
    void stateChanged (ChangeEvent e)
    {
        Object     kallan = e.getSource ();
        JSlider    slider = (JSlider) kallan;

        int   varde = slider.getValue ();
        System.out.println (varde);
    }
}

slider.addChangeListener (new Reaktion ());
```

Så snart vallinjalens tillstånd ändras (en händelse - ett nytt värde väljs), skapas ett objekt av typen `ChangeEvent` som representerar händelsen. Metoden `stateChanged` anropas, och vallinjalens nya värde visas på standardutmatningsenheten. Detta värde kan naturligtvis användas på ett annat sätt, till exempel för att justera andra variabler i programmet (en storlek, en position, en färg och så vidare).

© FADIL GALJIĆ

Javas standardbibliotek definierar olika typer av grafiska komponenter och händelser, och olika lyssnargränssnitt. Genom att använda dessa element på ett lämpligt sätt, kan man skapa ett grafiskt gränssnitt mellan ett program och dess användare.

Lyssnare med flera metoder

Flera händelser av olika typ kan representeras med en gemensam klass. Detta görs till exempel i samband med ett fönster. Man kan öppna eller stänga ett fönster, omvandla ett fönster till en ikon eller återställa det, göra ett fönster aktivt eller lämna det. Alla dessa händelser representeras med en gemensam klass, `java.awt.event.WindowEvent`. Händelser i ett fönster kan vara av olika typ, men i grunden är de alla av typen `WindowEvent`.

Ett lämpligt lyssnargränssnitt har definierats, för att händelser som genereras i ett fönster ska kunna hanteras. Det är gränssnittet `WindowListener` i paketet `java.awt.event`. Detta gränssnitt definieras så här:

```
public interface WindowListener
{
    void windowOpened (WindowEvent e);
    void windowClosing (WindowEvent e);
    void windowClosed (WindowEvent e);
    void windowIconified (WindowEvent e);
    void windowDeiconified (WindowEvent e);
    void windowActivated (WindowEvent e);
    void windowDeactivated (WindowEvent e);
}
```

Gränssnittet `WindowListener` definierar sju olika lyssnarmetoder. Metoden `windowOpened` anropas när ett fönster öppnats, metoden `windowClosing` anropas när användaren trycker på ett fönsters stängningsknapp, metoden `windowClosed` anropas när ett fönster stängts, och så vidare. Olika lyssnarmetoder anropas vid olika typer av händelser. Trots att alla dessa händelser i grunden är av samma typ (`WindowEvent`), skiljer Java dessa händelser från varandra och anropar alltid rätt lyssnarmetod.

En klass som implementerar ett lyssnargränssnitt med flera metoder, måste implementera alla dessa metoder. Alla metoderna måste implementeras även om det inte finns något behov av att hantera alla typer av händelser. Man kanske vill hantera den händelse som uppstår när användaren trycker på ett fönsters stängningsknapp. Motsvarande reaktion definieras då i metoden `windowClosing`. Men även de andra metoderna i gränssnittet `WindowListener` måste implementeras. Om inte händelserna som motsva-

 © FADIL GALJIĆ

rar dessa metoder behöver hanteras, kan metoderna implementeras på så sätt att de inte gör någonting. Det går att göra så här:

```
class Avslutare implements WindowListener
{
    void windowClosing (WindowEvent e)
    {
        System.exit (0);
    }

    void windowOpened (WindowEvent e) {}
    void windowClosed (WindowEvent e) {}
    void windowIconified (WindowEvent e) {}
    void windowDeiconified (WindowEvent e) {}
    void windowActivated (WindowEvent e) {}
    void windowDeactivated (WindowEvent e) {}
}
```

Ett objekt av klassen `Avslutare` kan användas som lyssnare i samband med olika fönster:

```
JFrame     frame = new JFrame ();
Avslutare    lyssnare = new Avslutare ();
frame.addWindowListener (lyssnare);
```

En ram skapas, och ett objekt av typen `Avslutare` registreras som lyssnare hos denna ram. När ramens stängningsknapp trycks anropas lyssnarens metod `windowClosing`, som avslutar programmet.

I vissa fall kan implementeringen av en lyssnarklass med flera metoder underlättas. För de gränssnitt i paketet `java.awt.event` som innehåller flera metoder, har motsvarande lyssnarklasser med tomma metoder skapats. Dessa klasser kallas för *adapterklasser*. En av dessa klasser är klassen `java.awt.event.WindowAdapter`. Denna klass implementerar metoder i gränssnittet `WindowListener` på så sätt att metoderna inte gör någonting (sju tomma metoder). En lyssnarklass kan skapas som en subklass till klassen `WindowAdapter`. På så sätt ärvs alla sju metoderna från denna klass. Metoderna behöver inte definieras på nytt. Man behöver bara omdefiniera de metoder vars beteende man vill ändra. Man kan till exempel göra så här:

```
class Avslutare extends WindowAdapter
{
    void windowClosing (WindowEvent e)
    {
        System.exit (0);
    }
}
```

© FADIL GALJIĆ

Lyssnarklassen `Avslutare` är en subklass till adapterklassen `WindowAdapter`. Eftersom klassen `WindowAdapter` implementerar gränssnittet `WindowListener`, implementerar även klassen `Avslutare` detta gränssnitt. Klassen `Avslutare` definierar sin egen `windowClosing`-metod, och ärver de andra metoderna från klassen `WindowAdapter`.

Olika sätt att implementera en lyssnarklass

En självständig lyssnarklass

En lyssnarklass kan implementeras som en självständig klass. Klassen kan definieras i en särskild fil, eller i den fil där man skapar de komponenter vars händelser lyssnarklassen hanterar. Om en lyssnarklass definieras som en självständig klass, måste man förse denna klass med all den information som är nödvändiga för händelsehanteringen. Denna information tillförs till lyssnarklassen antingen som argument till klassens konstruktorer, eller som argument till klassens (`set`-) metoder.

Säg att man skapar en panel med ett textfält och en textarea. Textfältet är placerat i panelens nedre del och textarean i mitten. En sådan panel kan utformas så här:

```
class FPanel extends JPanel
{
    public FPanel ()
    {
        JTextField    textFalt = new JTextField (40);
        JTextArea     textArea = new JTextArea (40, 20);

        JPanel    panel = new JPanel ();
        panel.add (textFalt);

        this.setLayout (new BorderLayout ());
        this.add (textArea, "Center");
        this.add (panel, "South");
    }
}
```

I textarean vill man visa det som användaren anger i textfältet. Detta beteende definieras i en särskild lyssnarklass, på följande vis:

```
class HandelseHanterare implements ActionListener
{
    private JTextArea     textArea;
```

 © FADIL GALJIĆ

```
    public HandelseHanterare (JTextArea textArea)
    {
        this.textArea = textArea;
    }

    public void actionPerformed (ActionEvent e)
    {
        Object    kallan = e.getSource ();
        JTextField    textFalt = (JTextField) kallan;

        String    text = textFalt.getText ();
        textArea.append (text + "\n");
    }
}
```

När klassen definierats, skapar man ett objekt av den och registrerar objektet som lyssnare hos textfältet. Detta görs på den plats där textfältet och textarean skapas (i konstruktorn i klassen `FPanel`). Man gör så här:

```
ActionListener    hanterare = new HandelseHanterare (textArea);
textFalt.addActionListener (hanterare);
```

Lyssnarklassen `HandelseHanterare` avläser texten från textfältet och visar den i textarean. Textfältet, som är källa till händelsen, erhålls med metoden `getSource`. Men det går inte att få textarean på detta sätt. Textarean måste anges som argument till klassens konstruktor, när ett objekt av klassen skapas. Detta visar vilka problem som uppstår när en lyssnarklass implementeras som en självständig klass. Lyssnarklassen finns på ett ställe i programmet, och komponenterna som har med händelser att göra på ett annat ställe. En bro mellan lyssnarklassen och dessa komponenter måste skapas. Detta görs genom att referenser till komponenterna anges som instansvariabler i lyssnarklassen. Dessa referenser sätts att referera till riktiga komponenter i lyssnarklassens konstruktorer, eller via motsvarande metoder i lyssnarklassen. På så sätt (via referenserna) kan en lyssnarklass belägen på en plats, påverka en grafisk komponent som finns på en annan plats.

En inre lyssnarklass

Istället för att implementera en lyssnarklass som en självständig klass, kan man implementera den som en inre klass i den klass där de relevanta grafiska komponenterna skapas. På så sätt blir det möjligt för lyssnarklassen att direkt använda alla dessa grafiska komponenter. Det behövs inte några referenser som broar mellan lyssnarklassen och komponenterna i

det grafiska gränssnittet. Lyssnarklassen definieras nära de grafiska komponenterna, så att dessa blir tillgängliga i lyssnarklassen.

På följande vis kan man definiera en panel med ett textfält, en textarea och en inre lyssnarklass som reagerar på händelser i textfältet (avläser texten från textfältet och visar den i textarean):

```
class FPanel extends JPanel
{
    private JTextField    textFalt;
    private JTextArea     textArea;


    private class HandelseHanterare implements ActionListener
    {
        public void actionPerformed (ActionEvent e)
        {
            String    text = textFalt.getText ();
            textArea.append (text + "\n");
        }
    }


    public FPanel ()
    {
        textFalt = new JTextField (40);
        textArea = new JTextArea (40, 20);

        ActionListener    hanterare = new HandelseHanterare ();
        textFalt.addActionListener (hanterare);

        JPanel    panel = new JPanel ();
        panel.add (textFalt);

        this.setLayout (new BorderLayout ());
        this.add (textArea, "Center");
        this.add (panel, "South");
    }
}
```

I detta fall definieras textfältet och textarean som instansvariabler i klassen `FPanel`. På så sätt blir alla dessa variabler tillgängliga i den inre klassen. Den inre klassen `HandelseHanterare` kan fritt använda variablerna för sina ändamål. På så vis underlättas implementeringen av lyssnarklassen. Man behöver inte använda metoden `getSource` för att få tillgång till händelsens källa (textfältet). Man behöver inte heller skicka textarean som argument till klassens konstruktor. Klassen `HandelseHanterare` behöver överhuvudtaget inte innehålla en sådan konstruktor.

© FADIL GALJIĆ

Ett objekt av en inre klass skapas alltid i samband med ett objekt av den omgivande (yttre) klassen. Det inre objektet kan fritt använda instansvariablerna och instansmetoderna i det omgivande objektet. Ett inre objekt fyller alltid sin funktion i samband med ett omgivande objekt. Ett objekt av typen `HandelseHanterare` skapas i klassen `FPanel`, i klassens konstruktor. Ett sådant objekt skapas därför alltid när en panel (ett objekt av den omgivande klassen) skapas. Koden i klassens konstruktor (i klassen `FPanel`) utförs, och ett inre objekt skapas i samband med den panelen som skapas. Detta inre objekt kan fritt använda panelens instansvariabler (`textFalt` och `textArea`).

Ett inre objekt kan även skapas i en metod i den omgivande klassen. I detta fall skapas ett sådant objekt varje gång som metoden exekveras. Det inre objektet skapas i samband med det omgivande objekt som aktiverar metoden, och kan använda det omgivande objektets instansvariabler.

En inre anonym lyssnarklass

En inre lyssnarklass kan definieras som en anonym (namnlös) klass, när ett objekt (en lyssnare) av klassen skapas. En inre, anonym lyssnarklass definieras i en konstruktor eller i en metod i den klass där de relevanta grafiska komponenterna skapas. På så sätt placeras definitionen av en lyssnarklass så nära de aktuella grafiska komponenterna som möjligt. En inre, anonym lyssnarklass kan använda den omgivande klassens instansvariabler och instansmetoder. Men även mer: en sådan klass kan även använda lokala `final`-variabler i den konstruktor eller metod som den definieras i.

En inre, anonym lyssnarklass som hanterar händelser i samband med ett textfält, kan definieras så här:

```
class FPanel extends JPanel
{
    private JTextField    textFalt;
    private JTextArea     textArea;


    public FPanel ()
    {
        textFalt = new JTextField (40);
        textArea = new JTextArea (40, 20);


        ActionListener    hanterare = new ActionListener ()
            {
```

```
                public void actionPerformed (ActionEvent e)
                {
                    String    text = textFalt.getText ();
                    textArea.append (text + "\n");
                }
            };

        textFalt.addActionListener (hanterare);


        JPanel    panel = new JPanel ();
        panel.add (textFalt);

        this.setLayout (new BorderLayout ());
        this.add (textArea, "Center");
        this.add (panel, "South");
    }
}
```

Här definieras en lyssnarklass, och samtidigt skapas ett objekt av denna klass. Klassen är namnlös, men en referens av typen `ActionListener` kan användas för att referera till det objekt som skapas (eftersom objektets klass implementerar gränssnittet `ActionListener`). Efter operatorn `new` anges det gränssnitt som den anonyma klassen implementerar, följt av två parenteser. Sedan definieras den anonyma klassens metoder mellan två klamrar. Hela satsen avslutas som vanligt, med ett semikolon. Om den anonyma klassen ärver från en klass (till exempel en adapterklass), anges superklassens namn efter operatorn `new`. I detta fall kan de nödvändiga argumenten till superklassens konstruktor anges mellan de parenteser som följer på superklassens namn.

Eftersom en inre, anonym klass kan använda lokala `final`-variabler, kan variablerna `textFalt` och `textArea` vara lokala variabler (istället för instansvariabler) i konstruktorn. Dessa variabler skapas i så fall så här:

```
final JTextField    textFalt = new JTextField (40);
final JTextArea     textArea = new JTextArea (40, 20);
```

Man kan ju undra om variablerna `textFalt` och `textArea` ska skapas som instansvariabler eller som lokala `final`-variabler. Variablerna kan vara instansvariabler för att framhäva en panels uppbyggnad. Via instansvariabler i en klass specificerar man vad ett objekt av klassen har och hur detta objekt ser ut. De variabler som används i klassen, men som inte är väsentliga för ett objekts uppbyggnad, kan skapas som lokala variabler där de behövs. Variabeln `panel`, till exempel, är en hjälpvariabel som endast används för att lagra textfältet. Man kan därför ha denna variabel som en lokal variabel i klassens konstruktor.

 © FADIL GALJIĆ

Lokala lyssnarklasser

Ett objekt kan skapas i förväg, och sedan kan detta objekt anges som argument till en metod. Ett objekt kan också skapas vid samma tillfälle som objektet anges som argument. Man gör på detta sätt om objektet skapas endast för att tillföras som argument till en metod (och inte för några andra ändamål). Man kan gå ett steg längre: man kan definiera en klass och skapa ett objekt av klassen när man vill använda ett sådant objekt som argument. En klass som definieras på detta sätt är en inre, anonym, lokal klass. Denna strategi används ofta för att definiera, skapa och registrera en lyssnare. På så vis förenklas lyssnarjobbet så mycket som möjligt.

En lokal lyssnarklass kan definieras och användas så här:

```
textFalt.addActionListener (new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            String    text = textFalt.getText ();
            textArea.append (text + "\n");
        }
    } );
```

En lyssnarklass definieras, ett objekt av klassen skapas och objektet registreras som lyssnare hos textfältet `textFalt`. Objektet skapas när det ska användas som argument till metoden `addActionListener`, och objektets klass definieras vid detta tillfälle. Liksom andra inre, anonyma klasser, kan även en lokal klass använda den omgivande klassens instansvariabler, instansmetoder och lokala `final`-variabler.

En grafisk komponent som lyssnare

En lyssnarklass kan definieras som en inre klass, som en inre, anonym klass eller som en inre, anonym, lokal klass. På så vis skapas lyssnarklassen nära de relevanta grafiska komponenterna, så att dessa komponenter blir tillgängliga i lyssnarklassen. Denna strategi underlättar implementeringen av en lyssnarklass, och placerar lyssnarklassen nära händelsernas källor. Detta ger en mer överskådlig programstruktur.

Ett annat sätt att underlätta implementeringen av en lyssnarklass är att välja de olika komponenternas behållare som lyssnare för komponenternas händelser. Så här kan detta göras:

```
class FPanel extends JPanel implements ActionListener
{
    private JTextField    textFalt;
```

© FADIL GALJIĆ

```
    private JTextArea     textArea;

    public FPanel ()
    {
        textFalt = new JTextField (40);
        textFalt.addActionListener (this);
        textArea = new JTextArea (40, 20);

        JPanel    panel = new JPanel ();
        panel.add (textFalt);

        this.setLayout (new BorderLayout ());
        this.add (textArea, "Center");
        this.add (panel, "South");
    }

    public void actionPerformed (ActionEvent e)
    {
        String    text = textFalt.getText ();
        textArea.append (text + "\n");
    }
}
```

Klassen `FPanel` definierar en panel som innehåller ett antal grafiska komponenter. En panel av denna klass är en behållare för dessa komponenter. Men även mer: en sådan panel är en lyssnare för de händelser som genereras av dessa komponenter. Klassen `FPanel` implementerar gränssnittet `ActionPerformed`, och definierar vad som ska hända när returtangenten trycks i ett textfält i panelen. Metoden `actionPerformed` finns i klassen `FPanel`, och kan därför använda alla instansvariabler och instansmetoder i denna klass. Det underlättar implementeringen av lyssnarklassen.

Ett textfält skapas i klassen `FPanel`, i dess konstruktor. När en panel av klassen skapas, skapas även ett textfält i panelen. Man refererar till den panelen som skapas med referensen `this` (den här panelen). Därför registreras panelen som lyssnare hos textfältet på följande sätt:

```
textFalt.addActionListener (this);
```

Om den här strategin används för att implementera en lyssnarklass, placeras lyssnarkoden lite längre bort från den komponent som genererar händelser. Programstrukturen blir mindre åskådlig än när en inre, anonym lyssnarklass används. Därför är inre, anonyma (och inre, anonyma, lokala) lyssnarklasser att föredra. Denna strategi med en behållare som lyssnare, har en nackdel till. En behållarklass kan inte ärva från en adapterklass, eftersom den redan ärver från en annan behållarklass. Klassen `FPanel` ärver redan från klassen `JPanel`, och kan därmed inte ärva från en klass till.

© FADIL GALJIĆ

Mushändelser

En mushändelse

När användaren klickar på en knapp, skapas en händelse av typen `java.awt.event.ActionPerformed`. Denna händelse kan fångas med en lyssnare av typen `java.awt.event.ActionListener`. En sådan lyssnare har metoden `actionPerformed`, och det är denna metod som anropas när knappen klickas. Metoden `actionPerformed` representerar programmets reaktion på en klickning på knappen.

En händelse av typen `ActionPerformed` är en *högnivåhändelse* (en *semantisk* händelse). Denna händelse är resultatet av flera lågnivåaktiviteter, eller *lågnivåhändelser*. Muspekaren flyttas in över en knapp, musknappen trycks ner, och släpps sedan upp. Det utförs flera samordnade operationer med musen i samband med knappen. Alla dessa operationer utgör olika händelser. Dessa lågnivåhändelser kan fångas var och en för sig, och hanteras på ett särskilt sätt. På så sätt kan man åstadkomma en större kontroll (om en sådan kontroll behövs) i ett program.

En händelse av typen `ActionPerformed` är resultatet av flera händelser som genereras med musen i samband med en knapp. Det är en högnivåhändelse. Exempel på andra högnivåhändelser är händelser av typen `AdjustmentEvent`, `ItemEvent`, `TextEvent` och `ChangeEvent`. En mus kan generera händelser av olika typer, men alla dessa händelser är i grunden av typen `java.awt.event.MouseEvent`. En händelse av typen `MouseEvent` är en lågnivåhändelse. Exempel på andra lågnivåhändelser är händelser av typen `ComponentEvent` (superklass till alla lågnivåhändelser), `InputEvent` (direkt superklass till klassen `MouseEvent`), `KeyEvent`, `MouseWheelEvent` och `WindowEvent`.

Klassen `MouseEvent` har flera metoder som gör det möjligt att få olika slags information om en mushändelse. Metoden `getClickCount` ger antalet klickningar. Användaren kan utföra en enkelklickning, en dubbelklickning, en trippelklickning och så vidare. Om en dubbelklickning utförs, registreras den första klickningen även som en enkelklickning, om en trippelklickning utförs, registreras även en enkelklickning och en dubbelklickning, och så vidare. Detta innebär att metoden `getClickCount` vid en trippelklickning returnerar först `1`, sedan `2` och slutligen `3`. Olika musknappar kan användas när en händelse skapas. Metoden `getButton` ger information om vilken knapp som ändrat sitt tillstånd. Metoden returnerar `1` för den vänstra knappen, `2` för den mittersta knappen och `3` för den högra knappen. Det returnerade värdet kan jämföras med en av flera kon-

© FADIL GALJIĆ

stanter i klassen MouseEvent. Dessa konstanter är BUTTON1, BUTTON2, BUTTON3 och NOBUTTON (om ingen knapp har ändrat sitt tillstånd). I vissa sammanhang är det viktigt att kunna bestämma den punkt i en grafisk komponent där en händelse med musen genererats. Metoderna getX och getY ger koordinaterna för denna punkt.

Hantera mushändelser

Händelser av typen MouseEvent genereras då muspekaren kommer in över en grafisk komponent och när den lämnar en grafisk komponent. Händelser av denna typ genereras även när en musknapp trycks ner över en komponent och när en nedtryckt musknapp släpps upp. Dessutom genereras händelser av typen MouseEvent när muspekaren flyttas över en komponent och när användaren drar med musen (trycker ner en musknapp över en komponent och sedan flyttar musen).

Mushändelser fångas med två typer av lyssnare. Händelser som genereras när muspekaren flyttas in över en komponent och när muspekaren lämnar en komponent, hanteras av en lyssnare av typen MouseListener. En lyssnare av denna typ används även för att hantera händelser som genereras när en musknapp trycks ner eller släpps upp. Händelser som genereras när man flyttar eller drar musen, fångas med en lyssnare av typen MouseMotionListener.

Gränssnittet MouseListener definierar följande metoder: mouseEntered (anropas när muspekaren flyttas in över en komponent), mouseExited (anropas när muspekaren lämnar en komponent), mousePressed (anropas när en musknapp trycks ner), mouseReleased (anropas när en nertryckt musknapp släpps upp) och mouseClicked (anropas när musen klickas i en komponent). När användaren klickar med musen, anropas tre metoder. Först anropas metoden mousePressed, därefter metoden mouseReleased och slutligen metoden mouseClicked. För att underlätta hanteringen av mushändelser, har det skapats en adapterklass som implementerar gränssnittet MouseListener. Det är klassen MouseAdapter. Normalt ärver man från denna adapterklass, och omdefinierar endast de metoder vars beteende man vill ändra (metoderna i klassen MouseAdapter gör ingenting).

Gränssnittet MouseMotionListener definierar två metoder: mouseMoved (anropas när muspekaren flyttas inuti en komponent) och mouseDragged (anropas när en musknapp trycks ned inuti en komponent och muspekaren sedan flyttas). När musen flyttas eller dras, genereras en serie händelser. Detta innebär att motsvarande metod ständigt anropas. Denna metod kan ge muspekarens aktuella position i den aktuella komponenten. Till

© FADIL GALJIĆ

detta används metoderna `getX` och `getY`. Liksom för andra gränssnitt (i paketet `java.awt.event`) som har flera metoder, definieras en adapterklass även för gränssnittet `MouseMotionListener`. Det är klassen `MouseMotionAdapter`, som har två tomma metoder.

En lyssnare av typen `MouseListener` registreras hos en grafisk komponent med metoden `addMouseListener`. En lyssnare av typen `MouseMotionListener` registreras med metoden `addMouseMotionListener`.

En panel kan definieras, där en boll ritas på den plats där musen befinner sig. När musen dras, dras även bollen med musen. Så här kan man göra:

```
class BollPanel extends JPanel
{
    private Ellipse2D    boll = null;

    public BollPanel ()
    {
        this.setBackground (Color.WHITE);

        MouseMotionListener lyssnare = new MouseMotionAdapter ()
            {
                public void mouseDragged (MouseEvent e)
                {
                    int    x = e.getX ();
                    int    y = e.getY ();

                    boll = new Ellipse2D.Double ();
                    boll.setFrameFromCenter (x, y,
                                     x - 5, y - 5);

                    repaint ();
                }
            };

        this.addMouseMotionListener (lyssnare);
    }

    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        if (boll != null)
            g.fill (boll);
    }
}
```

En lyssnare av typen `MouseMotionListener` definieras och registreras hos panelen. Lyssnaren fångar och hanterar de händelser som genereras när

© FADIL GALJIĆ

musen dras. Dessa händelser hanteras i metoden mouseDragged. Metoden bestämmer musens aktuella position och skapar en boll på denna plats. Panelens repaint-metod anropas för att rita om panelen, och därmed bollen. Metoden mouseDragged anropas så länge musen dras (även när musen dras utanför panelen). Som en konsekvens av detta kommer en boll att röra sig tillsammans med musen.

Tangentbordshändelser

En tangentbordshändelse

Varje gång användaren trycker på en tangent på tangentbordet, skapas en händelse. En sådan händelse representeras med ett objekt av typen java.awt.event.KeyEvent. En händelse av typen KeyEvent kan skapas på tre olika sätt. Ett sätt att generera en sådan händelse är att trycka ner en tangent på tangentbordet (ner). Ett annat sätt är att släppa upp en nedtryckt tangent (upp). En händelse av typen KeyEvent genereras även när en tangent på tangentbordet trycks på (ner och upp).

I klassen KeyEvent har ett antal metoder och konstanter definierats, som gör det möjligt att identifiera den aktuella tangenten och det skrivna tecknet. För att kunna identifiera en tangent, har ett antal heltalskonstanter som representerar olika tangenter definierats. De tangenter som innehåller olika bokstäver representeras med konstanterna VK_A, VK_B, VK_C och så vidare. De tangenter som innehåller olika siffror representeras med konstanterna VK_0, VK_1, VK_2 och så vidare. Bland de övriga konstanterna finns även VK_COMMA, VK_OPEN_BRACKET, VK_SPACE, VK_ENTER, VK_TAB, VK_ESCAPE, VK_CONTROL, VK_SHIFT, VK_ALT, VK_F1, VK_F2, VK_LEFT och VK_RIGHT.

Den tangent som genererat en händelse identifieras med metoden getKeyCode. Metoden returnerar ett heltal av typen int, som representerar den aktuella tangentens kod (eng. virtual key code). Detta heltal jämförs sedan med en av flera konstanter i klassen KeyEvent. Om e representerar en referens till ett objekt av typen KeyEvent, kan man göra så här:

```
int    tangentKod = e.getKeyCode ();
if (tangentKod == KeyEvent.VK_RIGHT)
    x++;
```

Om händelsen genereras med högerpilen, uppdateras variabeln x.

Den statiska metoden getKeyText i klassen KeyEvent ger en beskrivning av den aktuella tangenten. En tangents kod anges som argument till me-

© FADIL GALJIĆ

toden, som returnerar en beskrivande teckensträng. Så här kan metoden användas:

```
String    tangentBeskrivning = KeyEvent.getKeyText (tangentKod));
```

Metoden ger beskrivningar som `Högerpil`, `Ctrl`, `N`, `Enter` och så vidare.

I vissa sammanhang är det viktigt att det tecken som den tryckta tangenten representerar, identifieras som ett Unicode-tecken. I detta fall används metoden `getKeyChar`. Metoden används så här:

```
char    tecken = e.getKeyChar ();
```

Metoden `getKeyChar` kan skilja mellan små och stora bokstäver, något som metoden `getKeyCode` inte kan. Metoden `getKeyCode` identifierar en tangent och metoden `getKeyChar` identifierar ett tecken. För att få vissa tecken, behöver flera tangenter användas. Tecknet `A`, till exempel, produceras genom att `SHIFT`-tangenten och `A`-tangenten trycks samtidigt. Om en tangent som inte representerar ett Unicode-tecken trycks (till exempel `Ctrl`-tangenten), returnerar metoden `getKeyChar` ett heltalsvärde som representeras med konstanten `CHAR_UNDEFINED` i klassen `KeyEvent`.

Ett antal tangenter är så kallade action-tangenter. Med metoden `isActionKey` kan man kontrollera om den aktuella tangenten är en action-tangent. Följande tangenter definieras i Java som action-tangenter: `TAB`, `CAPS LOCK`, `NUM LOCK`, `BACKSPACE`, `ENTER`, `INSERT`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `F1` till `F24`, `PRINT SCREEN`, `SCROLL LOCK`, `PAUSE`, `DELETE`, `HOME`, `END`, `PAGE UP` och `PAGE DOWN`.

Klassen `KeyEvent` ärver flera metoder från sin superklass `InputEvent` (gemensam superklass till `MouseEvent` och `KeyEvent`). Bland de ärvda metoderna finns `isAltDown`, `isAltGraphDown`, `isShiftDown`, `isControlDown` och `isMetaDown`. Med dessa metoder kan man kontrollera om en av kontrolltangenterna är nertryckt.

Hantera tangentbordshändelser

För att kunna fånga händelser av typen `KeyEvent`, skapar man en lyssnare av typen `java.awt.event.KeyListener`. Gränssnittet `KeyListener` definierar följande metoder: `keyPressed`, `keyReleased`, och `keyTyped`. Dessa metoder implementeras (som tomma metoder) i adapterklassen `java.awt.event.KeyAdapter`. Man kan ärva från den klassen, och behöver då endast omdefiniera de metoder vars beteende man vill ändra. Metoden `keyPressed` omdefinieras om en händelse som uppstår när en tangent trycks ner ska hanteras. Metoden anropas så länge en tangent hålls ned-

© FADIL GALJIĆ

tryckt. Metoden `keyReleased` omdefinieras för att fånga det ögonblick då en nedtryckt tangent släpps upp. Metoden `keyTyped` omdefinieras för att få det Unicode-tecken som produceras när användaren trycker på en tangent (eller på en kombination av tangenter).

För att tangentbordshändelser ska kunna hanteras, måste en tangentbordslyssnare bindas till en eller flera grafiska komponenter i det grafiska användargränssnittet. Tangentbordhändelser fångas endast när komponenten är i fokus. För att en grafisk komponent ska hamna i fokus, aktiveras motsvarande fönster och sedan flyttas musen över komponenten. I Windows måste användaren klicka på komponenten. När en komponent hamnar i fokus, ändras komponentens utseende på något sätt. Det förvalda beteendet för vissa komponenter är att de inte kan sättas i fokus. Detta gäller till exempel för en panel. Detta beteende kan ändras med metoden `setFocusable` i klassen `java.awt.Component`. Metoden anropas i samband med en grafisk komponent, och `true` anges som argument till metoden. Med metoden `isFocusOwner` (i klassen `java.awt.Component`) kan man kontrollera om en grafisk komponent är i fokus för tillfället. Genom att använda metoden `requestFocus` (i klassen `java.awt.Component`) i samband med en komponent, kan man begära inifrån ett program att komponenten hamnar i fokus.

I vissa sammanhang vill man ta reda på vilken av flera grafiska komponenter i det grafiska användargränssnittet som är i fokus. Till detta används metoden `getFocusOwner`, som returnerar komponenten i fokus som ett objekt av typen `java.awt.Component`. Metoden `getFocusOwner` finns i klassen `java.awt.KeyboardFocusManager`. Ett objekt av klassen erhålls med klassens statiska metod `getCurrentKeyboardFocusManager`.

Hanteringen av tangentbordshändelser kan illustreras med följande exempel:

```
class TeckenPanel extends JPanel
{
    private char    tecken = 'A';
    private int     x = 100;
    private int     y = 80;

    public TeckenPanel ()
    {
        this.setBackground (Color.WHITE);
        this.setFocusable (true);

        KeyListener    lyssnare = new KeyAdapter ()
        {
            public void keyTyped (KeyEvent e)
            {
                tecken = e.getKeyChar ();
```


© FADIL GALJIĆ

```
                repaint ();
            }

            public void keyPressed (KeyEvent e)
            {
                int    tangentKod = e.getKeyCode ();

                if (tangentKod == KeyEvent.VK_RIGHT)
                    x = x + 10;
                else if (tangentKod == KeyEvent.VK_LEFT)
                    x = x - 10;
                else if (tangentKod == KeyEvent.VK_DOWN)
                    y = y + 10;
                else if (tangentKod == KeyEvent.VK_UP)
                    y = y - 10;

                repaint ();
            }
        };

        this.addKeyListener (lyssnare);
    }

    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);
        Graphics2D    g = (Graphics2D) gr;

        g.setFont (
             new Font ("Serif", Font.BOLD + Font.ITALIC, 40));
        g.setPaint (Color.PINK);
        g.drawString ("" + tecken, x, y);
    }
}
```

Här definieras en panel, som visar ett givet tecken på en bestämd plats i panelen. För att kunna ändra det tecken som visas och flytta tecknet i panelen, skapar man en tangentbordslyssnare och registrerar den hos panelen. Adapterklassen `KeyAdapter` används som superklass, och metoderna `keyTyped` och `keyPressed` omdefinieras. Metoden `keyTyped` anropas varje gång ett Unicode-tecken produceras med en tangent eller med en kombination av tangenter (till exempel `Shift-A`). Tecknet som användaren producerat bestäms och visas i panelen (genom att panelens `repaint`-metod anropas). Metoden `keyPressed` anropas varje gång en tangent trycks ner. Metoden anropas så länge tangenten hålls nedtryckt. I metoden `keyPressed` identifieras den nedtryckta tangenten, och om denna är någon av piltangenterna ändras det aktuella tecknets position i panelen. På så sätt kan tecknet flyttas i panelen med hjälp av piltangenterna.

© FADIL GALJIĆ

© FADIL GALJIĆ

Kapitel 6
Användargränssnittets funktioner

Texthantering
Textutmatning
Textinmatning
Bevaka ändringar i ett dokument

Val mellan olika alternativ
Valkomponenter
Komponenter med två lägen
Symbolbehållare
Symbolgeneratorer

Strukturerade val
Ett verktygsfält
Menyer

Dialoger
Använda dialoger

Datahantering via tabeller
Presentera data via en tabell
Redigera data via en tabell
Val via en tabell

© FADIL GALJIĆ

Texthantering

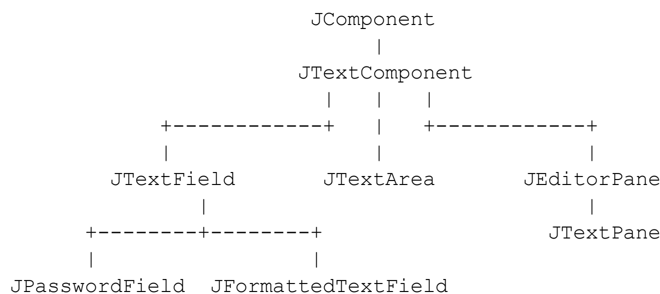
Textutmatning

Textkomponenter

Ett program tillför olika uppgifter till användaren under exekveringen. Om programmet har ett grafiskt gränssnitt, används detta som en kommunikationskanal mellan användaren och programmet. Olika komponenter i gränssnittet kan användas för att mata ut olika uppgifter.

Vissa uppgifter kan representeras med en eller flera teckensträngar. I dessa fall kan olika *textkomponenter* användas för att mata ut uppgifterna. Det grafiska gränssnittet kan innehålla en eller flera rutor, som programmet skriver ut olika meddelanden och resultat i. Det kan även innehålla mer avancerade textkomponenter, som gör det möjligt att kombinera text med bilder och grafiska komponenter.

Javas standardbibliotek har flera klasser som representerar olika textkomponenter. Dessa klasser bildar en klasshierarki, med den abstrakta klassen `JTextComponent` i roten (klassen `JTextComponent` finns i paketet `javax.swing.text` och dess subklasser i paketet `javax.swing`). Denna klasshierarki kan representeras så här.



Klassen `JTextField` och dess subklasser kan användas för att hantera en teckensträng bestående av en rad. Klassen `JTextArea` gör det möjligt att hantera en text bestående av ett godtyckligt antal rader. Klasserna `JEditorPane` och `JTextPane` gör det möjligt att hantera mer komplext stoff, som även kan inkludera bilder.

© FADIL GALJIĆ

Mata ut enkel text

Ett *textfält* av typen `javax.swing.JTextField` kan användas som ett utmatningsfält. En enradig teckensträng kan visas i ett sådant textfält. En teckensträng placeras i ett textfält med metoden `setText`. Teckensträngen anges som argument till metoden. Om en tom sträng tillförs som argument, töms det aktuella textfältet.

Man kan skapa ett textfält, och placera en teckensträng i det, på följande vis:

```
JTextField    outFalt = new JTextField (20);
outFalt.setEditable (false);
outFalt.setText ("Kärleken är så vacker!");
```

Här skapas ett textfält med `20` synliga positioner. En position kan rymma minst ett tecken (olika tecken har olika bredd). Om för många tecken skrivs, blir vissa tecken osynliga.

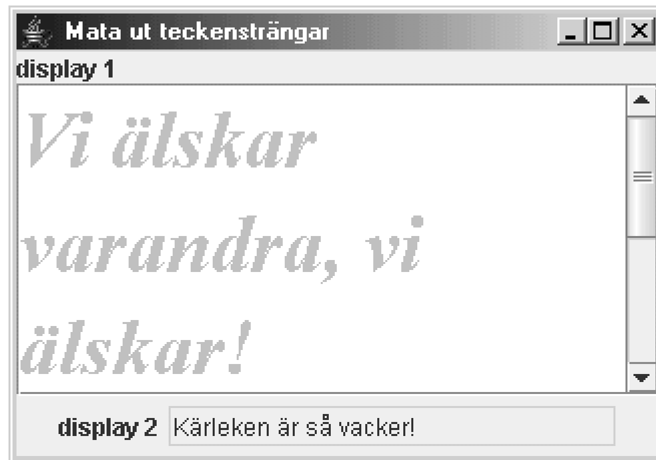
Ett textfält kan användas både som utmatningskomponent och inmatningskomponent. Inmatningsfunktionen kan inaktiveras genom att `false` anges som argument till metoden `setEditable`. Användaren kan därmed inte påverka textfältets innehåll (användaren kan inte skriva text i textfältet).

För att mata ut en text bestående av flera rader, används en *textarea* av typen `javax.swing.JTextArea`. Man kan skapa en textarea och placera en text i den på följande vis:

```
JTextArea    outArea = new JTextArea (4, 6);
outArea.setEditable (false);
outArea.setText ("Vi älskar varandra, vi älskar!\n\n" +
                 "Kärleken är så vacker!");
outArea.setLineWrap (true);
outArea.setWrapStyleWord (true);
```

Här skapas en textarea med `4` rader och `6` positioner i varje rad. Inmatningsfunktionen inaktiveras (så att användaren inte kan skriva i textarean), och en text bestående av flera rader placeras i textarean. Metoden `setLineWrap` används för att infoga brytningar vid textareans högra kant om raderna är för långa. För att denna brytning inte ska ske mitt i ett ord (enskilda ord ska inte delas upp på olika rader), används metoden `setWrapStyleWord` (se bilden nedan, som visar en textarea, ett textfält och två etiketter).

© FADIL GALJIĆ



Klassen JTextArea har flera metoder, som gör det enkelt att hantera texten i en textarea. Metoden getLineCount returnerar antalet rader i texten. Metoden append lägger till en given teckensträng till texten. Metoden insert infogar en given teckensträng på en given position i texten. Metoden replaceRange byter ut tecknen i ett givet område mot en given teckensträng. Dessa metoder kan användas så här:

```
int    antal = outArea.getLineCount ();
outArea.append ("\nKärleken är så vacker!");
outArea.insert ("\n", 30);
outArea.replaceRange ("", 0, 32);
```

I superklassen JTextComponent definieras ett antal metoder, och dessa metoder är gemensamma för alla textkomponenter. Bland annat definieras i denna klass metoderna setText (som placerar en given text i en textkomponent) och getText (som returnerar texten i en textkomponent). Metoderna select (som returnerar texten som finns mellan två givna positioner) och getSelectedText (som returnerar den markerade texten) definieras också här. Klassen definierar även metoderna isEditable (som kontrollerar om en textkomponent är redigerbar) och setEditable (som gör en textkomponent redigerbar eller icke-redigerbar).

Etiketter till textkomponenter

En textkomponent har inte en egen rubrik. En textkomponents funktion kan förtydligas genom att en *etikett* placeras i närheten av komponenten.

© FADIL GALJIĆ

En etikett kan representeras med ett objekt av typen `javax.swing.JLabel`. En etikett kan skapas så här:

```
JLabel    etikett = new JLabel ("display");
```

Här skapas en etikett med texten `display`. En komponent av typen `JLabel` kan innehålla en text eller en bild, eller både en text och en bild. Etikettens text kan specificeras med metoden `setText`, och dess bild med metoden `setImage`. Metoden `setImage` tar en *ikon* som argument. En ikon representeras med ett objekt vars klass implementerar gränssnittet `javax.swing.Icon`.

En ikon kan representeras med ett objekt av klassen `javax.swing.ImageIcon`. Detta är en direkt subklass till klassen `java.lang.Object`, som implementerar gränssnittet `javax.swing.Icon`. En ikon kan skapas utifrån en fil som innehåller en bild, till exempel så här:

```
ImageIcon    ikon = new ImageIcon ("fil.gif");
```

En ikon kan även skapas utifrån ett objekt av typen `java.awt.Image`. Man kan till exempel definiera en bild i en subklass till klassen `java.awt.image.BufferedImage` (som är en subklass till klassen `Image`), och tillföra ett objekt av denna subklass till motsvarande konstruktor när en ikon skapas.

Klassen `JLabel` har flera metoder som gör det möjligt att placera en text och en ikon på olika ställen i en etiketts area.

Visa en HTML-sida

Ett textdokument kanske bara innehåller den text som visas när dokumentet visas i en textkomponent. Men dokumentet kan även innehålla en text som inte syns. Den osynliga texten beskriver i detta fall den visade textens utseende. Fonter, färger, paragrafer och annan information om den synliga texten anges. Dokumentet innehåller både den text som visas och information av olika slag om denna texts utseende.

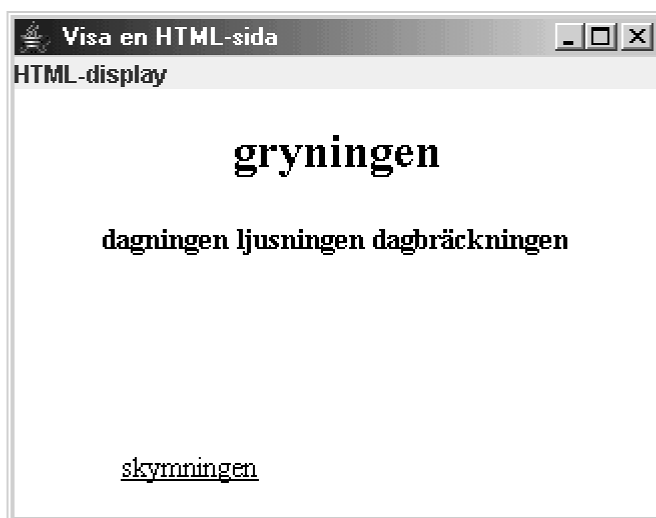
Ett HTML-dokument är ett exempel på ett dokument som innehåller både visad information och olika typer av redigeringsinformation. HTML (HyperText Modeling Language) är ett språk, som har utvecklats för att olika sidor på webben ska kunna beskrivas. En HTLM-sida beskrivs i en html-fil, som placeras på någon plats på Internet. HTLM-sidan kan sedan kommas åt och läsas med en webbläsare.

En text kan beskrivas med HTML på följande vis:

© FADIL GALJIĆ

```
<html>

    <head>
    <title>gryningen</title>
    </head>

    <body bgcolor="#ffffff">

    <center>
    <h1>gryningen</h1>
    <h3>dagningen  ljusningen  dagbräckningen</h3>
    </center>

    <br> <br> <br> <br>

    <ul>
    <a href="./skymningen.html"> skymningen</a>
    </ul>

    </body>

</html>
```

Den här texten kan placeras i filen `gryningen.html`. Filen kan sedan öppnas i en webbläsare. Sidan kommer endast att visa en del av filens text (se bilden nedan). Resten av texten beskriver den synliga textens utseende.



HTML är ett omfattande språk, som gör det möjligt att utforma webbsidor på önskat sätt. Om man behärskar detta språk, kan man även använda det för att formulera olika dokument i sina Javaprogram. En textkomponent

 © FADIL GALJIĆ

av typen `javax.swing.JEditorPane` kan visa en HTML-sida. Till detta används metoden `setPage`, och motsvarande html-fil anges som argument till metoden. Filen representeras via ett objekt av typen `java.net.URL`. Så här, till exempel, kan man göra:

```
JEditorPane    htmlDisplay = new JEditorPane ();
htmlDisplay.setEditable (false);

try
{
    URL    url = new URL (http://java.sun.com/index.html);
    htmlDisplay.setPage (url);
}
catch (IOException ex)
{
    htmlDisplay.setText (ex.toString ());
}
```

En textkomponent av typen `JEditorPane` skapas, och ett objekt av typen `URL` som representerar filen `index.html`, som finns i datorn `java.sun.com`. Denna fil finns på Internet, och kan nås med protokollet `http` (HyperText Transfer Protocol). För att visa den HTML-sida som beskrivs i filen, använder man metoden `setPage`.

När man ansluter till en dator på Internet och avläser en webbsida, kan olika kommunikationsproblem uppstå. I dessa fall kastas undantag av olika typer, och därför placeras motsvarande satser i ett `try`-block. Konstruktorn som skapar ett objekt av typen `URL` kan kasta ett undantag av typen `java.net.MalformedURLException` (en subklass till klassen `java.io.IOException`)., och metoden `setPage` kan kasta ett undantag av typen `java.io.IOException`.

En HTML-sida kan innehålla länkar till andra HTML-sidor. När man klickar på en sådan länk, visas den utpekade HTML-sidan. För att dessa länkar ska kunna fungera i samband med en komponent av typen `JEditorPane`, måste två saker göras. För det första måste man göra det omöjligt för användaren att ändra textkomponentens innehåll. Detta görs med metoden `setEditable`, med `false` som argument. Det andra som måste göras är att skapa en lyssnare av typen `javax.swing.event.HyperlinkListener`, och registrera denna lyssnare hos textkomponenten. Detta kan göras så här:

```
HyperlinkListener    linkListener = new HyperlinkListener ()
    {
        public void hyperlinkUpdate (HyperlinkEvent e)
        {
            if (e.getEventType ()
                    == HyperlinkEvent.EventType.ACTIVATED)
            {
```

© FADIL GALJIĆ

```
                try
                {
                    URL    url = e.getURL ();
                    htmlDisplay.setPage (url);
                }
                catch (IOException ex)
                {
                    htmlDisplay.setText (ex.toString ());
                }
            }
        }
    };

htmlDisplay.addHyperlinkListener (linkListener);
```

Varje gång muspekaren flyttas in över en länk, eller när användaren klickar med musen på en länk, skapas en händelse av typen `javax.swing.event.HyperlinkEvent`. Denna händelse fångas i metoden `hyperlinkUpdate` i en lyssnare av typen `HyperlinkListener`. Metoden `getEventType` (i klassen `HyperlinkEvent`) används för att avgöra vilken av tre möjliga händelser som inträffat, och det returnerade värdet jämförs med en av konstanterna i klassen `javax.swing.event.HyperlinkEvent.EventType`. Konstanten `ENTERED` betyder att muspekaren flyttats in över länken, konstanten `ACTIVATED` betyder att länken klickats och konstanten `EXITED` betyder att länken lämnats. Metoden `getURL` (i klassen `HyperlinkEvent`) ger det `URL`-objekt som motsvarar den aktuella länken (som anges i motsvarande html-fil). Därefter kan den motsvarande sidan visas (med metoden `setPage`).

Man kan definiera flera textdokument genom att använda HTML. Dessa dokument kan lagras som `html`-filer i den aktuella datorn. Dokumenten kan bindas i en helhet med en uppsättning länkar, och sedan hanteras med ett Javaprogram. Dokumenten kan visas, och det går att bläddra mellan dem. På så sätt kan en mängd olika information hanteras i ett program.

När en `html`-fil i den aktuella datorn representeras med ett `URL`-objekt, inleds den motsvarande teckensträngen med `file:` (inte med `http:`). Därefter anges sökvägen till filen, till exempel så här:

```
URL    url = new URL ("file:./gryningen.html");
```

Detta representerar filen `gryningen.html`, som finns i den aktuella katalogen i den aktuella datorn.

En textkomponent av typen `JEditorPane` kan visa en HTML-sida. Men den stöder inte alla de möjligheter som HTML erbjuder. Därför kan det hända att sidan som visas inte helt ser ut som det motsvarande HTML-

© FADIL GALJIĆ

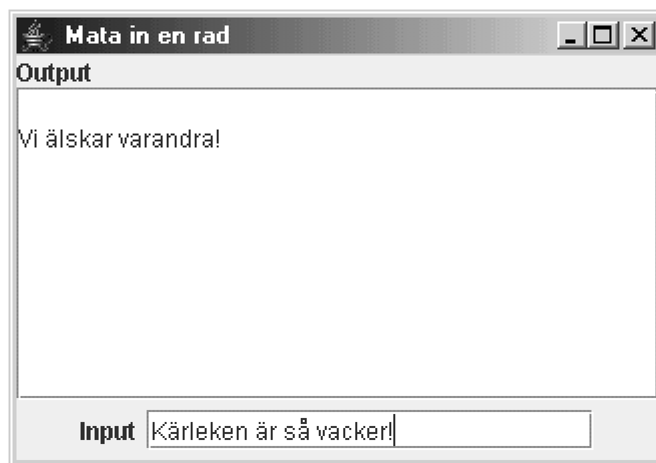
dokumentet preciserar. Men om man själv utformar sina textdokument med HTML, kan man prova och se om vissa element stöds eller inte. På så sätt kan man formulera en uppsättning (möjligtvis länkade) textdokument, som kan användas i olika Javaprogram.

Textinmatning

Under exekveringen av ett program kan olika uppgifter tillföras till det. Om programmet har ett grafiskt gränssnitt, andvänds detta gränssnitt som kommunikationskanal mellan användaren och programmet. Olika komponenter i gränssnittet kan användas för att mata in olika uppgifter.

Vissa uppgifter representeras via en eller flera teckensträngar. I dessa fall kan olika textkomponenter användas för att mata in uppgifterna. Dessa komponenter gör det möjligt för användaren att redigera en text, och för programmet att avläsa denna text vid ett lämpligt tillfälle. Programmet kan analysera den avlästa texten och bestämma olika uppgifter utifrån den.

Ett textfält av typen `javax.swing.JTextField` kan användas för att mata in en teckensträng bestående av en rad. Användaren skriver in teckensträngen i textfältet och trycker på returtangenten medan markören fortfarande är kvar i textfältet (se bilden nedan). På så sätt skapas en händelse av typen `java.awt.ActionEvent`. Händelsen kan fångas med en lyssnare av typen `java.awt.ActionListener` och hanteras i metoden `actionPerformed`. Med metoden `getText` kan texten i inmatningsfältet avläsas. Texten kan sedan analyseras och användas på olika sätt.



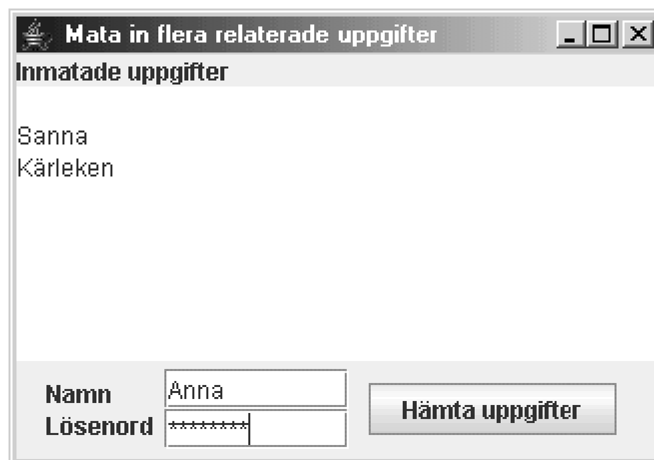
© FADIL GALJIĆ

Ett textfält kan användas som inmatningsverktyg enligt följande mönster:

```
JTextField    inFalt = new JTextField (20);
inFalt.addActionListener (new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            String    rad = inFalt.getText ();
            inFalt.setText("");

            outArea.append ("\n" + rad);
        }
    } );
```

En rad i textfältet avläses, och textfältet töms sedan. Den inmatade raden visas i en textarea (`outArea`).

Ibland kan man vilja mata in flera relaterade uppgifter samtidigt. En användares namn och lösenord, till exempel, kanske ska matas in vid samma tillfälle. I sådana fall kan olika inmatningsfält användas för de olika uppgifterna. En knapp kan användas för att alla dessa uppgifter ska kunna avläsas samtidigt. När användaren klickar på knappen, hämtas de olika uppgifterna. Dessa uppgifter används sedan på något sätt i programmet (se bilden nedan). I detta fall registreras en lyssnare hos knappen, inte hos de enskilda inmatningsfälten.

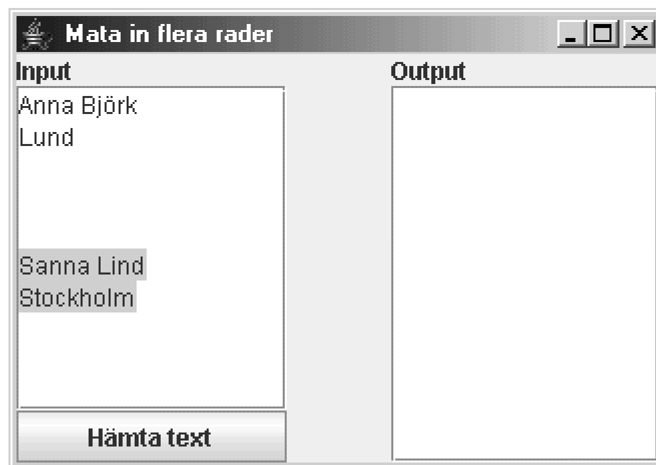


Ett textfält av typen `javax.swing.JPasswordField` (en subklass till klassen `javax.swing.JTextField`) kan användas som inmatningsfält för ett lösenord. De (maskerade) tecken som finns i ett sådant textfält avläsess (som en `char`-vektor) med metoden `getPassword`. Det kan göras så här:

© FADIL GALJIĆ

```
JPasswordField    losenordFalt = new JPasswordField (16);

// inuti en lyssnarmetod
char[]    losenordChars = losenordFalt.getPassword ();
String    losenord = new String (losenordChars);
```

Även en textarea av typen `javax.swing.JTextArea` kan användas som inmatningsverktyg. I detta fall är det möjligt att mata in en text bestående av ett godtyckligt antal rader. Användaren redigerar en text i textarean, och texten kan sedan avläsas med metoden `getText`. Avläsningsögonblicket bestäms med en annan komponent i det grafiska gränssnittet, till exempel med en knapp (se bilden nedan). En lyssnare av typen `ActionListener` kan inte användas i samband med en textarea (av typen `JTextArea`), och därför används en annan komponent för att signalera att texten ska avläsas.



Användaren kan markera en viss text i en textarea. Den markerade texten kan sedan avläsas med metoden `getSelectedText`. Om ingen text är markerad, returnerar metoden `null`.

Bevaka ändringar i ett dokument

Ett *textdokument* är en behållare för en serie tecken. Ett textdokument definieras via gränssnittet `Document` i paketet `javax.swing.text`. Det finns flera klasser i standardbiblioteket som implementerar detta gränssnitt, till exempel klasserna `javax.swing.text.PlainDocument`, `javax.swing.text.html.HTMLDocument` och `ja-`

vax.swing.text.DefaultStyledDocument. Dessa klasser representerar olika typer av dokument. Ett dokument kan innehålla enkel text, html-text eller någon annan typ av text.

En textkomponent (ett textfält, en textarea, …) kan användas för att visa och redigera ett textdokument. Via en textkomponent kan textdokumentet göras synligt, så att det blir tillgängligt för användaren. Metoden getDocument (i klassen javax.swing.text.JTextComponent) returnerar en textkomponents dokument. I samband med ett textfält som refereras av referensen inFalt, erhålls motsvarande dokument (som innehåller de tecken som finns i textfältet) så här:

```
Document    dokument = inFalt.getDocument ();
```

Man kan bevaka ändringar i ett dokument och reagera på dessa ändringar. Varje ändring i ett dokument skapar en händelse av typen javax.swing.event.DocumentEvent. En sådan händelse kan fångas med en lyssnare av typen javax.swing.event.DocumentListener. Gränssnittet DocumentListener definierar tre metoder. Metoden insertUpdate anropas när någonting läggs till i ett dokument. Metoden removeUpdate anropas när något tas bort från ett dokument. Metoden changedUpdate anropas när ett attribut i ett dokument ändras (olika attribut definierar olika egenskaper hos texten i ett dokument).

Hos ett dokument (av typen Document) registreras en lyssnare av typen DocumentListener med metoden addDocumentListener. En lyssnare kan definieras, skapas och registreras så här:

```
DocumentListener    inLyssnare = new DocumentListener ()
    {
        public void insertUpdate (DocumentEvent e)
        {
            String    text = inFalt.getText ();
            outArea.setText (text);
        }

        public void removeUpdate (DocumentEvent e)
        {
            String    text = inFalt.getText ();
            outArea.setText (text);
        }

        public void changedUpdate (DocumentEvent e)
        {}
    };

dokument.addDocumentListener (inLyssnare);
```

 © FADIL GALJIĆ

Vid varje ändring av texten i textfältet `inFalt` avläses texten. Denna text visas i en textarea (som refereras av referensen `outArea`). På så vis ändras texten i textarean på samma sätt som texten i textfältet. Texten som visas i ett textfält är en enkel text, som inte kan ha något attribut. Metoden `changedUpdate` lämnas därför tom.

Man kan bevaka ändringar av texten i ett textfält och reagera på dessa ändringar. På samma sätt kan ändringar av texten i en annan textkomponent bevakas. Man får en referens till motsvarande dokument, och registrerar en lyssnare av typen `DocumentListener` hos dokumentet. Man lyssnar på textändringar i en textkomponent genom att lyssna på ändringar av det motsvarande dokumentet (som erhålls med metoden `getDocument`).

© FADIL GALJIĆ

Val mellan olika alternativ

Valkomponenter

En vanlig funktion som ett programs användargränssnitt behöver uppfylla är att tillhandahålla olika valmöjligheter. Användaren ska kunna välja mellan olika alternativ vid olika tillfällen, och på så sätt navigera genom programmet. Genom en serie val kan ett programs flöde kontrolleras.

För att ett val ska kunna utföras via ett grafiskt användargränssnitt, placeras en lämplig grafisk komponent i gränssnittet. En sådan komponent måste innehålla ett antal *symboler*, kunna generera ett antal symboler, eller vara en symbol i sig. Varje symbol representerar ett *alternativ* i programmet. Detta innebär att användaren väljer ett visst alternativ genom att välja motsvarande symbol. När en symbol väljs, skapas en händelse av någon typ. Man kan fånga en sådan händelse och aktivera motsvarande *handling* i programmet. Programmet kan till exempel innehålla en lista med namn på flera olika färger. Dessa namn är symboler som representerar olika alternativ i ett program. Ett namn kan representera en lämplig färg på en figur. Att ha en figur i olika färger innebär att det finns olika alternativ i ett program. Genom att välja ett namn, aktiverar användaren motsvarande handling (den motsvarande koden anropas) som sätter figurens färg till den valda färgen. Användaren väljer ett alternativ genom att välja motsvarande symbol.

I Javas standardbibliotek finns ett antal grafiska komponenter som möjliggör olika val. Dessa komponenter kan indelas i flera grupper. En grupp utgörs av de komponenter som innehåller ett antal symboler. Användaren kan på något sätt komma åt en symbol och välja denna. Både teckensträngar och bilder kan användas som symboler i sådana komponenter.

En annan grupp komponenter som används i samband med val är de komponenter som kan befinna sig i två olika lägen. Ett alternativ i ett program symboliseras med en sådan komponent. Alternativet väljs genom att motsvarande komponent sätts i på-läge. Med flera tvålägeskomponenter kan flera alternativ i ett program symboliseras.

I vissa fall väljer användaren en symbol från en följd av symboler. Symbolerna skapas med en symbolgenerator. En sådan generator har vanligtvis två knappar och en display. På displayen visas generatorns aktuella värde. Den ena knappen kan användas för att generera nästa värde i följden, och den andra knappen för att generera det föregående värdet i följden. Sym-

 © FADIL GALJIĆ

bolerna lagras inte, utan genereras efter behov. Genom att manipulera symbolgeneratorn, väljer användaren olika alternativ.

Komponenter med två lägen

En tvålägeskomponent

Ett användargränssnitt kan ha en komponent som kan befinna sig i två olika lägen. Dessa lägen kan kallas för på och av (on / off). Ett alternativ i programmet kan knytas till en sådan *tvålägeskomponent*. Komponenten blir då en symbol för alternativet. Alternativet väljs genom att komponenten sätts i på-läge.

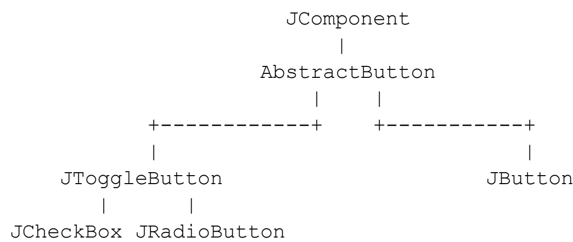
En tvålägeskomponent kan utformas så att båda lägena är stabila. När ett läge väljs, förblir komponenten i detta läge så länge användaren inte ändrar det. Sådana komponenter kan kallas för *bistabila tvålägeskomponenter*. En tvålägeskomponent kan även utformas så att bara ett av två möjliga lägen är stabilt. Det kan till exempel vara en komponent med stabilt avläge. Komponentens på-läge kan nås kortvarigt, men komponenten kommer automatiskt tillbaka till sitt av-läge. Sådana komponenter kan kallas för *monostabila tvålägeskomponenter*.

En bistabil komponent kan användas för att aktivera eller inaktivera ett alternativ i programmet. Så länge komponenten är på är alternativet aktivt. Programmet kan till exempel innehålla en bistabil komponent som symbol för fet stil i en text. Stilen är i så fall fet så länge komponenten är på. Monostabila komponenter, å andra sidan, används för att aktivera en handling. Komponentens läge kan bytas kortvarigt, och på så sätt kan en process initieras. Processen följer sedan sina egna regler, oberoende av komponenten. Det kan röra sig om att öppna eller spara en fil, skriva ut en sida, ändra en färg och så vidare.

En tvålägeskomponent representerar ett alternativ i ett program (är symbol för alternativet, står för alternativet). En grupp med sådana komponenter kan användas för att representera flera relaterade alternativ. I detta fall kan två olika strategier användas. Man kan tillåta att endast ett alternativ åt gången kan väljas. En annan strategi är att tillåta användaren att välja flera alternativ från en uppsättning alternativ. I detta fall kan olika alternativ kombineras.

En tvålägeskomponent definieras i Javas standardbibliotek i klassen `javax.swing.AbstractButton`. Det är en abstrakt klass med flera subklasser. Klassen `javax.swing.JButton` är en direkt subklass till klassen `Abstract-`

`Button`, som representerar en monostabil tvålägeskomponent. En komponent av typen `JButton` kallas vanligtvis för en *knapp*. En annan subklass till klassen `AbstractButton` är klassen `javax.swing.JToggleButton`. Denna klass representerar en komponent med två stabila lägen. Olika designmönster kan användas för att symbolisera en komponents olika lägen. För att tillhandahålla sådana möjligheter, har två subklasser till klassen `JToggleButton` definierats. Dessa subklasser är `javax.swing.JCheckBox` och `javax.swing.JRadioButton`. Den motsvarande klasshierarkin kan representeras så här.

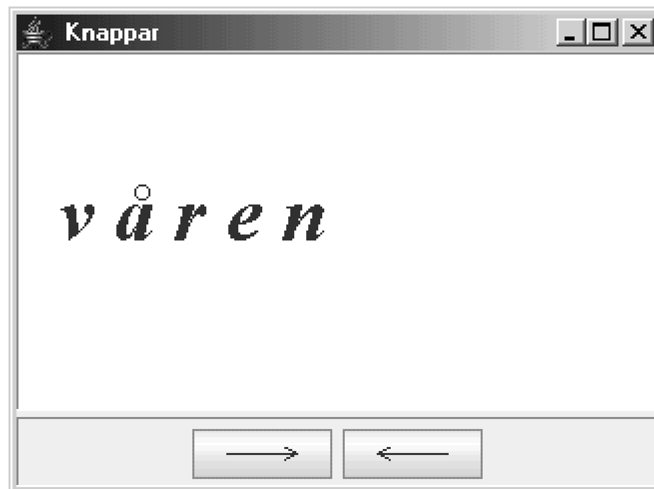


Klassen `AbstractButton` definierar flera metoder som är gemensamma för alla tvålägeskomponenter. Bland annat definieras metoderna `setText` och `setIcon`, som bestämmer en komponents text och ikon. Dessutom definieras metoderna `setVerticalAlignment` och `setHorizontalAlignment`, som preciserar en texts eller en ikons position i komponenten.

Knappar

En knapp kan användas för att representera ett alternativ i ett program. Knappen är då en symbol för alternativet. Användaren väljer alternativet genom att klicka på knappen. Motsvarande kod anropas, och på så sätt startas en handling. Handlingen fortgår sedan oberoende av knappen. Ett grafiskt användargränssnitt kan innehålla flera knappar (se bilden nedan). Varje knapp representerar (symboliserar, står för) i så fall ett särskilt alternativ (en viss handling). Användaren kan kontrollera programmet genom att använda dessa knappar.

 © FADIL GALJIĆ



En knapp kan representeras med ett objekt av typen `javax.swing.JButton`:

```
JButton    knapp = new JButton ("Avsluta");
```

Koden skapar en knapp med texten `Avsluta` på. När knappen trycks, skapas en händelse av typen `java.awt.event.ActionEvent`. Denna händelse kan fångas med en lyssnare av typen `java.awt.event.ActionListener`. Metoden `actionPerformed` implementeras, och koden som ska utföras när knappen trycks placeras i metoden. Lyssnaren registreras sedan hos knappen. Så här kan det gå till:

```
ActionListener    lyssnare = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            System.exit (0);
        }
    };

knapp.addActionListener (lyssnare);
```

En handling (ett alternativ) definieras i metoden `actionPerformed`, och handlingen knyts till knappen. Knappen står för (symboliserar) handlingen. Genom att trycka på knappen kan användaren välja motsvarande alternativ i programmet.

För att förtydliga en knapps funktion, placeras normalt en text på knappen. Texten anges som argument till motsvarande konstruktor när knappen skapas. Texten kan även anges (eller ändras) efter det att knappen

skapats. Till detta används metoden `setText`, och den motsvarande teckensträngen tillförs som argument till metoden.

En knapp kan även ha en bild. Detta kan åstadkommas genom att en ikon placeras på knappen. En ikon anges antingen som argument till motsvarande konstruktor, eller som argument till metoden `setIcon`. Det går även att kombinera en text och en ikon på en knapp.

En knapp kan även dekoreras direkt, utan att en ikon används. Man kan rita på en knapp (det går att rita på alla lättviktiga Swing-komponenter). En subklass till klassen `JButton` skapas, och klassens `paintComponent`-metod omdefinieras. I denna metod definieras den bild som ska visas på en knapp av denna typ. Detta kan till exempel göras så här:

```
class ArrowButton extends JButton
{
    public void paintComponent (Graphics gr)
    {
        super.paintComponent (gr);

        Graphics2D    g = (Graphics2D) gr;

        int    w = this.getWidth ();
        int    h = this.getHeight ();

        int    x1 = 25 * w / 100;
        int    y1 = h / 2;
        int    x2 = 75 * w / 100;
        int    y2 = h / 2;
        Line2D    line = new Line2D.Double (x1, y1, x2, y2);
        Line2D    linea = new Line2D.Double (x2 - 6, y2 - 3,
                                             x2, y2);
        Line2D    lineb = new Line2D.Double (x2 - 6, y2 + 3,
                                             x2, y2);

        g.draw (line);
        g.draw (linea);
        g.draw (lineb);
    }
}
```

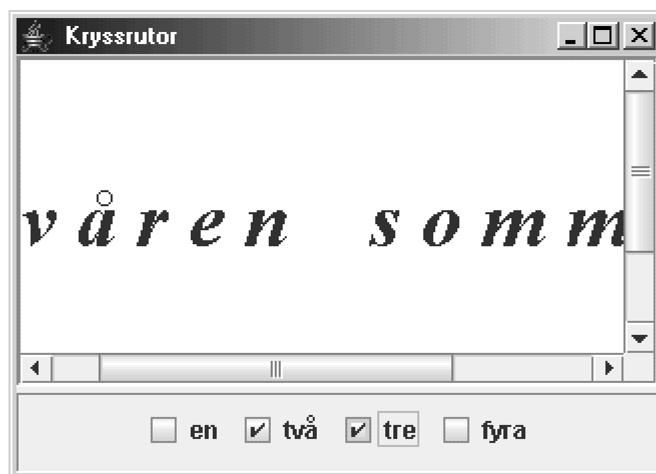
Ett objekt av typen `ArrowButton` representerar en knapp med en högerriktad pil. Klassen `ArrowButton` kan utvidgas, så att det även blir möjligt att välja pilens riktning. En pil är en lämplig symbol i flera olika sammanhang, och klassen `ArrowButton` är därför en användbar klass.

 © FADIL GALJIĆ

Kryssrutor

En *kryssruta* (eng. checkbox) är en ruta som kan befinna sig i två olika tillstånd: den kan vara markerad eller inte markerad. De båda tillstånden är stabila. En kryssrutas tillstånd ändras genom att man klickar i den med musen.

En kryssruta kan användas för att symbolisera ett alternativ i ett program. Alternativet väljs genom att kryssrutan markeras. Alternativet gäller så länge kryssrutan är markerad. För att representera flera relaterade alternativ, kan en uppsättning kryssrutor användas (se bilden nedan). Användaren kan välja inget, ett eller flera alternativ i uppsättningen.



En kryssruta kan representeras med ett objekt av typen `javax.swing.JCheckBox`. Två kryssrutor kan skapas så här:

```
JCheckBox    cb1 = new JCheckBox ("en");
JCheckBox    cb2 = new JCheckBox ("två");
```

Här skapas en kryssruta med texten `en` och en kryssruta med texten `två`. Ingen, en eller båda kryssrutorna kan markeras. En kryssrutas tillstånd kontrolleras med metoden `isSelected`. Metoden returnerar `true` om den aktuella kryssrutan är markerad. En kryssruta kan markeras via koden med metoden `setSelected` (`true` anges som argument).

När en kryssrutas tillstånd ändras, skapas en händelse av typen `java.awt.event.ActionEvent`. Händelsen kan fångas via en lyssnare av typen `java.awt.event.ActionListener`. Man kan ha en gemensam lyssnare för flera kryssrutor. Då kan man testa tillståndet för var och en av

© FADIL GALJIĆ

kryssrutorna, och utifrån detta bestämma nästa steg. Detta illustreras med följande exempel:

```
ActionListener    lyssnare = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            String    s = "";
            if (cb1.isSelected ())
                s += " v i n t e r n";
            if (cb2.isSelected ())
                s += " v å r e n";

            textArea.setText (s);
        }
    };

cb1.addActionListener (lyssnare);
cb2.addActionListener (lyssnare);
```

En teckensträng bestäms, och visas i en textarea. Teckensträngen beror på vilka kryssrutor som är markerade. Den avspeglar tillståndet för samtliga kryssrutor.

Radioknappar

En knapp av typen `javax.swing.JRadioButton` är en knapp som kan befinna sig i två olika lägen: antingen på eller av. En knapp av typen `JRadioButton` kan därför användas på samma sätt som en kryssruta av typen `JCheckBox` (eller som en knapp av deras gemensamma superklass `JToggleButton`). Men så brukar man inte göra. Vanligtvis grupperar man flera knappar av typen `JRadioButton` och använder dessa knappar som radioknappar (se bilden nedan). Detta innebär att bara en knapp i taget kan vara på. När en knapp sätts på, blir alla andra knappar automatiskt av. På så sätt kan ett (och endast ett) av flera tillgängliga alternativ väljas.

 © FADIL GALJIĆ

En grupp radioknappar kan skapas med hjälp av klassen `javax.swing.ButtonGroup`. Till exempel så här:

```
JRadioButton    rb1 = new JRadioButton ("en");
JRadioButton    rb2 = new JRadioButton ("två");
ButtonGroup    bg = new ButtonGroup ();
bg.add (rb1);
bg.add (rb2);
```

När en radioknapp trycks ner, skapas en händelse av typen `java.awt.event.ActionEvent`. Denna händelse kan fångas med en lyssnare av typen `java.awt.event.ActionListener`. Man kan ha en gemensam lyssnare för alla radioknappar i en grupp (det går även att ha separata lyssnare). Då kan man testa vilken radioknapp som är nedtryckt, och utifrån detta bestämma nästa steg. Man kan till exempel göra så här:

```
ActionListener    lyssnare = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            String    s = "";
            if (rb1.isSelected ())
                s = " v i n t e r n";
            if (rb2.isSelected ())
                s = " v å r e n";

            textArea.setText (s);
        }
    };

rb1.addActionListener (lyssnare);
```

```
rb2.addActionListener (lyssnare);
```

En teckensträng bestäms, och visas sedan i en textarea. Vilken teckensträng som visas beror på vilken av radioknapparna som valts. Teckensträngen avspeglar den valda knappen.

Symbolbehållare

En valruta med en fast uppsättning symboler

Flera symboler kan lagras i en ruta – en *valruta*. Denna valruta kan sedan öppnas, och en av symbolerna kan väljas. Genom att välja en av dessa symboler, väljer användaren motsvarande alternativ i programmet.

En valruta kan representeras med ett objekt av klassen `javax.swing.JComboBox` (en direkt subklass till klassen `javax.swing.JComponent`). En sådan valruta innehåller en display, en knapp och ett antal symboler. Den valda symbolen visas i valrutans display. Valrutan kan öppnas med en tryckning på dess knapp. När valrutan öppnas, visas en lista med dess symboler (se bilderna nedan). Användaren väljer en av symbolerna genom att klicka på den med musen. Därefter stängs valrutan, och det alternativ i programmet som motsvarar den valda symbolen (kan) aktiveras.

© FADIL GALJIĆ

En valruta av typen `JComboBox` kan skapas utifrån en vektor med symboler, till exempel så här:

```
Object[]     symboler = {" en", " två", " tre", " fyra"};
JComboBox    valRuta = new JComboBox (symboler);
```

Den symbol som anges först, visas på rutans display som förvalt värde. Den förvalda symbolen kan bestämmas ifrån programmet, antingen med metoden `setSelectedItem` (tar emot en symbol som argument), eller metoden `setSelectedIndex` (tar emot ett index som argument). Så här kan detta göras:

```
valRuta.setSelectedIndex (1);
```

När en symbol väljs i valrutan, skapas en händelse av typen `java.awt.event.ActionEvent`. Denna händelse kan fångas i metoden `actionPerformed` i en lyssnare av typen `java.awt.event.ActionListener`. Metoden tar reda på den valda symbolen, och utifrån detta kan ett visst alternativ väljas. Det går att göra så här:

```
final String[]    alternativ = {"v i n t e r n", "v å r e n",
                              "s o m m a r e n", "h ö s t e n"};

ActionListener    lyssnare = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
```

© FADIL GALJIĆ

```
        {
            JComboBox    source = (JComboBox) e.getSource ();

            int    valdaIndexet = source.getSelectedIndex ();

            textArea.setText ("\n  " + alternativ[valdaIndexet]);
        }
    };

valRuta.addActionListener (lyssnare);
```

Metoden `getSelectedIndex` ger index för den valda symbolen (för att få själva symbolen används metoden `getSelectedItem`). Sedan visas den teckensträng som motsvarar symbolen i en textarea.

En flexibel valruta

En valruta kan innehålla en fast uppsättning symboler, vilket inte ger användaren möjlighet att själv påverka uppsättningen. Men det går att skapa mer flexibla valrutor. Man kan till exempel tillåta användaren att mata in en egen symbol via valrutan. I detta fall blir den symbolen den som väljs i valrutan. Vid olika val kan olika symboler anges. På så vis skapas en större flexibilitet. Förutom de symboler som redan finns i valrutan, kan användaren ange en egen symbol.

En valruta av typen `JComboBox` kan göras till en flexibel valruta med metoden `setEditable` (som anropas i samband med valrutan). Argumentet `true` tillförs till metoden. Därmed kan valrutans display även användas som inmatningsfält. Användaren skriver in en egen symbol i fältet och trycker på returtangenten (se bilden nedan). På så sätt väljs användarens symbol istället för någon av de befintliga symbolerna.

© FADIL GALJIĆ

En valruta kan göras ännu mer flexibel genom att möjlighet att ändra innehållet i den implementeras. Användaren kan då lägga till en ny symbol i valrutan, eller ta bort en befintlig symbol. Valrutan blir föränderlig. Det går till exempel att mata in en ny symbol via en valruta, och lägga till denna symbol i valrutan.

En valruta av typen `JComboBox` är (i förvalt läge) en föränderlig valruta. I en sådan valruta läggs en ny symbol till med metoden `addItem`. En symbol tas bort från valrutan med metoden `removeItem` (metoden tar en symbol som argument), eller med metoden `removeItemAt` (metoden tar ett index som argument). Med metoden `removeAllItems` kan alla symboler tas bort från en valruta av typen `JComboBox`. Det går att göra så här:

```
String[]    symboler = {" 30", " 35", " 40", " 45"};
JComboBox    valRuta = new JComboBox (symboler);
valRuta.setEditable (true);

ActionListener    lyssnare = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            JComboBox    source = (JComboBox) e.getSource ();

            String    symbol = (String) source.getSelectedItem ();

            int    valdaIndexet = source.getSelectedIndex ();
            if (valdaIndexet < 0)
                source.addItem (symbol);

            int    size = Integer.parseInt (symbol.trim ());
```

```
            textArea.setFont (
              new Font ("Serif", Font.BOLD + Font.ITALIC, size));
        }
    };

valRuta.addActionListener (lyssnare);
```

Här skapas en valruta som gör det möjligt för användaren att välja storleken på det som visas i en textarea. Flera förvalda symboler (storlekar) placeras i valrutan, men det finns också möjlighet att mata in nya symboler. När användaren gör ett val bestäms storleken på texten, motsvarande font skapas och sätts att användas i textarean. På så sätt ändras storleken på det som visas i textarean.

Lyssnaren kontrollerar om användaren matat in en ny symbol genom att bestämma den valda symbolens index. Om indexet är mindre än noll, betyder det att symbolen inte finns i valrutan. Det är en ny symbol, och därför läggs den till i valrutan (detta är dock inte nödvändigt – en ny symbol kan matas in utan att läggas till i valrutan). Den nya symbolen hanteras i fortsättningen på samma sätt som alla andra symboler i valrutan.

En lista med symboler

En uppsättning symboler kan ordnas i en ruta, som kan användas som komponent i ett grafiskt användargränssnitt. En sådan ruta med symboler kallas för en *lista* (en vallista, se bilden nedan). En symbol i en lista kan definieras via en teckensträng eller via en bild. Användaren väljer en symbol i listan genom att klicka med musen på symbolen. Det går även att välja flera symboler vid ett och samma tillfälle. Symbolerna i en lista kan knytas till ett antal alternativ i programmet. Användaren kan i så fall välja en uppsättning alternativ genom att välja en uppsättning symboler i listan.

© FADIL GALJIĆ

En lista representeras i Java med ett objekt av typen `javax.swing.JList` (en direkt subklass till klassen `javax.swing.JComponent`). Ett sådant objekt kan skapas utifrån en vektor med symboler, på följande sätt:

```
Object[]    symboler = {" ett", " två", " tre", " fyra", " fem",
                        " sex", " sju", " åtta", " nio", " tio"};
JList    list = new JList (symboler);
```

Normalt kan en eller flera symboler väljas från en lista. För att välja flera symboler använder man `Ctrl`-tangenten, som hålls nedtryckt medan man väljer. `Shift`-tangenten kan användas för att välja ett intervall med symboler. Man markerar den första symbolen i intervallet, trycker ner `Shift`-tangenten, och markerar sedan den sista symbolen i intervallet. På så sätt väljs samtliga symboler i intervallet.

Det går att sätta restriktioner för valmöjligheter i samband med en lista. Valmöjligheterna kan till exempel begränsar så att användaren bara kan välja en symbol i taget. Detta görs så här:

```
list.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
```

Konstanten `SINGLE_SELECTION` (i klassen `javax.swing.ListSelectionModel`) används som argument till metoden `setSelectionMode`. Även konstanten `SINGLE_INTERVAL_SELECTION` kan användas som argument till denna metod. I detta fall begränsas valmöjligheterna till ett intervall. Med konstanten `MULTIPLE_INTERVAL_SELECTION` (förvalt värde) tillåts användaren välja en godtycklig uppsättning symboler.

Ett val från en lista skapar en händelse av typen `javax.swing.event.ListSelectionEvent`. En sådan händelse fångas i metoden `valueChanged`, i en lyssnare av typen `javax.swing.event.ListSelectionListener`. Så här kan detta göras:

```
list.addListSelectionListener (new ListSelectionListener ()
    {
        public void valueChanged (ListSelectionEvent e)
        {
            JList    source = (JList) e.getSource ();

            Object[]  valdaSymboler = list.getSelectedValues ();

            String    val = "";
            for (int i = 0; i < valdaSymboler.length; i++)
                val = val + " " + valdaSymboler[i] + "\n";

            textArea.setText (val);
        }
    } );
```

De valda symbolerna bestäms, och en teckensträng skapas utifrån dessa symboler. Sedan visas teckensträngen i en textarea.

De valda symbolerna erhålls antingen med metoden `getSelectedValue` (returnerar den symbol som valts först, eller `null` om ingenting valts), eller med metoden `getSelectedValues` (returnerar alla valda symboler). Symbolerna returneras som objekt av typen `java.lang.Object`. Det går även att använda metoderna `getSelectedIndex` och `getSelectedIndices`, som returnerar index istället för symboler. Ett val inifrån ett program kan framtvingas med metoderna `setSelectedIndex` och `setSelectedIndices`.

Storleken på en cell i en lista kan anges med metoderna `setFixedCellWidth` och `setFixedCellHeight`. Färgerna i en lista kan anges med metoderna `setBackground`, `setForeground`, `setSelectionBackground` och `setSelectionForeground`.

En redigerbar lista

Det går att skapa en lista vars innehåll kan ändras. Det innebär att olika symboler kan läggas till, infogas och tas bort under programmets gång. Listans innehåll kan anpassas till de aktuella behoven.

För att skapa en redigerbar lista kan man utgå ifrån ett objekt av typen `javax.swing.DefaultListModel`. Ett objekt av denna typ representerar ett slags behållare för olika objekt. Med metoden `addElement` kan ett objekt

© FADIL GALJIĆ

läggas till i en sådan behållare. Med någon av metoderna `remove` och `removeElement` kan ett element tas bort från en behållare. Det går även att använda metoderna `add`, `set`, `get`, `clear`, `size` och så vidare.

Man kan skapa en behållare av typen `DefaultListModel`, och placera flera teckensträngar i behållaren, på följande sätt:

```
DefaultListModel    listModel = new DefaultListModel ();
listModel.addElement (new String ("helheten"));
listModel.addElement (new String ("sanningen"));
listModel.addElement (new String ("balansen"));
listModel.addElement (new String ("kärleken"));
```

En lista kan skapas utifrån den här behållaren så här:

```
JList    list = new JList (listModel);
```

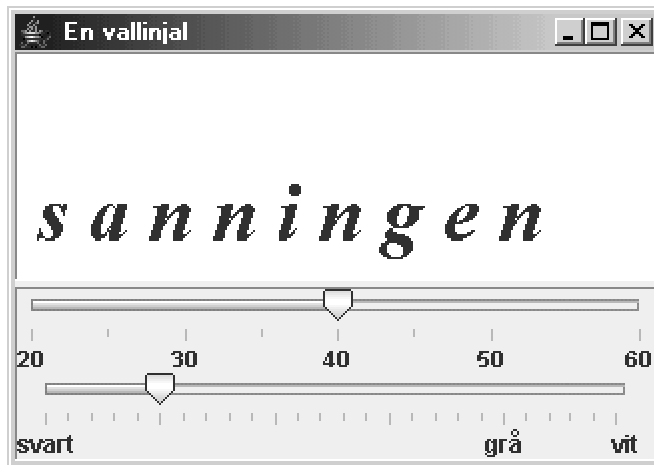
Behållaren `listModel` är en bas, en modell på vilken listan `list` baseras. Listan kan kontrolleras via denna listmodell. När ett objekt läggs till i listmodellen, läggs motsvarande symbol till i listan. Ändringen blir synlig omedelbart. På samma sätt kan enskilda symboler tas bort från listan.

En lista med bilder

Istället för teckensträngar kan bilder användas som symboler i en lista. En sådan lista kan skapas utifrån en vektor med ikoner. En ikon kan representeras med ett objekt av klassen `javax.swing.ImageIcon`. Man kan skapa en sådan ikon utifrån en fil som innehåller en bild (filens namn anges som argument till motsvarande konstruktor). En annan möjlighet är att representera en bild med ett objekt av typen `java.awt.image.BufferedImage`, och skapa en ikon utifrån detta objekt.

En vallinjal

En vallinjal är en grafisk komponent som innehåller ett intervall med symboler och en flyttbar "punkt". Användaren väljer olika symboler genom att flytta punkten längs vallinjalen. Den symbol som den flyttbara punkten pekar på är den valda symbolen (se bilden nedan). Olika symboler på vallinjalen kan knytas till olika alternativ i ett program. Användaren väljer i så fall ett alternativ genom att välja motsvarande symbol på linjalen.



En vallinjal representeras med ett objekt av typen `javax.swing.JSlider` (en direkt subklass till klassen `javax.swing.JComponent`). Det går att skapa en vallinjal utifrån olika argument, till exempel så här:

```
JSlider    linjal = new JSlider (20, 60, 40);
```

Här skapas en horisontell vallinjal, som representerar heltal i ett intervall. Intervallets minsta värde är `20`, dess största värde är `60` och det valda värdet är `40`. Även en vertikal vallinjal kan skapas. Linjalens läge anges med ett argument, på följande vis:

```
JSlider    linjal = new JSlider (JSlider.VERTICAL, 20, 60, 40);
```

Varje gång den flyttbara punktens position på linjalen ändras, skapas en händelse av typen `javax.swing.event.ChangeEvent`. Denna händelse kan fångas i en lyssnare av typen `javax.swing.event.ChangeListener`, till exempel så här:

```
linjal.addChangeListener (new ChangeListener ()
    {
        public void stateChanged (ChangeEvent e)
        {
            int    vardet = linjal.getValue ();

            Font    font = new Font (
                 "Serif", Font.BOLD + Font.ITALIC, vardet);
            textArea.setFont (font);
        }
    } );
```


© FADIL GALJIĆ

Det valda värdet bestäms med metoden getValue. Värdet används sedan för att justera storleken på en text i en textarea.

När linjalens punkt flyttas, skapas en serie händelser av typen ChangeEvent. Det kan dock hända att man bara är intresserad av det slutliga valet. Med metoden getValueIsAdjusting kan man i så fall ta reda på om punkten på linjalen flyttas eller inte. Metoden returnerar true så länge linjalens värde ändras. Metoden kan anropas i metoden stateChanged enligt följande mönster:

```
if (!(linjal.getValueIsAdjusting ())
{
    int    vardet = linjal.getValue ();

    // använd den valda symbolen
}
```

En vallinjals egenskaper kan bestämmas med ett antal metoder i klassen JSlider. Metoderna setMinimum, setMaximum och setValue anger en vallinjals minimivärde, maximivärde och valda värde. Med metoden setPaintTicks anges om linjalens streck ska vara synliga. För att göra strecken synliga anges true som argument till metoden. Med metoden setPaintLabels görs en vallinjals märken synliga. Avståndet mellan linjalens långa streck anges med metoden setMajorTickSpacing, och avståndet mellan de korta strecken med metoden setMinorTickSpacing. Man kan till exempel göra så här:

```
linjal.setMajorTickSpacing (10);
linjal.setMinorTickSpacing (5);
linjal.setPaintTicks (true);
linjal.setPaintLabels (true);
```

Avståndet mellan de långa strecken sätts till 10, och avståndet mellan de korta strecken till 5. Både linjalens streck och märken görs synliga.

Normalt används heltal som märken på en vallinjal. Men även teckensträngar och bilder kan användas som märken. Objekt av typen javax.swing.JComponent används för att representera dessa märken. Det gåt till exempel att använda etiketter (objekt av typen javax.swing.JLabel) som märken på en vallinjal. Ett antal etiketter skapas, och lagras sedan i en hashtabell. Etiketterna sätts sedan som märken på en vallinjal med metoden setLabelTable. Detta kan till exempel göras så här:

```
Hashtable<Integer, JComponent>    etiketter =
                new Hashtable<Integer, JComponent> ();
etiketter.put (new Integer (0), new JLabel ("svart"));
etiketter.put (new Integer (200), new JLabel ("grå"));
etiketter.put (new Integer (255), new JLabel ("vit"));
```

© FADIL GALJIĆ

```
linjal.setLabelTable (etiketter);
```

Vallinjalen får här teckensträngarna `svart`, `grå` och `vit` som märken. Dessa märken finns på positionerna `0`, `200` och `255`.

Symbolgeneratorer

Generera en följd av symboler

Det kan hända att man behöver välja mellan ett stort antal alternativ. Det kan till exempel gälla att välja ett kalenderår mellan år 1900 och 2100. Det kan även vara frågan om en obegränsad följd av alternativ. Motsvarande symboler behöver i detta fall inte lagras, utan kan genereras efter behov. En *symbolgenerator* kan skapas, som genererar en följd av symboler. En sådan generator har vanligtvis en display och två knappar. Den valda symbolen visas i generatorns display. Via generatorns knappar kan nästa eller föregående symbol från den givna följden genereras. Den ena av dessa två knappar har vanligtvis en uppåtriktad pil (nästa symbol i följden), och den andra knappen en nedåtriktad pil (föregående symbol i följden). När användaren trycker på en av dessa två knappar, genereras motsvarande symbol och visas i generatorns display (se bilden nedan). Så länge knappen hålls nedtryckt, genereras och visas symboler från följden. Symbolerna lagras inte, utan genereras när de behövs.

© FADIL GALJIĆ

En symbolgenerator kan representeras med ett objekt av typen `javax.swing.JSpinner` (en direkt subklass till klassen `javax.swing.JComponent`). Ett sådant objekt skapas utifrån en modell som definierar motsvarande följd av symboler. Modellen måste implementera gränssnittet `javax.swing.SpinnerModel`. Standardklassen `javax.swing.AbstractSpinnerModel` (en direkt subklass till klassen `java.lang.Object`) är en abstrakt klass, som (delvis) implementerar detta gränssnitt. Klasserna `javax.swing.SpinnerNumberModel`, `javax.swing.SpinnerListModel` och `javax.swing.SpinnerDateModel` är tre direkta subklasser till klassen `AbstractSpinnerModel`. Dessa klasser kan användas för att definiera olika följder av symboler. Klassen `SpinnerNumberModel` representerar en följd av tal. Klassen `SpinnerListModel` representerar en följd av symboler av en godtycklig typ. Klassen `SpinnerDateModel` representerar en följd av datum.

En modell som representerar en följd av tal mellan 1900 och 2100 kan skapas så här:

```
SpinnerModell   arModell =
             new SpinnerNumberModel (2005, 1900, 2100, 1);
```

`2005` väljs som den symbol som visas initialt (startvärdet). Användaren kan välja mellan `1900` och `2100` med steg `1`.

Med en modell av typen `SpinnerModel` kan motsvarande symbolgenerator skapas. En symbolgenerator är en grafisk komponent, som baseras på en sådan modell. Modellen tillförs som argument när en symbolgenerator skapas. Så här kan detta göras:

```
JSpinner    arGenerator = new JSpinner (arModell);
```

Klassen `SpinnerListModel` kan användas för att definiera en följd av symboler av godtyckliga typer. Klassen kan till exempel användas för att definiera en följd av teckensträngar, på följande vis:

```
String[]    manader = {"Januari", "Februari", "Mars",
                "April", "Maj", "Juni", "Juli", "Augusti",
                "September", "Oktober", "November", "December"};

SpinnerModel    manadModell = new SpinnerListModel (manader);
```

Om en standardmodell används vid skapandet av en symbolgenerator, kan ett värde från den motsvarande följden matas in via generatorns editor (som i så fall är både en display och ett inmatningsfält).

När en symbol väljs via en symbolgenerator, skapas en händelse av typen `javax.swing.event.ChangeEvent`. Man kan fånga en sådan händelse med

en lyssnare av typen `javax.swing.event.ChangeListener` och reagera på den. Till exempel så här:

```
arGenerator.addChangeListener (new ChangeListener ()
    {
        public void stateChanged (ChangeEvent e)
        {
            Integer    ar = (Integer) arGenerator.getValue ();

            textArea.setText ("\n  " + ar);
        }
    } );
```

Den valda symbolen erhålls med metoden `getValue` (i klassen `JSpinner`). Symbolen används sedan för att aktivera det motsvarande alternativet i programmet.

Ibland används en grupp relaterade symbolgeneratorer. Ett datum, till exempel, kan genereras med hjälp av tre symbolgeneratorer. En generator genererar dagar, en annan månader och en tredje genererar år. Programmet behöver i detta fall endast reagera på användarens val när användaren valt alla tre värdena. Detta kan åstadkommas genom att en knapp läggs till i användargränssnittet. Via denna knapp kan användaren bekräfta sitt val. Programmet reagerar i så fall på en tryckning på knappen (i en lyssnare av typen `java.awt.ActionListener`), istället för att reagera varje gång tillståndet för någon av gruppens generatorer ändras.

Använda en egen modell

I vissa fall kan en följd av symboler definieras med en standardklass. Följden 2, 4, 6, 8, 10, 12, …, 100, till exempel, kan definieras med en standardmodell av typen `SpinnerNumberModel`. Det kan dock hända att man vill använda en följd av symboler, som inte kan representeras av en standardmodell. En följd av primtal (2, 3, 5, 7, 11, 13, 17, …), till exempel, kan inte representeras med en standardmodell. I detta fall måste man definiera en egen modell för symbolgenerering.

En modell för symbolgenerering definieras i en klass som implementerar gränssnittet `SpinnerModel`. Klassen `AbstractSpinnerModel` implementerar en del av de metoder som definieras i detta gränssnitt. Dessa metoder gäller händelsehanteringen, och behöver inte implementeras på nytt. Därför skapas vanligtvis en modellklass som subklass till klassen `AbstractSpinnerModel`. I detta fall behöver metoderna `getValue`, `setValue`, `getNextValue` och `getPreviousValue` definieras.

 © FADIL GALJIĆ

En modell för symbolgenerering kan definieras enligt följande mönster:

```
class ArModell extends AbstractSpinnerModel
{
    public static int    MIN_AR = 1900;
    public static int    MAX_AR = 2100;

    private Integer    ar;


    public ArModell (Integer ar)
    {
        if (ar < MIN_AR  ||  ar > MAX_AR)
            throw new IllegalArgumentException ();

        this.ar = ar;
    }

    public Object getValue ()
    {
        return ar;
    }

    public void setValue (Object ar)
    {
        if (!(ar instanceof Integer))
            throw new IllegalArgumentException ();

        if (ar < MIN_AR  ||  ar > MAX_AR)
            throw new IllegalArgumentException ();

        this.ar = (Integer) ar;

        this.fireStateChanged ();
    }

    public Object getNextValue ()
    {
        Integer    next = ar;

        if (ar < MAX_AR)
            next = ar + 1;

        return next;
    }

    public Object getPreviousValue ()
    {
        Integer    previous = ar;

        if (ar > MIN_AR)
```

© FADIL GALJIĆ

```
          previous = ar - 1;

        return previous;
    }
}
```

Här definieras en modell som gör det möjligt att välja ett år mellan 1900 och 2100. Det aktuella året representeras med ett objekt av typen Integer. Metoderna i gränssnittet SpinnerModel har parametrar och returvärden av typen Object, och därför är typen Integer att föredra framför den primitiva datatypen int.

En modell av typen ArModell skapas utifrån ett objekt av typen Integer. Modellens aktuella värde anges (det värde som först visas i motsvarande generator). Metoden getValue returnerar modellens aktuella värde. Metoden setValue sätter det aktuella värdet till ett givet värde. Metoden anropar metoden fireStateChanged (som implementeras i superklassen AbstractSpinnerModel), som i sin tur anropar metoden stateChanged i alla (registrerade) lyssnare av typen ChangeListener. Metoderna getNextValue och getPreviousValue preciserar hur nästa och föregående värde i följden erhålls. Dessa metoder ändrar inte det aktuella värdet. När användaren trycker på en knapp i motsvarande generator (som skapas utifrån den här modellen), anropas en av dessa två metoder. Därefter anropas generatorns metod setValue, för att ange det returnerade värdet (från getNextValue eller getPreviousValue) som generatorns aktuella värde. Metoden setValue anropas alltså automatiskt i samband med val via generatorns knappar (metoden kan naturligtvis även anropas direkt).

Det går att se ett mönster i en modellklass. Det finns ett aktuellt värde. Detta värde returneras med metoden getValue, och anges med metoden setValue. I metoden getNextValue bestäms nästa värde i följden (relativt det aktuella värdet), och i metoden getPreviousValue bestäms det föregående värdet. Det här mönstret är tillräckligt öppet för att godtyckliga följder av symboler ska kunna definieras.

© FADIL GALJIĆ

Strukturerade val

Ett verktygsfält

Ett programs användare utför olika operationer via programmets grafiska gränssnitt: öppnar och sparar filer, justerar fonter och färger, skriver ut olika dokument, och så vidare. Alla dessa operationer utförs via motsvarande grafiska komponenter. Dessa kan vara knappar, valrutor, menyer, och så vidare. För att komponenterna lättare ska kunna användas, måste de ordnas på ett lämpligt sätt i ett fönster. Komponenterna måste struktureras så att det blir lätt att hitta dem, komma åt dem och använda dem.

Ett sätt att strukturera flera komponenter är att placera dem i ett fält. På så sätt skapas ett fält med olika verktyg, ett *verktygsfält*. Ett verktygsfält innehåller vanligtvis olika tvålägeskomponenter med ikoner och valrutor, men kan även innehålla andra komponenter. Verktygsfältet placeras på en lämplig plats i fönstret, vanligtvis i dess övre del (se bilden nedan). Verktygen används sedan för att utföra olika val i ett program.

Ett verktygsfält representeras med ett objekt av typen `javax.swing.JToolBar`. Ett horisontellt verktygsfält skapas så här:

```
JToolBar    verktyg = new JToolBar ();
```

För att skapa ett vertikalt verktygsfält, anger man konstanten `JToolBar.VERTICAL` som argument till motsvarande konstruktor.

De olika verktygen placeras i verktygsfältet med metoden `add`. Med metoden `addSeparator` kan ett mellanrum skapas i verktygsfältet. Det går att göra så här:

```
JButton    oppna = new JButton ("Öppna");
JButton    spara = new JButton ("Spara");
JButton    skrivUt = new JButton ("Skriv ut");

verktyg.add (oppna);
verktyg.add (spara);
verktyg.addSeparator ();
verktyg.add (skrivUt);
```

Ett verktyg i ett verktygsfält har ofta en bild som symboliserar dess funktion. Denna funktion kan förtydligas ytterligare genom att en hjälptext läggs till, till exempel så här:

```
oppna.setToolTipText ("Öppna");
```

När muspekaren placeras över knappen `oppna`, visas efter ett tag den angivna texten (`Öppna`).

Ibland vill man införa lite större avstånd mellan två grupper av verktyg inom ett verktygsfält. Detta kan åstadkommas genom att en panel med en angiven storlek placeras i verktygsfältet (metoden `setPreferredSize` används för att precisera panelens storlek). Ett större avstånd kan även införas på detta vis:

```
verktyg.add (Box.createGlue ());
```

För att skapa en osynlig komponent anropas metoden `createGlue` i klassen `javax.swing.Box`. Den osynliga komponentens dimensioner anpassas automatiskt till de andra komponenterna i verktygsfältet.

Vanligtvis placeras ett verktygsfält överst i ett fönster, eller vid en annan kant i fönstret. Ett verktygsfält kan flyttas manuellt från en plats till en annan. För att göra verktygsfältet oflyttbart, anropas metoden `setFloatable` i samband med verktygsfältet med argumentet `false`.

Menyer

En meny

Ett alternativ i ett program kan representeras med en komponent som kan befinna sig i två lägen. Genom att komponenten sätts i på-läge (som kan vara stabilt eller icke-stabilt), väljs motsvarande alternativ i programmet.

 © FADIL GALJIĆ

För att representera olika alternativ i ett program kan flera tvålägeskomponenter användas. Komponenterna kan ordnas i ett verktygsfält, som placeras överst i ett fönster. På så sätt görs komponenterna lättåtkomliga för användaren. Användaren kan lätt hitta en komponent och aktivera motsvarande alternativ.

Strategin med verktygsfältet lämpar sig väl så länge antalet komponenter inte är för stort. Om ett program innehåller många alternativ, måste en annan strategi användas för att strukturera de olika valkomponenterna. Komponenterna måste döljas på något sätt, så att de inte är synliga hela tiden. Det måste i detta fall finnas en komponent som symboliserar ett antal andra komponenter, och som representerar en ingång för dessa komponenter. Genom denna symbolkomponent kan användaren komma åt en viss valkomponent och utföra val. Flera sådana symbolkomponenter kan organiseras i en rad. Dessa komponenter ska vara synliga hela tiden. Via symbolkomponenterna ska det gå att komma åt och använda ett stort antal andra komponenter.

Flera relaterade tvålägeskomponenter kan grupperas och döljas bakom ett symboliskt namn. På så sätt bildas en *meny*. Genom att välja detta namn (användaren klickar på namnet), öppnas menyn och användaren får tillgång till alla de valkomponenter som finns i menyn, och kan fritt välja mellan dessa komponenter. Flera menyer kan placeras i ett fält, en så kallad *menyrad*. Via denna menyrad kan användaren komma åt ett stort antal valkomponenter, och kontrollera ett programs flöde via dessa komponenter (se bilden nedan). En menyrad med flera menyer är ett överlägset sätt att strukturera val i ett program med många alternativ.

© FADIL GALJIĆ

En valkomponent i en meny kan representeras med ett objekt av typen `javax.swing.JMenuItem`. Detta är en knapp, anpassad för användning i menyer. Användaren väljer motsvarande alternativ genom att klicka på knappen. En meny kan representeras med ett objekt av typen `javax.swing.JMenu`. En komponent placeras i en meny med metoden `add`. Metoden `addSeparator` används för att införa en separeringslinje mellan två komponenter i en meny.

På följande vis kan man skapa ett antal valkomponenter, och utifrån dessa en meny:

```
JMenuItem    serifItem = new JMenuItem ("Serif");
JMenuItem    sansSerifItem = new JMenuItem ("SansSerif");
JMenuItem    monospacedItem = new JMenuItem ("Monospaced");
JMenuItem    dialogItem = new JMenuItem ("Dialog");
JMenuItem    dialogInputItem = new JMenuItem ("DialogInput");

JMenu    fontNameMenu = new JMenu ("Font");
fontNameMenu.add (serifItem);
fontNameMenu.add (sansSerifItem);
fontNameMenu.addSeparator ();
fontNameMenu.add (monospacedItem);
fontNameMenu.addSeparator ();
fontNameMenu.add (dialogItem);
fontNameMenu.add (dialogInputItem);
```

Ett objekt av typen `JMenuItem` är ett slags knapp som har ett stabilt läge. När knappen klickas, skapas en händelse av typen `java.awt.ActionEvent`. Denna händelse kan fångas med en lyssnare av typen `java.awt.ActionListener`, och hanteras i metoden `actionPerformed`. Det går att göra så här:

```
ActionListener    fontNameListener = new ActionListener ()
    {
        public void actionPerformed (ActionEvent e)
        {
            JMenuItem    val = (JMenuItem) e.getSource ();
            String    namn = val.getText();
            Font    font0 = displayArea.getFont ();

            Font    font = new Font (namn, font0.getStyle (),
                                           font0.getSize ());

            displayArea.setFont (font);
        }
    };
```

Först avläses den text som visas på den valda komponenten. Denna text används sedan för att välja motsvarande font i en textarea.

 © FADIL GALJIĆ

Det behövs en lyssnare för varje komponent i en meny. En komponents lyssnare ska reagera på ett lämpligt sätt när komponenten väljs. Man kan naturligtvis ha en gemensam lyssnare för flera komponenter i en meny. I så fall bestämmer man först vilken komponent som användaren valt, och reagerar sedan på lämpligt sätt. Lyssnaren `fontNameListener`, till exempel, kan användas för samtliga komponenter i menyn `fontNameMenu`:

```
serifItem.addActionListener (fontNameListener);
sansSerifItem.addActionListener (fontNameListener);
monospacedItem.addActionListener (fontNameListener);
dialogItem.addActionListener (fontNameListener);
dialogInputItem.addActionListener (fontNameListener);
```

Vanligtvis placeras flera menyer i en menyrad, som sedan placeras överst i ett fönster. En menyrad representeras med ett objekt av typen `javax.swing.JMenuBar`. En meny placeras i en menyrad med metoden `add`, till exempel på detta vis:

```
JMenuBar    menus = new JMenuBar ();
menus.add (fontNameMenu);
```

Kryssrutor och radioknappar i en meny

En meny kan innehålla komponenter av olika typer. Förutom tvålägeskomponenter med ett stabilt läge, kan även komponenter med två stabila lägen användas. En kryssruta, som kan vara markerad eller inte, kan användas. Menyn kan även innehålla en radioknapp, som kan vara på eller av. Radioknappar grupperas vanligtvis i en grupp, där bara en knapp i taget kan vara på (se bilden nedan).

© FADIL GALJIĆ

En kryssruta i en meny kan representeras med ett objekt av typen `javax.swing.JCheckBoxMenuItem`:

```
JCheckBoxMenuItem     boldStyle = new JCheckBoxMenuItem ("Fet");
JCheckBoxMenuItem     italicStyle =
                           new JCheckBoxMenuItem ("Kursiv");

JMenu    fontStyleMenu = new JMenu ("Stil");
fontStyleMenu.add (boldStyle);
fontStyleMenu.add (italicStyle);
```

 © FADIL GALJIĆ

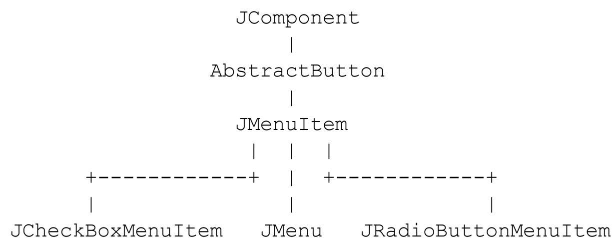
Här skapas en meny med två kryssrutor. Ändringar av en kryssrutas tillstånd kan fångas med en lyssnare av typen `java.awt.ActionListener`. Med metoden `isSelected` kan man kontrollera om en viss kryssruta är markerad.

En radioknapp i en meny kan representeras med ett objekt av typen `javax.swing.JRadioButtonMenuItem`. En grupp sådana radioknappar kan representeras med ett objekt av typen `javax.swing.ButtonGroup`. Man kan till exempel göra så här:

```
JRadioButtonMenuItem    blackColor =
                       new JRadioButtonMenuItem ("Svart");
JRadioButtonMenuItem    blueColor =
                       new JRadioButtonMenuItem ("Blå");
JRadioButtonMenuItem    redColor =
                       new JRadioButtonMenuItem ("Röd");
ButtonGroup    colorGroup = new ButtonGroup ();
colorGroup.add (blackColor);
colorGroup.add (blueColor);
colorGroup.add (redColor);

JMenu    colorMenu = new JMenu ("Färg");
colorMenu.add (blackColor);
colorMenu.add (blueColor);
colorMenu.add (redColor);
```

Här skapas en meny med tre radioknappar. Ändringar i en radioknapps tillstånd kan fångas med en lyssnare av typen `java.awt.ActionListener`. Med metoden `isSelected` kan man kontrollera om en viss radioknapp är markerad.

Klasserna `JCheckBoxMenuItem` och `JRadioButtonMenuItem` är subklasser till klassen `JMenuItem`. Klassen `JMenuItem` är en subklass till klassen `javax.swing.AbstractButton`. Motsvarande klasshierarki kan representeras så här:



Klassen `AbstractButton` är en gemensam superklass till klasserna `JMenuItem`, `JButton` och `JToggleButton` (som är superklass till klasserna `JCheckBox` och `JRadioButton`). Denna klass och dess subklasser representerar olika komponenter med två lägen. Alla dessa klasser har ett antal gemen-

© FADIL GALJIĆ

samma metoder. Metoderna `setText` och `setIcon`, till exempel, definieras i klassen `AbstractButton`. Man kan därför ha en text och/eller en ikon på en godtycklig komponent av denna typ. Klassen `JMenuItem` och dess subklasser representerar komponenter som är anpassade för användning i en meny. Text och ikoner kan användas även i samband med komponenter i en meny.

Submenyer

En meny kan användas som en komponent i en annan meny. En meny i en annan meny kallas för en *submeny* (se bilden nedan). Genom att även använda submenyer kan man åstadkomma välorganiserade valstrukturer. Det blir också möjligt att ha ett stort antal komponenter i en och samma meny.

En meny med en submeny kan skapas så här:

```
JMenuItem    whiteBackground = new JMenuItem ("Vit");
JMenuItem    lightgrayBackground = new JMenuItem ("Ljusgrå");
JMenuItem    grayBackground = new JMenuItem ("Grå");
JMenuItem    darkgrayBackground = new JMenuItem ("Mörkgrå");

JMenu    grayMenu = new JMenu ("Grå");
grayMenu.add (lightgrayBackground);
grayMenu.add (grayBackground);
grayMenu.add (darkgrayBackground);

JMenu    backgroundMenu = new JMenu ("Bakgrund");
backgroundMenu.add (whiteBackground);
```

 © FADIL GALJIĆ

```
backgroundMenu.add (grayMenu);
```

Här skapas en huvudmeny med två komponenter. En av dessa två komponenter är en annan meny (en submeny), som innehåller tre komponenter.

Klassen `JMenu` är en subklass till klassen `JMenuItem`. Det betyder att en meny är utformad så att den passar bra som komponent i en annan meny. En meny är en behållare, som alla andra komponenter av typen `javax.swing.JComponent`. Därför går det att placera andra komponenter i en meny. En meny är dessutom en komponent med två lägen (som alla andra komponenter av typen `javax.swing.AbstractButton`). Den kan vara öppen eller stängd. Användaren kan därmed välja via en meny på samma sätt som via andra tvålägeskomponenter (ett av två möjliga lägen väljs). En meny är därför ett passande element i en annan meny.

Blockera och aktivera en menys komponenter

Ett alternativ i ett program kan under vissa förhållanden bli inaktuellt. Användaren kan i detta fall förhindras att välja alternativet. Den komponent i den motsvarande menyn som används för att välja alternativet kan blockeras. Om alternativet sedan åter blir aktuellt, kan komponenten aktiveras igen.

En komponent i en meny kan blockeras och aktiveras med metoden `setEnabled` (i klassen `javax.swing.JMenuItem`). Om `false` anges som argument till metoden, blockeras motsvarande komponent. Komponenten visas i en särskild färg när den aktuella menyn öppnas (se bilderna nedan), och kan inte användas. Om sedan metoden `setEnabled` anropas med argumentet `true`, aktiveras komponenten på nytt och motsvarande alternativ kan väljas. Med metoden `isEnabled` (i klassen `java.awt.Component`) kan man kontrollera om en komponent i en meny är aktiv.

© FADIL GALJIĆ

Ett program kan innehålla ett fönster som visar en text. I en meny kan det finnas en komponent som gör det möjligt att ta bort texten i fönstret. Om fönstret redan är tomt, kan denna komponent blockeras. Om denna komponent refereras av referensen `rensa`, kan man göra så här:

```
rensa.setEnabled (false);
```

När en text sedan visas i fönstret, kan komponenten `rensa` (som gör det möjligt att rensa fönstret) aktiveras på nytt. Detta görs genom att metoden `setEnabled` anropas med argumentet `true`.

 © FADIL GALJIĆ

Kortkommandon i samband med menyer

Användaren kan öppna en meny genom att klicka på den med musen. Ett alternativ i programmet väljs sedan med en klickning på motsvarande komponent i menyn. Men det finns även ett annat sätt att välja via en meny. En tangent eller en kombination av tangenter kan knytas till en komponent i menyn. Motsvarande alternativ kan i så fall aktiveras med denna tangent eller tangentkombination. Användaren väljer med kortkommandon från tangentbordet, istället för att klicka med musen på motsvarande komponenter. På så sätt går det snabbare och lättare att använda menyer.

En tangent kan knytas till en menykomponent med metoden `setMnemonic`. Tangenten blir då komponentens *mnemonic*. Tangentens virtuella kod anges som argument till metoden `setMnemonic`. Man kan till exempel göra så här:

```
JMenuItem    item = new JMenuItem ("Red");
int   mnemonic = KeyEvent.VK_R;
item.setMnemonic (mnemonic);
```

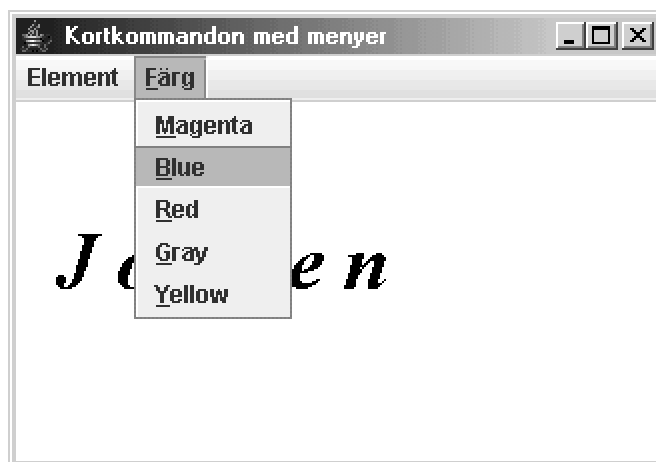
Här anges att tangenten som innehåller bokstaven `R` ska vara mnemonic för den menykomponent som innehåller texten `Red`.

En tangents virtuella kod preciseras via motsvarande konstant från klassen `java.awt.event.KeyEvent`. Konstanterna `VK_A`, `VK_0`, `VK_F1`, `VK_ESCAPE` och andra kan användas. För de tangenter som representerar siffror och engelska (stora) bokstäver är en tangents virtuella kod lika med motsvarande Unicode-kod. Detta innebär att en siffra eller en engelsk stor bokstav kan anges som argument till metoden `setMnemonic`, till exempel så här:

```
item.setMnemonic ('R');
```

När en tangent som innehåller ett tecken knyts till en komponent vars text innehåller detta tecken, visas tecknet understruket. Bokstaven `R` i texten `Red`, till exempel, är understruken (se bilden nedan). Om ett tecken förekommer flera gånger i en komponents text, blir den första förekomsten av tecknet understruken. Detta förvalda beteende kan sättas ur spel med metoden `setDisplayedMnemonicIndex` (som ärvs från klassen `javax.swing.AbstractButton`). Som argument till metoden anges då index för det tecken som ska vara understruket.

© FADIL GALJIĆ



När en tangent knutits till en menykomponent (när en mnemonic har skapats), kan det motsvarande alternativet aktiveras via tangenten. Användaren öppnar den aktuella menyn, håller ned `Alt`-tangenten och trycker därefter på mnemonic-tangenten. Kortkommandot `Alt-R`, till exempel, kan användas för att aktivera motsvarande alternativ.

En tangent kan även knytas till en meny (en mnemonic kan även skapas för en meny). Menyn kan i så fall öppnas med tangenten (mnemonic-tangenten trycks medan `Alt`-tangenten hålls ned). Så här, till exempel, kan detta göras:

```
JMenu    meny = new new JMenu ("Färg");
meny.add (item);
meny.setMnemonic (KeyEvent.VK_F);
```

I det här fallet blir bokstaven `F` i menyn understruken. Menyn kan öppnas genom att `Alt`-tangenten hålls ned medan `F`-tangenten trycks. Användaren kan sedan direkt fortsätta genom att trycka på `R`-tangenten. På så sätt väljs motsvarande alternativ. Det blir lätt och bekvämt att använda menyn.

I vissa fall vill man aktivera ett alternativ på menyn utan att öppna menyn. Det kan gälla viktiga alternativ i ett program, som används ofta och som man vill komma åt så snabbt som möjligt. Till detta används så kallade *acceleratorer*. En accelerator är en tangent, eller en kombination av en tangent och en eller flera kontrolltangenter, som kan användas för att aktivera ett menyalternativ.

© FADIL GALJIĆ

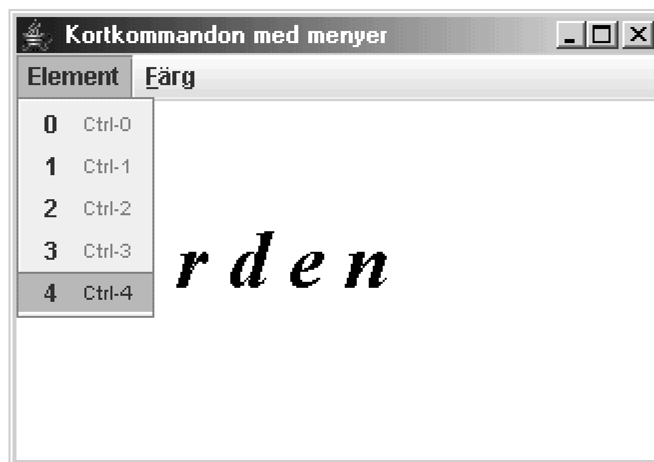
En menykomponent knyts till en accelerator med metoden `setAccelerator`. Acceleratorn anges som argument till metoden. En accelerator representeras med ett objekt av typen `javax.swing.KeyStroke`. Ett sådant objekt skapas med (den statiska) metoden `getKeyStroke` i klassen `KeyStroke`. Så här, till exempel:

```
JMenuItem    item = new JMenuItem (" 1 ");
KeyStroke    accelerator =
    KeyStroke.getKeyStroke (KeyEvent.VK_1, InputEvent.CTRL_MASK);
item.setAccelerator (accelerator);
```

Acceleratorn `Ctrl-1` knyts till ett alternativ i programmet. När kortkommandot utförs, aktiveras motsvarande alternativ. Det går snabbt och lätt att välja i menyn.

Acceleratortangenten (dess virtuella kod) och kontrolltangenten anges som argument till metoden `getKeyStroke`. En kontrolltangent representeras med en lämplig konstant från klassen `java.awt.event.InputEvent`. Konstanterna `SHIFT_MASK`, `CTRL_MASK`, `META_MASK` och `ALT_MASK` kan användas. Man kan även använda en kombination av kontrolltangenter genom att addera flera konstanter (till exempel `InputEvent.CTRL_MASK + InputEvent.ALT_MASK`). Om bara en tangent ska användas som accelerator (utan kontrolltangenter), anges `0` som andra argument till metoden `getKeyStroke`.

En accelerator visas när motsvarande meny öppnas. Den visas till höger i den aktuella komponenten (se bilden nedan).



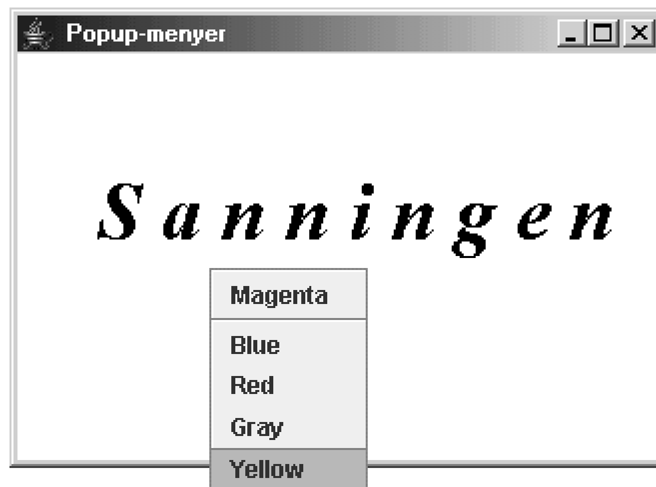
Med hjälp av mnemonics och acceleratorer kan användaren snabbt och bekvämt välja mellan olika alternativ på ett programs menyer.

© FADIL GALJIĆ

Vanligtvis används olika kortkommandon på den plattform där ett program exekveras. I Windows, till exempel, används kortkommandona `Ctrl-C`, `Ctrl-V`, `Ctrl-X` och andra. Man ska därför vara försiktig med sina val av kortkommandon, så att de inte förväxlas med den aktuella plattformens kortkommandon.

Popup-menyer

En meny placeras vanligtvis i en menyrad, överst i ett programs fönster. Men det finns också en annan strategi som kan användas. En meny kan knytas till en godtycklig komponent i ett fönster, och öppnas genom att användaren klickar på någon plats i komponenten. En sådan meny kallas för popup-meny (se bilden nedan).



En popup-meny representeras med ett objekt av typen `javax.swing.JPopupMenu` (en direkt subklass till klassen `javax.swing.JComponent`). Med metoden `add` placeras olika komponenter i en popup-meny, till exempel så här:

```
String[]  fargNamn = {"Magenta", "Blue", "Red", "Gray", "Yellow"};
JMenuItem[]    fargValjare = new JMenuItem[5];
for (int i = 0; i < 5; i++)
    fargValjare[i] = new JMenuItem (fargNamn[i]);

JPopupMenu    popupMeny = new JPopupMenu ();
for (int i = 0; i < fargValjare.length; i++)
{
```

 © FADIL GALJIĆ

```
    popupMeny.add (fargValjare[i]);

    if (i == 0)
        popupMeny.addSeparator ();
}
```

Här skapas en popup-meny utifrån flera komponenter av typen `JMenuItem`. Komponenterna representerar några fördefinierade färger i klassen `java.awt.Color`. För att motsvarande alternativ ska kunna aktiveras via dessa komponenter, måste de förses med lämpliga lyssnare (av typen `ActionListener`).

När en popup-meny har skapats, måste den knytas till en komponent i programmets grafiska gränssnitt. Detta kan till exempel göras så här:

```
JTextArea    displayArea = new JTextArea (4, 6);
displayArea.setText ("\n    S a n n i n g e n");

MouseListener    popupLyssnare = new MouseAdapter ()
    {
        public void mousePressed (MouseEvent e)
        {
            Component    komponent = e.getComponent ();
            int    x = e.getX ();
            int    y = e.getY ();

            if (e.isPopupTrigger ())
                popupMeny.show (komponent, x, y);
        }

        public void mouseReleased (MouseEvent e)
        {
            Component    komponent = e.getComponent ();
            int    x = e.getX ();
            int    y = e.getY ();

            if (e.isPopupTrigger ())
                popupMeny.show (komponent, x, y);
        }
    };

displayArea.addMouseListener (popupLyssnare);
```

Här skapas en textarea och en muslyssnare, och muslyssnaren registreras hos textarean. I definitionsklassen för lyssnaren preciseras vad som ska hända när användaren trycker på en musknapp. Först bestäms komponenten där musknappen trycktes, och koordinaterna för den punkt där knappen trycktes. Därefter visas popup-menyn i komponenten (förälderkomponenten) på den angivna platsen.

© FADIL GALJIĆ

Metoden `show` i klassen `JPopupMenu` används för att visa en popup-meny. Förälderkomponenten och koordinaterna för popup-menyns övre vänstra hörn anges som argument till metoden. Vanligtvis visas en popup-meny på den plats där användaren tryckte på musknappen. När en popup-meny har öppnats, kan ett av dess alternativ väljas. Efter valet blir popup-menyn åter osynlig, och det krävs en ny klickning på förälderkomponenten för att den ska visas igen.

Man använder olika musknappar på olika plattformar för att aktivera en popup-meny. Därför ska metoden `isPopupTrigger` (i klassen `java.awt.event.MouseEvent`) användas för att kontrollera om användaren klickat med den knapp som normalt används på den aktuella plattformen för att aktivera en popup-meny. Om denna knapp har använts, ska den motsvarande popup-menyn visas.

På vissa plattformar aktiveras en popup-meny endast när motsvarande musknapp trycks ner. Sådana händelser fångas i metoden `mousePressed`, och det är bara i den metoden som metoden `isPopupTrigger` kan returnera `true`. På andra plattformar aktiveras en popup-meny när motsvarande musknapp släpps upp. Sådana händelser fångas i metoden `mouseReleased`, och metoden `isPopupTrigger` returnerar `true` bara i den metoden. Därför måste man implementera både metoden `mousePressed` och metoden `mouseReleased` i en muslyssnare som aktiverar popup-menyer. Metoderna ska naturligtvis innehålla samma kod. Koden kan placeras i en hjälpmetod, som anropas både från metoden `mousePressed` och från metoden `mouseReleased`.

Olika popup-menyer kan alltså knytas till olika komponenter i ett grafiskt användargränssnitt. På detta sätt utökas gränssnittets användbarhet.

© FADIL GALJIĆ

Dialoger

Använda dialoger

En *dialog* kan användas för att tillföra ett meddelande till användaren under programmets gång, och/eller för att hämta olika information från användaren. En dialog kan göras synlig via en meny, en knapp, eller någon annan komponent i ett grafiskt användargränssnitt. Vanligtivis används modala dialoger. När en sådan dialog visas blockeras användningen av andra komponenter i det grafiska gränssnittet, och dessa kan inte användas förrän användaren har stängt dialogen. Användaren stänger dialogen via dess stängningsknapp eller via någon komponent inuti dialogen, till exempel genom att trycka på en knapp eller genom att trycka på returtangenten i ett inmatningsfält (se bilden nedan).

En dialog kan representeras med ett objekt av typen `javax.swing.JDialog`. Metoderna `setTitle`, `setSize` och `setLocation` används för att precisera en dialogs rubrik, storlek och placering på skärmen. Metoden `setModal` används för att göra en dialog modal. Metoden `setLayout` används för att bestämma en dialogs layout, och metoden `add` används för att placera olika komponenter i en dialog.

© FADIL GALJIĆ

Vanligtvis skapar man en subklass till klassen `JDialog`, för att definiera en typ av dialoger som passar bra i ett konkret sammanhang. Om dialogen ska användas för att visa information, kan man definiera den i en fristående klass. Men om olika information ska hämtas från en dialog, kan det vara lämpligt att definiera dialogen som en inre klass. En sådan dialog kan använda omgivande variabler, och reagera på ett lämpligt sätt på användarens inmatning.

Ett användargränssnitt kan definieras enligt följande modell:

```
class Display extends JPanel
{
    private String    ordsprak;
    private JTextArea    displayArea;
    private JMenu    ordsprakMeny;
    JDialog    ordsprakDialog = null;


    public Display ()
    {
        this.ordsprak = "Morgonstund har guld i mun!";
        displayArea = new JTextArea (4, 6);
        displayArea.setText (ordsprak);

        JMenuItem    item1 = new JMenuItem ("Förvalt");
        JMenuItem    item2 = new JMenuItem ("Välj");
        ordsprakMeny = new JMenu ("Ordspråk");
        ordsprakMeny.add (item1);
        ordsprakMeny.add (item2);

        item1.addActionListener (new ActionListener ()
            {
                public void actionPerformed (ActionEvent e)
                {
                    ordsprak = "Morgonstund har guld i mun!";
                    displayArea.setText (ordsprak);
                }
            } );

        item2.addActionListener (new ActionListener ()
            {
                public void actionPerformed (ActionEvent e)
                {
                    if (ordsprakDialog == null)
                        ordsprakDialog = new OrdsprakDialog ();

                    ordsprakDialog.setVisible (true);
                 }
             } );
```

 © FADIL GALJIĆ

```
        JMenuBar    menyrad = new JMenuBar ();
        menyrad.add (ordsprakMeny);

        this.setLayout (new BorderLayout ());
        this.add (menyrad, "North");
        this.add (new JScrollPane (displayArea), "Center");
    }
}
```

Här visas ett ordspråk i en textarea. För att olika ordspråk ska kunna visas i textarean, skapas en meny så att användaren kan välja. Användaren kan välja ett förvalt ordspråk, eller mata in ett godtyckligt ordspråk via en dialog. Denna dialog är av typen `OrdsprakDialog` (klassen definieras senare), och visas när användaren väljer alternativet `Välj` i menyn `Ordspråk`. En dialog skapas så här:

```
if (ordsprakDialog == null)
    ordsprakDialog = new OrdsprakDialog ();
```

Dialogen skapas bara den första gången som användaren väljer alternativet `Välj`. När användaren har matat in sitt ordspråk döljs dialogen, men den kan användas även vid ett senare tillfälle. Det är samma dialog som används hela tiden, man gör den bara synlig eller osynlig. Dialogen görs synlig via alternativet `Välj` i menyn, och osynlig genom att användaren stänger den (antingen via dess stängningsknapp eller genom att trycka på returtangenten i dess inmatningsfält).

När användaren anger en text i dialogens inmatningsfält, behöver texten avläsas och användas i textarean. Dialogen måste därför på något sätt kopplas till textarean. Detta kan man göra genom att använda textarean som argument till dialogens konstruktor. En bättre strategi är att definiera dialogen i en inre klass till klassen `Display`. Dialogklassen kan då använda alla instansvariabler i klassen `Display`. Det blir lättare att påverka dessa variabler från den lyssnare som lyssnar och reagerar på användarens inmatning.

Dialogklassen kan definieras som en inre klass i klassen `Display`. Den kan definieras så här:

```
private class OrdsprakDialog extends JDialog
{
    private JLabel    etikett;
    private JTextField    textFalt;

    public OrdsprakDialog ()
    {
        this.setTitle ("Ordspråk-dialog");
        this.setSize (300, 160);
```

© FADIL GALJIĆ

```
        this.setLocation (200, 280);
        this.setModal (true);

        etikett = new JLabel ("Ett ordspråk:");
        textFalt = new JTextField (20);
        textFalt.addActionListener (new ActionListener ()
            {
                public void actionPerformed (ActionEvent e)
                {
                    ordsprak = textFalt.getText ();
                    textFalt.setText ("");

                    displayArea.setText ("\n  " + ordsprak);

                    setVisible (false);
                }
            } );

        JPanel    panel1 = new JPanel ();
        panel1.add (etikett);
        JPanel    panel2 = new JPanel ();
        panel2.add (textFalt);
        this.setLayout (new GridLayout (2, 1));
        this.add (panel1);
        this.add (panel2);
    }
}
```

En dialog av typen `OrdsprakDialog` innehåller en etikett och ett textfält. En lyssnare definieras, som lyssnar på användarens inmatning via textfältet. När användaren har matat in ett ordspråk, tas det och visas i textarean. Sedan görs dialogen osynlig. Dialogen kan göras synlig igen via motsvarande menyalternativ (`Välj`). Eftersom klassen `OrdsprakDialog` definieras som en inre klass i klassen `Display`, kan variablerna `ordsprak` och `displayArea` (som definieras i klassen `Display`) användas. Variablerna kan användas i klassen `OrdsprakDialog`, och därmed i dess inre, anonyma lyssnarklass.

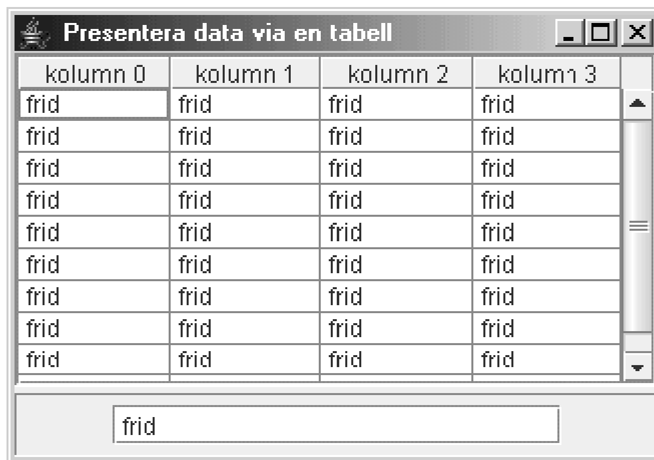
En dialog är ett bekvämt verktyg som kan användas för kommunikation mellan programmet och användaren. En dialog kan visas och döljas efter behov. Dialogen kan visas via en meny, och döljas när användaren har stängt den. En dialog kan definieras i en inre klass, och på sätt integreras dialogen i sin omgivning.

 © FADIL GALJIĆ

Datahantering via tabeller

Presentera data via en tabell

Olika uppgifter i ett program kan presenteras via en *tabell* (se bilden nedan). En tabell är en grafisk komponent som består av ett antal celler. Cellerna är fördelade i ett antal rader och ett antal kolumner. En enskild uppgift (ett element) lagras i en cell i tabellen. Varje tabellkolumn har ett namn.



En tabell kan representeras med ett objekt av typen `javax.swing.JTable`. Man skapar en sådan tabell utifrån en lämplig *modell*. Modellen definierar en tabells data och datans fördelning mellan olika rader och kolumner. Den definierar också de olika kolumnernas namn.

En modell definieras i en klass som implementerar gränssnittet `javax.swing.table.TableModel`. Klassen `javax.swing.table.AbstractTableModel` är en abstrakt klass som implementerar ett antal viktiga metoder i detta gränssnitt. Man kan ärva från denna klass, och endast implementera de metoder i gränssnittet `TableModel` som inte implementeras i klassen `AbstractTableModel`. Dessa metoder är `getRowCount`, `getColumnCount` och `getValueAt`. Man kan också omdefiniera vissa metoder i klassen `AbstractTableModel`, för att anpassa dem till konkreta behov.

En modell för en tabell kan definieras enligt följande mönster:

```
class EnkelModell extends AbstractTableModel
```

© FADIL GALJIĆ

```
{
    private int         antalRader;
    private int         antalKolumner;
    private String[]    kolumnNamn;
    private Object      data = " frid";

    public EnkelModell (int antalRader, int antalKolumner,
                        String[] kolumnNamn)
    {
        this.antalRader = antalRader;
        this.antalKolumner = antalKolumner;
        this.kolumnNamn = kolumnNamn;
    }


    public void setData (Object data)
    {
        this.data = data;

        this.fireTableDataChanged ();
    }

    public int getRowCount ()
    {
        return antalRader;
    }

    public int getColumnCount ()
    {
        return antalKolumner;
    }

    public Object getValueAt (int rad, int kolumn)
    {
        Object    element = data;

        return element;
    }

    public String getColumnName (int kolumn)
    {
        return kolumnNamn[kolumn];
    }
}
```

En modell av typen `EnkelModell` innehåller uppgifter om antalet rader och kolumner i en tabell. Den innehåller också uppgifter om kolumnernas namn och de data som ska visas i tabellen. I den här modellen används samma uppgift i alla celler i tabellen. Det går naturligtvis att an-

© FADIL GALJIĆ

vända olika uppgifter i olika celler (i så fall ska returvärde från metoden getValueAt vara rad- och kolumnberoende).

En modell av typen EnkelModell får den uppgift som ska visas i en tabells celler via metoden setData. Denna metod anropar metoden fireTable-DataChanged (i superklassen AbstractTableModel), för att uppdatera den motsvarande tabellen varje gång de underliggande uppgifterna i modellen ändras.

Metoden getRowCount returnerar antalet rader i tabellen, och metoden getColumnCount returnerar antalet kolumner. Metoden getValueAt returnerar den uppgift som ska visas i den cell som finns i rad nummer rad och kolumn nummer kolumn. Via den här metoden definieras innehållet i tabellens celler. Dessa uppgifter kan beräknas, hämtas från en vektor, läsas in från en fil, eller erhållas på något annat sätt. En modell kan ha en referens till de uppgifter som ska visas (som i den här klassen), eller en referens till en uppgiftskälla (till exempel en referens till ett textfält eller till en fil).

För att precisera namnen på de olika kolumnerna omdefinieras metoden getColumnName i klassen EnkelModell. Superklassen AbstractTableModel använder stora bokstäver (A, B, C, ...) som namn på en tabells kolumner. För att använda andra namn, omdefinierar man metoden getColumn-Name.

En modell av typen EnkelModell kan skapas så här:

```
String[]    kolumnNamn = {"kolumn 0", "kolumn 1",
                          "kolumn 2","kolumn 3"};
EnkelModell    modell = new EnkelModell (10, 4, kolumnNamn);
```

Antalet rader och kolumner i tabellen anges, samt kolumnernas namn. När en modell har skapats, kan man använda denna för att skapa en tabell:

```
JTable    tabell = new JTable (modell);
```

Vanligtvis placeras en tabell i en behållare av typen javax.swing.JScrollPane. I detta fall visas de olika kolumnernas namn. Om tabellen placeras direkt i ett fönster visas inte dessa namn, utan måste då läggas till som en separat komponent.

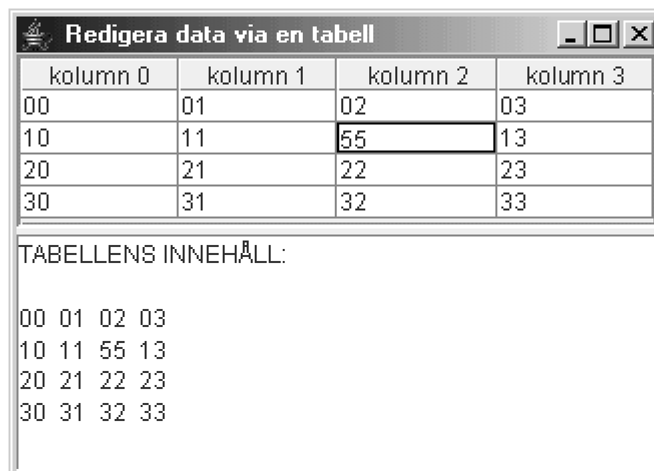
Tabellen tabell kan uppdateras med metoden setData. En ny sträng anges som argument till metoden. Det är denna sträng som ska visas i tabellens celler.

I vissa fall finns de uppgifter som ska presenteras i en tvådimensionell vektor, och kolumnernas namn i en annan vektor. En tabell kan då skapas

utifrån dessa två vektorer (vektorerna anges som argument när tabellen skapas). Man behöver inte skapa motsvarande modell själv, eftersom konstruktorn gör detta. Genom att definiera en egen modell, kan man åstadkomma större flexibilitet. Man kan till exempel bestämma de värden som ska visas i en tabells celler, utan att lagra värdena i en separat lagringsstruktur (värdena kan till exempel beräknas i metoden `getValueAt`, eller matas in från en fil). Man kan också definiera olika sätt att bearbeta data (mata in eller ändra data) via en tabell.

Redigera data via en tabell

Det går att komma åt en viss cell i en tabell, och redigera uppgiften i den (se bilden nedan). Man kan ändra en befintlig uppgift, eller lägga till en ny uppgift i en tom cell i tabellen.



För att uppgifter i en tabell ska kunna redigeras, måste en lämplig modell för tabellen skapas. I denna modell preciseras vilka celler som ska vara redigerbara. Detta görs i metoden `isCellEditable` (metoden omdefinieras). Metoden `setValueAt` måste också implementeras. Denna metod anropas automatiskt varje gång en cells innehåll redigeras. Metoden får den nya uppgiften (som ett objekt av typen `Object`) som sitt första argument. Förutom det får metoden cellens position i tabellen via det andra och det tredje argumentet. I metoden `setValueAt` kan cellens nya värde användas på olika sätt. Den motsvarande datamodellen måste dock uppdateras. När redigeringen av en cell är avslutad, ritas tabellen om. För att

 © FADIL GALJIĆ

bestämma tabellens nya innehåll, anropas metoden `getValueAt` igen. Metoden använder nya, uppdaterade data. Om inte dessa data ändrats via metoden `setValueAt` hämtar tabellen gamla värden, och de värden som matats in via tabellens celler tappas bort. Metoderna `setValueAt` och `getValueAt` möjliggör kommunikation mellan datamodellen och tabellen. Metoden `setValueAt` avläser en uppgift i tabellen, och modifierar den underliggande modellen. Metoden `getValueAt` avläser en uppgift i modellen, och ändrar innehåll i en cell i tabellen.

En modell som gör det möjligt att redigera innehållet i motsvarande tabell kan definieras så här:

```
class TabellModell extends javax.swing.table.AbstractTableModel
{
    private Object[][]    data = null;

    public void setData (Object[][] data)
    {
        this.data = data;

        this.fireTableDataChanged ();
    }

    public int getRowCount ()
    {
        return 4;
    }

    public int getColumnCount ()
    {
        return 4;
    }

    public Object getValueAt (int rad, int kolumn)
    {
        Object    element = data[rad][kolumn];

        return element;
    }

    public boolean isCellEditable (int rad, int kolumn)
    {
        return true;
    }

    public void setValueAt (Object inputVarde,
                            int rad, int kolumn)
    {
        data[rad][kolumn] = inputVarde;
```

© FADIL GALJIĆ

```
    }
}
```

Här anges att alla celler i tabellen ska vara redigerbara (naturligtvis är det möjligt att göra endast vissa celler redigerbara). Via metoden `setValueAt` uppdateras underliggande data varje gång innehållet i en cell i tabellen ändras.

Man kan skapa en modell av typen `TabellModell` och en tabell utifrån denna modell, på följande vis:

```
String[][]    data = new String[4][4];
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
        data[i][j] = new String (i + "" + j);
TabellModell    modell = new TabellModell ();
modell.setData (data);
JTable    tabell = new JTable (modell);
```

För att en tabells uppgifter ska kunna redigeras, måste användaren kunna markera (välja) en viss cell i tabellen. Så här görs detta möjligt:

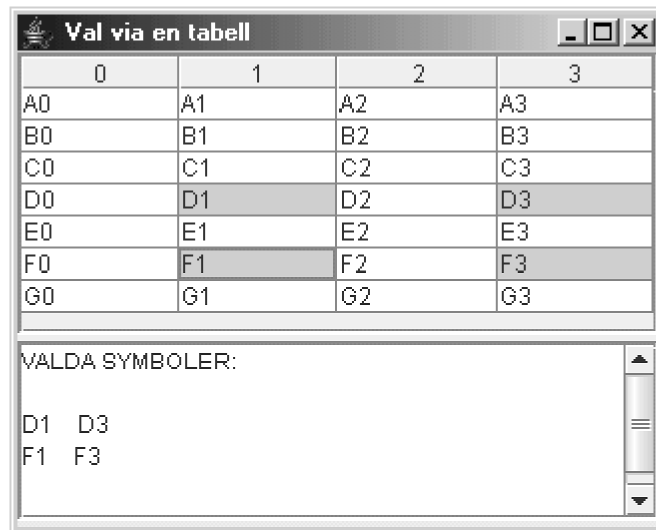
```
tabell.setCellSelectionEnabled (true);
```

En tabell kan alltså användas både för att presentera data och för att redigera data. En tabell är både en utmatningskomponent och en inmatningskomponent. Den tar olika uppgifter från ett program och presenterar dessa uppgifter för användaren. Den tar också användarens uppgifter och överför dem till programmet. En tabell möjliggör för data att flöda mellan programmet och användaren på ett välstrukturerat sätt.

Val via en tabell

Olika alternativ i ett program kan väljas via en tabell (se bilden nedan). Användaren kan välja en eller flera celler, och därmed de symboler som finns i dessa celler. Symbolerna kan sedan användas för att aktivera motsvarande alternativ. Varje nytt val kan registreras, så att programmet kan reagera på ett lämpligt sätt.

 © FADIL GALJIĆ



För att val ska kunna göras via en tabell, måste användaren först tillåtas markera olika celler. Om referensen `tabell` refererar till en tabell, kan detta göras så här:

```
tabell.setCellSelectionEnabled (true);
```

För att kunna reagera på användarens val, måste man ha tillgång till motsvarande *valmodell*. Varje tabell har en inbyggd valmodell, som representeras via ett objekt av typen `javax.swing.ListSelectionModel`. Modellen returneras av metoden `getSelectionModel`:

```
ListSelectionModel    valModell = tabell.getSelectionModel ();
```

När användaren markerar (väljer) en eller flera celler i en tabell, skapas en händelse av typen `javax.swing.event.ListSelectionEvent`. Denna händelse kan fångas i en lyssnare av typen `javax.swing.event.ListSelectionListener`. Händelsen hanteras i metoden `valueChanged`. En lyssnare av typen `ListSelectionListener` registreras hos en tabells valmodell (inte hos en tabell). Detta kan till exempel göras så här:

```
valModell.addListSelectionListener (new ListSelectionListener ()
    {
        public void valueChanged (ListSelectionEvent e)
        {
            int[]    valdaRader = tabell.getSelectedRows ();
            int[]    valdaKolumner = tabell.getSelectedColumns ();
```

```
            for (int i = 0; i < valdaRader.length; i++)
            {
                for (int j = 0; j < valdaKolumner.length; j++)
                {
                    Object    enValdSymbol = tabell.getValueAt (
                                 valdaRader[i], valdaKolumner[j]);
                    textarea.append (enValdSymbol + "\n" );
                }
            }
        }
    } );
```

Metoderna `getSelectedRows` och `getSelectedColumns` ger de valda raderna och kolumnerna. Motsvarande ordningsnummer lagras i två heltalsvektorer. Användaren kanske väljer cellerna i den andra och tredje kolumnen i den första raden. Man får i så fall en heltalsvektor som bara innehåller ett element (`0`, första raden), och en heltalsvektor som innehåller två element (`1` och `2`, andra och tredje kolumnen). Utifrån dessa vektorer vet man vilka celler som är valda.

Metoden `getValueAt` (i klassen `JTable`) returnerar den symbol (det värde) som finns i en markerad cell. Motsvarande rad och kolumn anges som argument till metoden. När de valda symbolerna erhållits, kan de användas på olika sätt. Utifrån dessa symboler väljs motsvarande alternativ i programmet.

Flera celler kan väljas vid ett och samma tillfälle. Detta görs med musen och tangenterna `Ctrl` och `Shift`. På följande vis kan valet begränsas till celler i flera successiva rader:

```
valModell.setSelectionMode(
           ListSelectionModel.SINGLE_INTERVAL_SELECTION)
```

Så här kan valet begränsas till celler i enskilda rader:

```
valModell.setSelectionMode(ListSelectionModel.SINGLE_SELECTION)
```

En valmodell använder konstanten `MULTIPLE_INTERVAL_SELECTION` som förinställt val, och därför kan godtyckliga celler väljas.

Valet kan begränsas till celler i enskilda rader, och celler i ett intervall med rader. Begränsningarna baseras på tabellens rader. Om man vill binda begränsningarna till tabellens kolumner, behövs det en valmodell som baseras på kolumnerna. Denna erhålls på följande vis:

```
ListSelectionModel    valModell =
                   tabell.getColumnModel ().getSelectionModel ();
```

 © FADIL GALJIĆ

En tabell är alltså en avancerad grafisk komponent, som gör det möjligt att presentera och redigera olika uppgifter, och att välja mellan olika alternativ. Via en tabell kan både utmatning, inmatning och val utföras.

© FADIL GALJIĆ

Sakregister

accelerator, 368
accept, 125
ActionEvent, 264
ActionListener, 264
actionPerformed, 264
adapterklass, 305
AffineTransform, 250
AlphaComposite, 232
anonym klass, 16
användargränssnitt, 148
Arc2D, 184
Area, 184
asynkron kommunikation, 108
await, 94
avbryta en tråd, 27
AWT, 268

BasicStroke, 200
bild, 210
bildens pixlar, 215
bildkvalitet, 231
blockerad tråd, 19
BlockingQueue, 107
Border, 274
BorderLayout, 280
Buffer, 106
BufferedImage, 210

Channel, 98
clip, 230
Color, 203, 228
ColorModel, 219
Component, 268
Condition, 94
Container, 268
CubicCurve2D, 184
currentThread, 18

demontråd, 24
dialog, 160, 373
dialogruta, 163
Drawable, 240
dödläge, 71

Ellipse2D, 184
etikett, 324
EventObject, 299

fildialog, 173
filfilter, 176
flertrådad server, 130
FlowLayout, 278
Font, 205
fyllnadsmönster, 227
färg, 198
fönster, 148
förflytta en figur, 252

GeneralPath, 184
geometrisk figur, 183
GradientPaint, 228
grafik, 199
grafisk kontext, 198
grafiskt användargränssnitt, 261
grafiskt program, 148, 261
Graphics, 199
GridBagLayout, 284
GridLayout, 281
GUI, 261

helt synkroniserade objekt, 52
host, 116

HTML, 325
händelse, 263, 299

Icon, 325
ikon, 325
ImageIcon, 325
ImageIO, 214
InetAddress, 117
inmatning, 329, 382
inre klasser, 307, 309
interna fönster, 295
Internetadress, 117
interrupt, 27
interrupted, 28
InterruptedException, 28

JButton, 336
JCheckBox, 339
JCheckBoxMenuItem, 362
JColorChooser, 178
JComboBox, 342
JDialog, 160, 373
JEditorPane, 327
JFileChooser, 174
JFrame, 149
JInternalFrame, 296
JLabel, 325
JList, 347
JMenu, 360
JMenuBar, 361
JMenuItem, 360
join, 22
JOptionPane, 163, 164
JPanel, 271
JPopupMenu, 370
JRadioButton, 340
JRadioButtonMenuItem, 363
JScrollPane, 292
JSlider, 350
JSpinner, 353
JSplitPane, 290
JTable, 377
JTextArea, 323
JTextComponent, 322
JTextField, 323, 329
JToolBar, 357
JWindow, 159

kanal, 135
KeyEvent, 316
klient, 116, 130
knapp, 336
kommunikation, 116
kommunikation mellan trådar, 98
komposition av två bilder, 232
konsolprogram, 148, 260
kontrollera en tråds aktivitet, 27
koordinatbyte, 249
kortkommandon, 367
kryssruta, 339
körbar tråd, 20

layout, 277
LayoutManager, 277
Line2D, 184
lista, 346
Lock, 88
lokala klasser, 16, 311
Look & Feel, 272
lyssnare, 264, 301
lyssnarklasser, 306
lås, 50
läslås, 92

meny, 359
menyrad, 361
mnemonic, 367
modal dialog, 161
MouseEvent, 313
MouseMotion, 314
Movable, 240
mushändelse, 313

 © FADIL GALJIĆ

notify, 77
notifyAll, 77

ObjectContainer, 100

Paint, 228
Point2D, 182
pool av trådar, 135
popup-meny, 370
port, 118
Printable, 220
PrinterJob, 221
prioritet för en tråd, 25
punkt, 182

QuadCurve2D, 184

radioknapp, 340
ram, 148
ramar runt komponenter, 274
raster, 215
Rectangle2D, 184
RectangularShape, 184
RenderingsHints, 231
rita figurer, 198
rita tecken, 205
ritningsområde, 229
rotera en figur, 254
RoundRectangle2D, 184
run, 6
Runnable, 6
rörliga figurer, 237

server, 116, 130
ServerSocket, 125
Shape, 183
showConfirmDialog, 169
showInputDialog, 169
showMessageDialog, 169
showOptionDialog, 169
signalAll, 94
skalfaktorer, 251
skjuva en figur, 255
skriva ut en bild, 220
skrivlås, 92
sleep, 19
Socket, 118
spara en bild, 214
start, 8
streckmönster, 224
submeny, 364
Swing, 269
symbolgenerator, 352
synchronized, 48, 56
synkron kommunikation, 111
synkronisera trådar, 48, 88
synkroniserad metod, 48
synkroniserade behållare, 65
synkroniserat block, 55
synkroniseringslås, 88

tabell, 377
TableModel, 377
tangentbordshändelse, 316
textarea, 323
textdokument, 331
textfält, 323, 329
textkomponent, 322
TexturePaint, 229
Thread, 7
ThreadGroup, 36
tillståndsberoende operationer, 75, 93
tråd, 6
tråd per klient, 131

UnknownHostException, 117
URL, 327
utmatning, 322, 382

wait, 77
wait-mängd, 78
val, 382
vallinjal, 349

valruta, 342
webbsida, 327
verktygsfält, 357
villkorsobjekt, 94
WritableRaster, 215
värddator, 116

© FADIL GALJIĆ