10 Uppräknade datatyper

Uppräknade datatyper

Hur ser vi till att en variabel endast kan anta vissa värden?

Från tidigare kapitel vet vi att vi kan använda publika klasskonstanter:

Letter.java

```
public class Letter {

    private int myPostageType;     // Ett av nedanstående värden
    private int myWeight;

    public static final int ECONOMY = 1;
    public static final int NORMAL  = 2;
    public static final int INSURED = 3;
    public static final int EXPRESS = 4;

    ...
}
```

Bristerna med denna teknik är flera:

- Ingen egen typ – vi "lånar" en vanlig typ som `int` till att betyda något annat
- Variabler kan sättas till ogiltiga värden utan att kompilatorn upptäcker det
- Svårt att veta om alla giltiga värden behandlas överallt i koden

Ett ständigt återkommande problem inom programmering är hur man ser till att en variabel endast antar värden ur en begränsad lista av möjliga alternativ. Ett exempel är en klass `Letter` som representerar brev, där vi vill att en instansvariabel ska representera vilken befordringstyp som gäller för ett enskilt brev. Giltiga alternativ är ekonomibrev, normalbrev, försäkrat (assurerat) brev och expressbrev.

I kapitlet *Klassegenskaper* såg vi exempel på hur publika klasskonstanter kunde användas för att sätta symboliska namn på ett antal giltiga värden som en typ kan ha. Den här tekniken kan användas för att representera de fyra olika befordringstyperna med fyra klasskonstanter i klassen `Letter`. Tekniken har emellertid flera brister:

- Även om de giltiga värdena framträder mycket tydligt i källkoden, har värdena trots allt inte en egen typ. Vi "lånar" en vanlig typ, som i det här fallet heltalstypen `int`, till att betyda något mycket mer specifikt. Det betyder att överallt där en variabel, en parameter eller ett returvärde avses vara av den här typen, kommer vi endast att se typen `int`. Vi måste själva veta vad detta egentligen betyder.
- Eftersom den underliggande typen (i vårt exempel `int`) kan innehålla många värden utöver de för oss giltiga värdena, finns risken att en variabel kommer att innehålla ett ogiltigt värde. Kompilatorn har ingen möjlighet att upptäcka sådana fel, eftersom alla kompileringskontroller sker mot den deklarerade typen, t ex `int`.
- Det blir svårt att försäkra sig om att samtliga giltiga värden verkligen behandlas överallt i källkoden där de används. Detta blir särskilt känsligt i ett underhållsläge när det tillkommer ett eller flera nya giltiga värden. Vad händer om vi plötsligt måste ta hänsyn till en femte befordringstyp?

Uppräknade datatyper

Definition

Fr o m version 5 av Java finns *uppräknade datatyper* som eget språkelement:

PostageType.java

```
public enum PostageType { ECONOMY, NORMAL, INSURED, EXPRESS }
```

Letter.java

```
public class Letter {

    private PostageType myPostageType;
    private int         myWeight;

    public Letter(PostageType type, int weight) {
        myPostageType = type;
        myWeight = weight;
    }

    public PostageType getPostageType() {
        return myPostageType;
    }

    public int getWeight() {
        return myWeight;
    }
}
```

I flera programspråk har *uppräknade datatyper* introducerats för att lösa de problem som beskrevs på föregående sida. Fr o m version 5 finns denna möjlighet även i Java. I princip ges därmed språkstöd för det som tidigare var ett framgångsrikt sätt att simulera uppräknade datatyper i Java.

En uppräknad datatyp i Java är i grund och botten en klass av objekt, men med ett förbestämt antal kända instanser. I stället för att definieras i en vanlig klassdefinition, används en definition där det reserverade ordet `class` ersatts av `enum` (härlett från en förkortning av engelskans *enumeration*, dvs uppräkning). Efter definitionens vänsterklammer följer en kommaseparerad lista av samtliga giltiga värden som variabler av typen kan anta:

```
public enum PostageType { ECONOMY, NORMAL, INSURED, EXPRESS }
```

Det går utmärkt att dela upp definitionen på flera källkodsrader:

```
public enum PostageType {
    ECONOMY,
    NORMAL,
    INSURED,
    EXPRESS
}
```

Den nya uppräknade datatypen (`PostageType` i vårt exempel) kan nu användas som ett självständigt typnamn för att ange typen för en variabel, en parameter eller ett returvärde:

```
public Letter(PostageType type, int weight) { … }
```

Uppräknade datatyper

Användning och regler

- Den uppräknade datatypen är egentligen en speciell form av klass.
- De giltiga värdena är referenser till fördefinierade instanser av denna klass.
- Det går inte att skapa ytterligare instanser av klassen.
- Konventionen är att namnge värdena med versaler.
- Värdena används på ett sätt som påminner om klasskonstanter:

```
Letter ltr = new Letter(PostageType.ECONOMY, 175);
```

- Jämförelse av värden med såväl == som `equals` fungerar:

```
if (ltr.getPostageType() == PostageType.NORMAL) ...
```

- Det går att skriva ut värdena:

```
System.out.print(ltr.getPostageType());   // ECONOMY
```

- Klassmetoden `valueOf` kan returnera värdet givet en textsträng:

```
PostageType pt = PostageType.valueOf("ECONOMY");
```

En uppräknad datatyp i Java är alltså egentligen en speciell form av klass. De värden vi räknar upp i typens definition är referenser till instanser av denna klass som skapas automatiskt av runtimesystemet. Språkdefinitionen garanterar att det inte går att skapa några ytterligare instanser av en klass som är en uppräknad datatyp.

Tekniskt är de uppräknade värdena publika klasskonstanter i klassen, även om bestämningarna `public`, `static` och `final` inte används för att deklarera dem. Detta kan också förklara varför de följer samma namnsättningskonvention som publika klasskonstanter, dvs de skrivs typiskt med enbart versaler (eller möjligen versaler samt understreck).

Även i användning påminner de mycket om klasskonstanter. Det fullständiga namnet för ett värde av en uppräknad datatyp är datatypens namn, punkt samt värdets namn:

```
Letter ltr = new Letter(PostageType.ECONOMY, 175);
```

Språkdefinitionen garanterar att jämförelse med såväl likhetsoperatorn == som instansmetoden `equals` fungerar:

```
if (ltr.getPostageType() == PostageType.NORMAL) …
```

Det går att skriva ut värden av en uppräknad datatyp, t ex åstadkommer satsen

```
System.out.print(PostageType.INSURED);
```

att texten `INSURED` skrivs ut. Det går även att gå åt andra hållet: klassmetoden `valueOf` kan returnera det värde av en uppräknad datatyp som skrivs ut med den textsträng som argumentet anger:

```
PostageType pt = PostageType.valueOf("ECONOMY");
```

Uppräknade datatyper

switch-satsen

Värden av uppräknade datatyper får användas som alternativ i `switch`-satser.

I `switch`-satsen ska endast värdenas korta namn användas:

```
PostageType letterType = ltr.getPostageType();
int daysToDeliver;

switch (letterType) {
    case ECONOMY:  daysToDeliver = 5;
                   break;

    case NORMAL:   daysToDeliver = 2;
                   break;

    case INSURED:  daysToDeliver = 3;
                   break;

    case EXPRESS:  daysToDeliver = 1;
                   break;

    default:       daysToDeliver = 2;
                   break;
}
```

Trots att värden av uppräknade datatyper formellt är objektreferenser till objekt av en speciell form av klass, får de användas som alternativ i `switch`-satser.

Det normala vid användning av värden från uppräknade datatyper är att vi alltid måste ange typens namn som prefix:

```
if (ltr.getPostageType() == PostageType.ECONOMY) …
```

`switch`-satsen utgör emellertid ett undantag från denna regel, eftersom värdena här endast ska anges med sina korta namn. Det är inte ens tillåtet att använda de fullständiga namnen:

```
switch (ltr.getPostageType()) {

   case ECONOMY:  daysToDeliver = 5;
                  break;

   ...
}
```

Precis som för vanliga `switch`-satser är det tillrådligt att ta med ett `default`-alternativ, som exekveras om inget annat alternativ stämmer med det aktuella värdet.

Uppräknade datatyper

Iteration över samtliga värden

Klassmetoden `values` returnerar en vektor med samtliga värden:

```
PostageType[] allValues = PostageType.values();
```

Detta gör det enkelt att iterera över samtliga värden för en uppräknad datatyp:

```
for (PostageType type : PostageType.values()) {
    System.out.print("Giltigt v\u00e4rde: ");
    System.out.println(type);
}
```

Notera att denna kod fortfarande fungerar om typen byggs ut med nya värden!

Varje uppräknad datatyp har automatiskt en klassmetod `values` som returnerar en vektor bestående av samtliga giltiga värden:

```
PostageType[] allValues = PostageType.values();
```

Denna kan bl a användas för att på ett enkelt sätt skapa en slinga där koden itererar över samtliga giltiga värden för en uppräknad datatyp:

```
for (PostageType type : PostageType.values())
    System.out.println("Giltigt v\u00e4rde: " + type);
```

Detta kan givetvis även uttryckas utan den förenklade `for`-satsen:

```
PostageType[] allValues = PostageType.values();
for (int i = 0; i < allValues.length; i++) {
    PostageType type = allValues[i];
    System.out.println("Giltigt v\u00e4rde: " + type);
}
```

Lägg märke till att dessa slingor inte behöver ändras om uppsättningen av giltiga värden förändras för den uppräknade datatypen! Detta var ett av de problem vi såg när vi använde publika klasskonstanter.

Det finns även en instansmetod `ordinal` som returnerar positionsnumret (räknat från noll) för ett giltigt värde inom den uppräknade följden. Denna metod bör emellertid undvikas, eftersom positionsnumren kan förändras om den uppräknade datatypen ändras.

Uppräknade datatyper

Instansvariabler och instansmetoder

PostageType.java

```
public enum PostageType {

    ECONOMY(5, 2000),
    NORMAL(2, 2000),
    INSURED(3, 1000),
    EXPRESS(1, 1000);

    private int myDeliveryTime;
    private int myMaxWeight;

    PostageType(int daysToDeliver, int maxWeight) {
        myDeliveryTime = daysToDeliver;
        myMaxWeight = maxWeight;
    }

    public int getDeliveryTime() {
        return myDeliveryTime;
    }

    public int getMaxWeight() {
        return myMaxWeight;
    }
}
```

Vi har redan nämnt att uppräknade datatyper egentligen är klasser, och att de olika värdena är referenser till i förväg skapade instanser av klassen. Därför är det tillåtet att förse definitioner av uppräknade datatyper med deklarationer av instansvariabler samt implementationer av metoder.

I detta exempel har vi försett definitionen av den uppräknade datatypen `PostageType` med två instansvariabler, som ska lagra antalet dagar det tar att förmedla en försändelse av den aktuella typen samt den maximala vikten för försändelsetypen:

```
private int myDeliveryTime;
private int myMaxWeight;
```

Vi har också tillfört två frågemetoder, `getDeliveryTime` resp `getMaxWeight`, för att andra objekt ska kunna få reda på den här informationen.

Vi har även möjligheten att deklarera konstruktorer. Dessa får dock inte vara `public`-deklarerade, eftersom det i så fall skulle innebära att det blev tillåtet att skapa nya instanser av klassen:

```
PostageType(int daysToDeliver, int maxWeight) { … }
```

Konstruktorer för uppräknade datatyper är underförstått `private`-deklarerade, vilket får men inte måste skrivas ut.

Listan av giltiga värden måste skrivas före deklarationer av instansdata och metoder i en definition av en uppräknad datatyp. Den kommaseparerade listan måste dessutom avslutas med semikolon. Om värden i listan ska initieras av en konstruktor med parametrar, skrivs argumentlistan efter värdets namn:

```
ECONOMY(5, 2000), NORMAL(2, 2000), INSURED(3, 1000), EXPRESS(1, 1000);
```

Uppräknade datatyper

Värdespecifika implementationer

PostageType.java

```
public enum PostageType {

    ECONOMY(5, 2000) {
        public double getPostage(int weight) {
            if (weight <= 20) return 5.50;
            else if (weight <= 100) return 11.00;
            else if (weight <= 250) return 22.00;
            else return 66.00;
        }
    },
    ...
    EXPRESS(1, 1000) {
        public double getPostage(int weight) {
            return 150.00;
        }
    };

    ...

    public int getMaxWeight() { return myMaxWeight; }

    public abstract double getPostage(int weight);
}
```

Språkdefinitionen tillåter även att en eller flera metoder ges en unik implementation per giltigt värde. Sådana metoder måste deklareras som abstrakta metoder i definitionen av den uppräknade datatypen:

```
public abstract double getPostage(int weight);
```

I listan av värden ska nu namnet på varje värde och den eventuella argumentlistan till konstruktorn följas av en *värdespecifik implementation* bestående av en eller flera metoder samlade inom en inledande och en avslutande klammer:

```
ECONOMY(5, 2000) {
    public double getPostage(int weight) {
        if (weight <= 20)
            return 5.50;
        else if (weight <= 100)
            return 11.00;
        else if (weight <= 250)
            return 22.00;
        else
            return 66.00;
    }
},
```

Varje giltigt värde ska ha en helt egen implementation av dessa metoder.

Syntaxen för värdespecifika implementationer är ganska svårläst. Ofta är det möjligt att välja andra alternativ som är enklare att förstå.