Objektorienterad Programmering DAT043

Föreläsning 4
23/1 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Repetion: Klasser och objekt

Vi definierar ***nya objekttyper i klasser***. Ett objekt med typen som definierats i en klass kallas ofta en *instans av klassen*.

- **Instansvariabler** - en uppsättning *per objekt av typen*.
- **Konstruktor-metoder** - instantierar instansvariabler när vi skapar ett nytt (`new`) objekt av typen. Samma namn som klassen.
- **Klass- eller statiska variabler** - en *per klass*. Delas av alla objekt.
- **Instansmetoder** - kallas alltid *på ett objekt* av klassens typ: `obj.instansmetod(argument)`. Här blir `obj` som ett extra (implicit) argument till metoden.
- **Klass- eller statiska metoder** - hör till klassen. Kallas med klassnamnet som prefix: `Klassnamn.statiskMetod(argument)`

Överlagring

- Java tillåter att man har flera metoder i samma namn i samma klass om:
  - De har olika antal argument, eller
  - Argumenten har olika typ.
- Vanligt för konstruktorer, men även tillåtet för övriga metoder.

```
Person p1 = new Person("Alice", "Alicedotter", 60, 1.60, 60.5);
Person p2 = new Person(“Bob”, “Bobsson”);
```


Metoder med samma namn, men med olika argument (antal/typer) kan skiljas åt av kompilatorn. Vi säger att metoderna är överlagrade. Detta är mycket vanligt för konstruktor-metoder, men även tillåtet för andra metoder.

Dagens föreläsning

- Arv och hierarkier av klasser (inheritance).
- Gränssnitt (interfaces).
- Undantag (exceptions) och I/O.

Arv

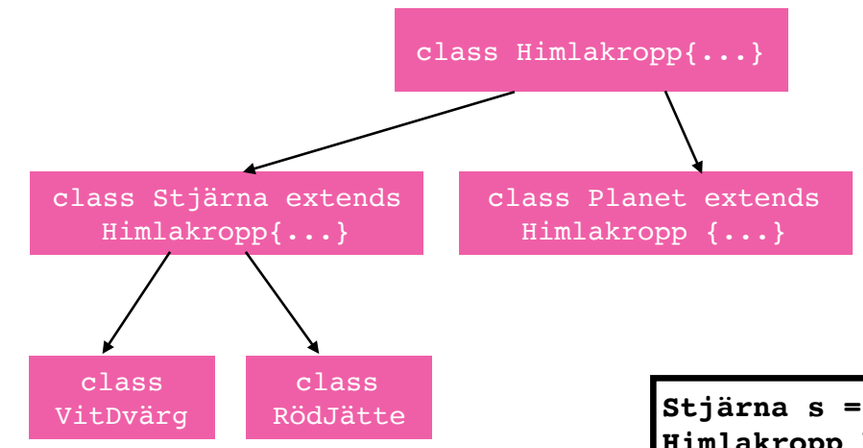
- **Arv:** mekanisk för att *gruppera och återanvända* kod.
- En klass (*subklass*) kan ärva från en annan klass (*superklass*):
  - Subklassen “inkluderar” alla variabler och metoder från superklassen.
  - Kan sedan lägga till t.ex. ytterligare variabler/metoder.

```
class Atlet extends Person {
   // utöver de instansvariabler vi ärver från Person
  private double syreupptagning;
  ...
}
```


Arv (inheritance) är en viktig mekanisk för att gruppera och återanvända kod. En klass som ärver en annan har automatiskt alla variabler och metoder som förälderklassen har. Förälderklassen kallas superklass och klassen som ärver subklass. I klassen som ärver kan ytterligare variabler och metoder läggas till.

Exempel: Anta att vi vill skriva en klass Atlet för en sportapp. Appen ska hålla reda på t.ex. längd, vikt, BMI och syreupptagningsförmåga. Vi har tidigare skrivit klassen Person (se förra föreläsningen). Här är det lämpligt att låta Atlet bli en subklass till Person. Då behöver vi bara lägga till en instansvariabel för syreupptagning, och kan återanvända resten av koden vi redan skrivit för klassen Person!

Arv




```
Stjärna s = new Stjärna(...);
Himlakropp h = s; // OK!
```

```
Stjärna s = new Stjärna(...);
Himlakropp h = s;
Stjärna s2 = h; // Inte OK!
```

```
Stjärna s = new Stjärna(...);
Himlakropp h = s;
Stjärna s2 = (Stjärna) h; // ok.
```

Om en klass B ärver en klass A så är ett objekt av klassen B också ett objekt av klassen A. Anta att vi har en klass Himmlakropp och två klasser Stjärna och Planet som ärver denna. Stjärna har i sin tur ytterligare två subklasser: VitDvärg och RödJätte. Både stjärnor och Planeter är Himlakroppar.

Eftersom objekt av typ Stjärna har alla instansvariabler och metoder som Himlakroppar har är det första exemplet OK (vi kan helt enkelt strunta i de extra variabler/metoder en stjärna har om vi vill behandla den som en Himlakropp). Däremot går det inte att göra det omvända automatisk, då klagar kompilatorn (andra exemplet). Det är inte alltid säkert att behandla en Himlakropp som om det vore en Stjärna (det skulle ju t.ex. kunna röra sig om en Planet - kompilatorn vet inte detta). Om man som programmerare vet att det rör sig om en Stjärna kan man tvinga kompilatorn att acceptera koden genom en explicit typomvandling (exempel 3). När programmet körs görs dock en explicit kontroll att det verkligen är en Stjärna, om inte ger programmet ett felmeddelande.

Klasserna i Javas API bildar på samma sätt en hierarki.

Vi kommer återkomma till arv senare. Då kommer vi prata om hur man skriver klasser som ärver varandra. Det väsentliga nu är att känna till att denna mekanism finns eftersom klassernas i Javas API utgör en hierarki där klasser ärver varandra på flera nivåer.

Klassen `Object`

- **Alla** klasser i Java ärver implicit grundklassen **`Object`**.
- De metoder som definieras i klassen `Object` ärvs alltså av alla klasser vi definerar själva. T.ex. `toString()`, `equals(...)`.

```
Stjärna s = new Stjärna(...);
Person p = new Person(...);
String str = "Jag är ett objekt";

p.toString(); //ärvs från Object
```

```
Object o1 = s;
Object o2 = p;
Object o3 = str;
```


Alla klasser i Java ärver implicit en grundklass som heter Object. (visa API)

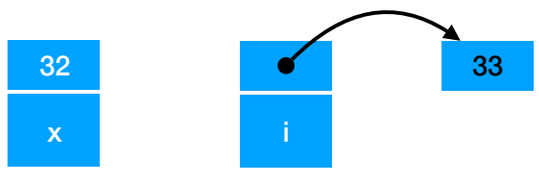
Det går alltså även att behandla objekt av alla typer som om dom hade typen Object, enligt principen på förra sliden.

Notera: Om vi försöker "skriva ut" ett objekt med System.out.println så kallas metoden toString, som ärvts av superklassen Object. Den skriver ut objektets s.k. hash-kod, vilker förmodligen inte var vad vi förväntade oss! Vi kan dock skriva en mer specifik toString()-metod för våra egna klasser som "överskuggar" den vi ärver från Object. Mer om detta senare i kursen.

Primitiva typer som Objekt

- Ibland är det praktiskt att “packetera” primitiva typer som objekt.
- Skicka som referenser eller använda null.
- Finns sammanhang där det inte går att använda primitiva typer (mer senare).
- Klasser med motsvarande namn:
  - `int` ---> `Integer`
  - `boolean` ---> `Boolean`
  - `double` ---> `Double` osv.

```
Integer i = new Integer(32);  // integer som objekttyp.
int x = i;                    // automatisk omvandling (unboxing)
i = x + i;                    // också OK (boxing)
```



Notera att Integer inte kan lagra godtyckligt stora tal som typen med samma namn i Haskell. Det är bara en klass med en instansvariabel som är av typen int.

Dessa klasser har en del statiska metoder, t.ex. parseInt och parseDouble som vi sett. Men man kan skapa objekt av dessa klasser.
Man har nytta av dessa klasser när man vill kunna skicka värden som referenser eller kunna ge variabler värdet null. Mest har man nytta av det i vissa sammanhang (som vi kommer senare) där det inte är möjligt att använda de primitiva typerna.
Java gör automatisk omvandling mellan primitiva värden och motsvarande objekt. Detta kallas boxing/unboxing.

Klassen `Math`

- All Java-kod skrivs i klasser.
- Vanligtvis beskriver varje klass en objekttyp samt tillhörande metoder.
  - T.ex. klassen Person från förra föreläsningen.
- Dock finns undantag, t.ex. bibliotheksklassen `Math` som samlar matematiska funktioner för primitiva typer.
  - Ingen konstruerare.
  - Enbart statiska klassmetoder och konstanter.
- Ett annat undantag är våra test-klasser som vi sett i programmeringsexempel.

All Javakod måste skrivas i en klass. Vanligtvis beskriver varje klass en typ, med associerade konstruktormetoder och instansmetoder. Men det finns även klasser som inte beskriver någon objekttyp, utan bara fungerar för att samla ett antal statiska metoder, t.ex. Math. Dessa klasser skapar man inte objekt utav, eftersom de inte beskriver någon typ.

Notering: Om vi tittar i Javas API ser vi att klassen Math har attributet final. För klasser betyder detta att vi inte kan skapa subklasser som ärver från Math. Vilket verkar rimligt, eftersom det inte är tänkt att vi ska skapa några objekt av typ Math heller!

Gränssnitt (Interfaces)

- Abstrakta klasser: specificerar enbart vilka metoder som ska finnas (headers). Påminner of type-classes i Haskell.
- För att kunna användas måste (minst en) klass *implementera* gränssnittet.
- Liksom med Haskells typklasser kan en Javaklass implementera flera interface.

```
interface Show{
  // signatur för en metod toString.
  // Signaturen anger returtyp, metod namn och argument.
  // Ingen implementation!
 String toString();
 }
```

```
class Person implements Show{
 ...
 String toString(){...} //Konkret implementationskod här!
```

Gränssnitt (interface) i Java är besläktade med klasser. De kan ses som abstrakta klasser på det viset att de bara säger vilka metoder som ska finnas. Metoderna definieras inte och inga klassvariabler anges. Ett interface specificerar alltså bara gränssnittet, inte hur representationen av objekten ska implementeras. För att kunna använda ett gränssnitt måste det finnas en klass som implementerar det.

Interface är det som liknar typklass-mekanismen i Haskell mest. Om vi tar klasserna Show och Eq i Haskell som exempel så skulle vi kunna definiera två interface i Java med samma namn. Interfacet Show skulle innehålla signaturen för en metod som skapar en sträng som återspeglar objektets värde. Interfacet Eq skulle innehålla signaturen för en metod som jämför objekt med varandra. En klass som sedan vill implementera någon av dessa interface behöver bara definiera vad motsvarande funktion ska göra. Notera att grundklassen Object innehåller metoderna toString och equals så interface motsvarande Show och Eq i Java är i praktiken överflödiga.

En klass kan implementera flera interface.

Detta kommer vi också återkomma till och nämns nu för att interface i Javas API är vanligt förekommande.

Undantag (exceptions)

- Hantera onormala situationer och resultat.
- Exempel: I/O-problem, filer som inte finns.
- Undantag är klasser som ärver från grundklassen **`Exception.`**
- Skapas med nyckelordet `throw.` Enbart objekt av typ som ärver från `Exception` kan kastas.

```
throw new Exception();

throw new Exception(“Mitt felmeddelande");
```

```
throw new ArrayIndexOutOfBoundsException(...);

throw new IllegalArgumentException()..;
```

Undantag (exceptions) är en mekanism i Java för att hantera onormala situationer och resultat. Vad man väljer att se som onormal är förstås inte uppenbart.

Metoder som har att göra med input och output till filer har man valt att hantera I/O-problem (som att filen inte finns) med hjälpa av undantag. Det skulle också kunna vara så att dessa metoder returnerar ett speciellt värde när fel uppstår, t.ex. null där det är lämpligt.

Närmsta motsvarigheten till undantag i Haskell är error.

Undantag är en hierarki med klasser där Exception är grundklassen.
För att skapa (kasta, throw) ett undantag använder man nyckelorder throw. Objektet man skapar måste vara en Exception eller en underklass till denna (man kan t.e.x inte skriva throw new String). Exception har flera konstruerare. Vanligt är att använda en som låter dig beskriva felet som uppstått med en sträng.

Om man vill kunna lagra information av specifik typ kan man själv skapa en underklass till Exception. Detta görs också ofta enbart för att ge undantagstypen ett beskrivande namn, t.ex. IllegalArgumentException, ArrayIndexOutOfBoundsException osv.

Kolla upp dessa klasser i Javas API!

try-catch

- Man kan fånga undantag för att undvika att programmet kraschar.
- `catch (Exception e)` fångar **alla** olika undantag, eftersom alla undantag ärver från `Exception`.

```
try {
  //koden som kan kasta undantag
} catch (Exception e) {
  //koden som hanterar undantaget
}
```

Precis som i Haskell där man kan fånga errors så kan man fånga undantag så att de inte orsakar programstopp och felmeddelanden. Detta gör man genom att omge koden som kan orsaka undantaget med en try-catch konstruktion.

Koden som kan kasta undantag kan innehålla flera ställen som kan kasta undantag. En vinst med undantag är att, istället för att behöva kontrollera för t.ex. varje läs- eller skrivoperation om något gick snett, kunna lägga flera sådana operationer i ett try-block och bara behöva hantera att något gått snett en gång.

Om man anger klassen Exception efter catch så fångas alla undantag eftersom alla undantag är en Exception. Det kan vara lämpligt att vara mer specifik i sin kod, och bara fånga de exceptions om man verkligen förutspår kan dyka upp och som man kan hantera på ett lämpligt sätt.

Om man anger t.ex. klassen IllegalArgumentException så fångas undantag av denna typ (och de som tillhör en underklass). Andra undantag som kastas i try-blocket skickas vidare uppåt i stacken av anropande metoder. Om inget try-catch-block fångar ett undantag så avbryts programexekveringen och undantaget skrivs ut.

Checked/unchecked exception

- **Checked:** varje metod som inte hanterar checkade undantag *måste explicit tala om att undantaget kan kastas*.
- **Unchecked:** metoder behöver inte hantera dem, eller deklarera att de kan kastas. E.g. `RuntimeException` och underklasser.

```
// Från Labb 1:
public static void main(String[] args) throws Exception
```

```
public static double maxDoubles(double[] a) {
  double max = a[0]; // Kan kasta ArrayIndexOutOfBounds
  for (int i = 1; i < a.length; i++) {
    if (a[i] > max) max = a[i];
  }
  return max; }
```

Det finns två sorters undantag, checked och unchecked. Den första sorten måste man för varje metod som inte hanterar dem tala om att de kan komma att kastas.

I labb 1 används följande signatur för main-metoden:
public static void main(String[] args) throws Exception

Detta för att inläsning av en fil, som man ombeds utföra, kan kasta undantag som är av typen checked. Meningen är att det tydligt ska synas, precis som det tydligt syns vilken sorts värde som returneras, att metoden kan kasta ett undantag. (Här borde man kanske egentligen vara mer tydlig med exakt vilken typ av exception som kan kastas).

För den andra sorten, unchecked, behöver man inte tala om att en metod kan kasta dem. Alla undantag som hör till klassen RuntimeException eller dess underklasser är unchecked. Alla undantag som inte gör det är checked.

Metoden maxDoubles letar rätt på maxvärdet i en array av doubles. Den fungerar bara om arrayen innehåller minst ett element. Om man anropar metoden med en array av längden 0 som argument så kommer första raden orsaka ett undantag, ett ArrayIndexOutOfBoundsException. Detta undantag är underklass till RuntimeException - alltså uncheckat. Därför måste man inte skriva throws ArrayIndexOutOfBoundsException i metodens signatur.

DEMO: maxDoubleDemo.java

```
public static double maxDoubles(double[] a) {
  if (a.length == 0) {
    throw new IllegalArgumentException(
     "maxDoubles: argument a must contain at least one
element");
  }
  double max = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > max) max = a[i];
  }
  return max;}
```

```
public static double maxDoubles(double[] a) {
  try {
    double max = a[0];
    for (int i = 1; i < a.length; i++) {
      if (a[i] > max) max = a[i];
    }
    return max;
  } catch (IndexOutOfBoundsException e) {
    throw new IllegalArgumentException(
     "maxDoubles: argument a must contain at least one
element");
  }}
```

För en användare av metoden kan det vara bättre att få en förklaring till varför argumentet inte är giltigt.

Ett alternativ är att låta IndexOutOfBoundsException uppstå, fånga det och kasta ett IllegalArgumentException. Vi kan då använda ett try-catch block.

När ett undantag inträffer i try-blocket avbryts exekveringen direkt av blocket. Men programmet kan befinna sig i ett tillstånd där den har resurser vars användning bör avslutas på korrekt sätt. Det kan t. ex. röra sig om öppna filer som behöver stängas. För att kunna hantera detta kan try-catch-satsen ha finally-block i slutet. Detta block exekveras oavsett om ett undantag uppstått.

Om man vill skilja på hanteringen av olika typer av undantag så kan man inkludera flera catch-block.

Övning: I klassen Person från förra föreläsningen använde vi negativa tal för att representera okända värden på längd/vikt. Modifiera metoden bmi() så att den kastar ett IllegalArgumentException med ett lämplig felmeddelande ifall den kallas på ett objekt med odefinierad längd eller vikt.

I/O

- Paketet java.io innehåller klasser och metoder för filåtkomst.
- Klassen Scanner i java.util är lämplig för att läsa in textfiler.

```
File infil = new File("in.txt");
```

```
Scanner sc = new Scanner(infil);
```

```
try {
    File infil = new File("in.txt");
    Scanner scanner = new Scanner(infil);
    System.out.println("Filen är öppen");
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Vi har sett exempel på att läsa från en fil genom att använda java.nio.file.Files.readAllBytes. java.nio är ett paket för filhantering på låg nivå. Vi ska nu titta på några klasser och metoder i paketet java.io som tillåter filåtkomst på högre nivå. Klassen File är en abstrakt representation av sökvägar till filer och kataloger, liknande Path i java.nio.

Om man vill läsa in från eller skriva till filer binärt så kan man använda klasserna FileInputStream, FileOutputStream. Med dessa kan man läsa och skriva sekvenser av bytes.

Här ska vi fokusera på att läsa och skriva textfiler. För att läsa textfiler använder man lämpligen klassen Scanner, som finns i paketet java.util.

Konstrueraren till Scanner kastar undantaget FileNotFoundException om filen inte finns. Vi måste hantera undantaget på något sätt (det är ett s.k. checked exception). Låt oss göra det med try-catch.

DEMO: I/O.java

I/O: Läsa textfiler

- Läser in en *token* åt gången.
- Skiljs åt med s.k. *delimeters* (här mellanrum)

```
try {
    File infil = new File(“in.txt");
    Scanner scanner = new Scanner(infil);

    while (scanner.hasNext()) { //Finns fler tokens?
        double x = scanner.nextDouble(); // Om ja, läs nästa tal
        System.out.println(Math.sqrt(x));
    }
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Inläsning via Scanner bygger på att läsa token i taget och vad som är ett token definieras av en avgränsare (delimiter). Förvald delimiter är mellanrum. Man kan avgöra om det finns fler tokens med metoden hasNext.

Man kan läsa in olika typer av värden med metoderns next..., t.ex. nextDouble. Man kan läsa in en hel rad med nextLine.
Låt oss läsa in flyttal så länge filen inte är slut.

I/O: hantera undantag

- Om filen in.txt innehåller något annat än decimaltal kastas ett `InputMismatchException.`
- Unchecked, ett `runtimeException`. Alltså måste vi inte fånga det (men det kan vara lämpligt ändå).

```
try {
    File infil = new File("in.txt");
    Scanner scanner = new Scanner(infil);

    while (scanner.hasNext()) {
        double x = scanner.nextDouble();
        System.out.println(Math.sqrt(x));
    }
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (InputMismatchException e) {
    System.out.println("Indata måste vara flyttal.");
}
```

Eftersom InputMismatchException ärver från RuntimeException får vi ingen kompilator-varning om att det kan ske. Det är ett s.k. unchecked exception. Det kan dock vara lämpligt att hantera ändå. Vi kan som nämnts ha flera olika catch-block för olika typer av exceptions.

I/O: `finally`-block

- När vi är färdiga med en fil bör vi stänga scannern med metoden: `scanner.close()`.
- Vad händer om ett undantag kastas och vi avbryter exekvering av ett try-block?
- finally-block exekveras alltid sist

```
Scanner scanner = null;
try {
    File infil = new File("in.txt");
    scanner = new Scanner(infil);

    while (scanner.hasNext()) {
        double x = scanner.nextDouble();
        System.out.println(Math.sqrt(x));
    }
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (InputMismatchException e) {
    System.out.println("Indata måste vara flyttal.”);
} finally { //scanner är null om filen inte gick att öppna.
  if (scanner != null) scanner.close();
}
```

Ofta uppstår inget problem om man glömmer att stänga en fil, men det är säkrast och snyggast att göra det explicit så fort man är klar med filen.

```
File infil = new File("in.txt");
Scanner scanner = new Scanner(infil);

while (scanner.hasNext()) {
    double x = scanner.nextDouble();
    System.out.println(Math.sqrt(x));
}

scanner.close();
```

Men om det uppstår ett undantag när filen är öppen så stängs den aldrig explicit eftersom exekveringen av try-blocked då avbryts. Vi kan använda ett finally-block för att hantera detta. finally-block exekveras alltid sist, efter eventuell hantering av undantag i catch-blocken.

Om ett undantag uppstod vid öppnandet av filen så är inte filen öppen och kan heller inte stängas. Denna situation motsvaras av att scanner är null.

I/O: Skriva textfiler

```
Scanner scanner = null;
PrintWriter writer = null;
try {
    File infil = new File("in.txt");
    scanner = new Scanner(infil);

    File utfil = new File("ut.txt");
    writer = new PrintWriter(utfil);

    while (scanner.hasNext()) {
        double x = scanner.nextDouble();
        writer.println(Math.sqrt(x));
    }
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (InputMismatchException e) {
    System.out.println("Indata måste vara flyttal.");
} finally {
  if (scanner != null) scanner.close();
  if (writer != null) writer.close();
}
```

Om vi vill skriva resultatet till en textfil istället för standard output så kan vi använda klassen PrintWriter.

Även här bör vi stänga utfilen i finally blocket.

for-each loopar

```
int[] a = ...;

for (int e : a) {
  ...
}
```

```
public static boolean member(int n, int[] a) {
  for (int e : a) {
    if (e == x) return true;
  }
  return false;
}
```

for-each loopar är en variant på for-loopar där man på ett smidigt sätt kan utföra något för varje element i en array.

Loopen upprepas en gång för varje element i a och för varje iteration är x värdet på aktuellt element. Man slipper använda ett index, ange vilka tal det ska löpa mellan och slå rätt element i arrayen.

Switch-satser

- Anta att heltal representerar veckodagar. Vi vill skriva ut öppentider:

```
public static void skrivÖppettider(int veckodag) {
  // 0 <= veckodag <= 6
  String s;
  if (veckodag <= 3) {  // måndag till torsdag
    s = "10:00 - 20:00";
  } else if (veckodag == 4) {  // fredag
    s = "10:00 - 19:00";
  } else if (veckodag == 5) {  // lördag
    s = "12:00 - 18:00";
  } else {  // söndag
    s = "12:00 - 16:00";
  }
  System.out.println(s);
}
```

Låt oss säga att vi använder heltal för att representera veckodagar. Låt 0 vara måndag, 1 tisdag, … och 6 vara söndag. Om vi sedan vill ha en metod som skriver ut öppettider för en affär för en viss veckodag så kan den se ut som ovan.

Långa nästlade if-else-if-else... kan bli svåröverskådliga. Därför kan man istället använda en s.k. switch sats...

Switch-satser

```
public static void skrivÖppettider(int veckodag) {
  String s;
  switch (veckodag) {
    case 0:
    case 1:
    case 2:
    case 3:
      s = "10:00 - 20:00";
      break;
    case 4:
      s = "10:00 - 19:00";
      break;
    case 5:
      s = "12:00 - 18:00";
      break;
    case 6:
      s = "12:00 - 16:00";
      break;
  }
  System.out.println(s);
}
```

Fallen 0 - 2 gör ingenting, så där fortsätter vi bara nedåt tills vi kommer till fall 3, där vi skriver ut öppentiderna som gäller mån-tors. Notera alltså att utan en break-sats fortsätter exekveringen förbi nästa case.

Switch-satser

```
public static void skrivÖppettider(int veckodag) {
  String s;
  switch (veckodag) {
    case 4:
      s = "10:00 - 19:00";
      break;
    case 5:
      s = "12:00 - 18:00";
      break;
    case 6:
      s = "12:00 - 16:00";
      break;
    default:  // måndag - torsdag
      s = "10:00 - 20:00";
      break;
  }
  System.out.println(s);
}
```

Det finns också något som motsvarar else. Det heter default och utförs om värdet ej motsvarar någon av fallen. Föregående switch-sats kan alltså också skrivas som ovan. Koden blir nu kortare och mer kompakt, men vad händer om någon ger metoden ett argument som egentligen inte representerar någon veckodag (t.ex. 8? Ska det tolkas som måndag igen? Eller 12, är det fredag nästa vecka?).

Switch satser kan användas för värden av de primitiva typerna byte, short, char och int. De fungerar också med de boxade varianterna av dessa typer (Byte, Short, Character, Integer) och med String.

Enumereringar

```
public enum Veckodag {
  MÅNDAG, TISDAG, ONSDAG, TORSDAG, FREDAG, LÖRDAG, SÖNDAG
}
```

```
public static void skrivÖppettider(Veckodag veckodag) {
  String s;
  switch (veckodag) {
    case MÅNDAG:
    case TISDAG:
    case ONSDAG:
    case TORSDAG:
      s = "10:00 - 20:00";
      break;
    case FREDAG:
      s = "10:00 - 19:00";
      break;
    case LÖRDAG:
      s = "12:00 - 18:00";
      break;
    case SÖNDAG:
      s = "12:00 - 16:00";
      break;
  }
  System.out.println(s);
}
```

Enumereringar kan man användas för att göra betydelsen av olika variabelvärden tydligare och värdena anpassade till situationen. Kallas även uppräkningstyper, och är typer som innehåller ett fixerat antal värden (här sju) till skillnad från “vanliga” objekttyper där man, i princip, kan skapa hur många olika instanser som helst.

Att som i förra avsnittet representera veckodagar med heltal kan lätt leda till kryptisk kod. Då är det bättre att definiera en enumerering. Switch-satser kan, förutom de typer som nämndes i förra avsnittet, användas med enumereringar.

Man kan deklarera variabler av denna typ och de olika värderna kan man hänvisa till genom att kvalificera dem med enumereringens namn:
Veckodag dag = Veckodag.MÅNDAG;

Man kan också använda == för att avgöra om två värden av en enumerering är lika.