Föreläsning 8

Undantag
Designmönstert Strategy
Gränssnittet Comparable
Gränssnittet Comparator

Fel i program

När man skriver program uppkommer alltid olika typer av fel:

- *Kompileringsfel*, fel som beror på att programmeraren bryter mot språkreglerna. Felen upptäcks av kompilatorn och är enkla att åtgärda. Kompileringsfelen kan indelas i *syntaktiska fel* och *semantiska fel*.
- *Exekveringsfel*, fel som uppkommer när programmet exekveras och beror på att ett uttryck evalueras till ett värde som ligger utanför det giltiga definitionsområdet för uttryckets datatyp. Felen uppträder vanligtvis inte vid varje körning utan endast då vissa specifika sekvenser av indata ges till programmet.
- *Logiska fel*, fel som beror på ett tankefel hos programmeraren. Exekveringen lyckas, men programmet gör inte vad det borde göra.

Alla utvecklingsmiljöer (JGrasp, Eclipse, NetBeans, etc) tillhandahåller verktyg för debugging, lär dej att använda dessa i den utvecklingsmiljö du använder.

Exceptionella händelser

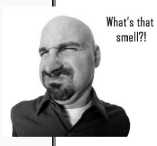
Fel som uppstår när programmet exekveras ger upphov till *exceptionella händelser* (kallas även för *undantag*).

En exceptionell händelse är en oväntad felsituation i ett program som normalt inte skall inträffa (och är ofta utanför programmets direkta ansvarsområde).

Denna typ av felsituationer är vanligtvis svåra att gardera sig emot på normalt sätt när vi utvecklar program. Vi skulle då tvingas att lägga in kontroller för alla tänkbara typer av fel.

Skicka felkoder: dålig lösning

```
public class  PurchaseOrder {
   public static final double ERROR_CODE1 = 1.0;
   private Product prod;
   private int quantity;
   . . .
   public double totalCost() {
      if (quantity < 0)
         return ERROR_CODE1;
      else
         return prod.getPrice() * quantity;
   }//totalCost
}//PurchaseOrder
```



Problem:

- Det finns oftast många olika typer av fel, d.v.s. många felkoder.
- Felkoderna måste ha samma typ som metoden normalt returnerar.
- Leder till att koden blir komplex och svårläst.

Orsaker till exceptionella händelser

Några orsaker till exceptionella händelser:

- Programmeringsfel (refererar till ett objekt som inte finns, adresserar utanför giltiga index i ett fält, …).
- Användaren av vår kod har inte läst dokumentationen (anropar metoder på fel sätt).
- Resursfel (nätverket gick ner, hårddisken är full, minnet är slut, databasen har kraschat, DVD:n var inte insatt, ...).

I Java finns en speciell mekanism för att hantera undantag. Ett undantag genereras av den del av koden som detekterar felet, men tas ofta om hand av en annan del av koden där man har bättre förutsättningar att analysera felet och vidta lämpliga åtgärder.

Hantera exceptionella händelser

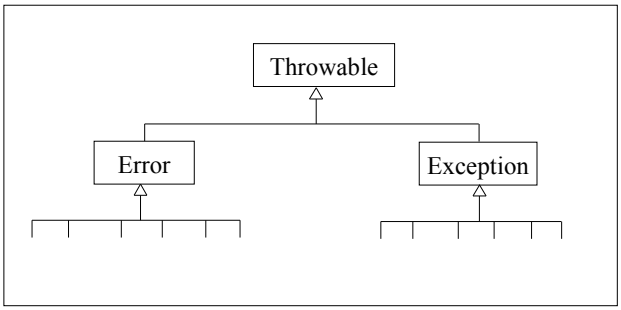
För att handha exceptionella händelser tillhandahåller Java:

- konstruktioner för att "kasta" exceptionella händelser (konstruktionen **throw**)
- konstruktioner för att "specificera" att en metod kastar eller vidarebefordrar exceptionella händelser (konstruktionen **throws**)
- konstruktioner för att fånga exceptionella händelser (konstruktionerna **try**, **catch** och **finally**).

Felhanteringen blir därmed en explicit del av programmet, d.v.s. felhanteringen blir synlig för programmeraren och kontrolleras av kompilatorn.

Klasshierarkin för exceptionella händelser

En exceptionell händelse beskrivs med hjälp av ett objekt som tillhör någon subklass till standardklassen `Throwable`.
Det finns två standardiserade subklasser till `Throwable`: klasserna `Exception` och `Error`.



Klasserna `Error` och `Exception`

Klassen `Error` används av Java Virtual Machine för att signalera allvarliga interna systemfel t.ex. länkningsfel, hårddisken är full och minnet tog slut.

Vanligtvis finns inte mycket att göra åt interna systemfel. Vi kommer inte att behandla denna typ av fel ytterligare.

De fel som hanteras i "vanliga" program tillhör subklasser till `Exception`. Denna typ av fel orsakas av själva programmet eller vid interaktion med programmet. Sådana fel kan fångas och hanteras i programmet.

Subklasser till klassen Exception

I Java's standardbibliotek finns en rad subklasser till klassen `Exception` definierade, varav några är:

`AclNotFoundException`	`ActivationException`
`AlreadyBoundException`	`ApplicationException`
`AWTException`	`BadLocationException`
`ClassNotFoundException`	`CloneNotSupportedException`
`DataFormatException`	`ExpandVetoException`
`FontFormatException`	`GeneralSecurityException`
`IllegalAccessException`	`InstantiationException`
`InterruptedException`	`IntrospectionException`
`IOException`	`LastOwnerException`
`LineUnavailableException`	`MidiUnavailableException`
`MimeTypeParseException`	`NamingException,`
`NoninvertibleTransformException`	`NoSuchFieldException`
`NoSuchMethodException`	`NotBoundException`
`NotOwnerException`	`ParseException`
`PrinterException`	`PrivilegedActionException`
`PropertyVetoException`	`RemarshalException`
`RuntimeException`	`ServerNotActiveException`
`SQLException`	`TooManyListenersException`
`UnsupportedAudioFileException`	`UserException`

Exempel på exceptionella händelser (1)

Exempel 1:

Betrakta nedanstående programsegment:

```
String input = JOptionPane.showInputDialog("Ange ett heltal: ");
int value1 = Integer.parseInt(input);
input = JOptionPane.showInputDialog("Ange ytterligare ett heltal: ");
int value2 = Integer.parseInt(input);
int result = value1 / value2;
```

Om vi vid inläsningen ger en sträng som inte kan tolkas som ett heltal erhålls ett undantag av typen `java.lang.NumberFormatException`.

Om vi vid inläsning av `value2` ger heltalet 0 uppstår ett fel vid beräkningen av uttrycket `value1/value2`, eftersom vi då har en division med 0. Ett undantag erhålls av typen `java.lang.ArithmeticException`.

Exempel på exceptionella händelser (2)

Exempel 2:

Betrakta nedanstående programsegment:

```
int[] elements = new int[10];
String input = JOptionPane.showInputDialog("Ange antalet element: ");
int number = Integer.parseInt(input);
for (int i = 0; i < number; i = i + 1) {
   input = JOptionPane.showInputDialog("Ange elementet: ");
   elements[i] = Integer.parseInt(input);
}
```

Om vi vid inläsning av antal ger ett värde större än 10 uppstår ett fel vid inläsningen av komponenten `elements[10]`, eftersom fältets element är indexerat mellan 0 och 9. Ett undantag erhålls av typen

`java.lang.ArrayIndexOutOfBoundsException`.

Exempel på exceptionella händelser (3)

Exempel 3:

Betrakta nedanstående programsegment:

```
String answer = JOptionPane.showInputDialog("Ange svar: ");
if (answer.equals("ja"))
   . . .;
else
   . . .;
```

Om vi vid inläsning av strängen `answer` trycker på Cancel-knappen) kommer `answer` att få värdet **null** och det uppstår ett fel vid anropet av `answer.equals("ja")`. Ett undantag erhålls av typen

`java.lang.NullPointerException`

Exempel på exceptionella händelser (4)

Exempel 4:

Betrakta nedanstående programsegment:

```
String fileName = JOptionPane.showInputDialog("Ange filnamn: ");
BufferedReader inFile =  new BufferedReader (
                         new FileReader(fileName));
```

Avsikten med koden är att läsa in namnet på en fil och koppla en ström till filen för att kunna läsa innehållet på filen.

Om vi vid inläsning ger namnet på en fil som inte finns eller inte kan öppnas för läsning uppstår ett fel vid anropet av konstruktorn `FileReader(fileName)`. Ett undantag erhålls av typen

`java.io.FileNotFoundException`.

Hantering av exceptionella händelser

När det gäller hantering av exceptionella händelser kan man ha olika ambitionsnivåer:

- Ta hand om händelsen och försöka vidta någon lämplig åtgärd i programenheten där felet inträffar så att exekveringen kan fortsätta.
- Fånga upp händelsen, identifiera den och skicka den vidare till anropande programenhet.
- Ignorera händelsen, vilket innebär att programmet avbryts om händelsen inträffar.

Java skiljer mellan två kategorier av exceptionella händelser

- kontrollerade exceptionella händelser (*checked exceptions*)
- okontrollerade exceptionella händelser (*unchecked exceptions*).

Kontrollerade och okontrollerade exceptions.

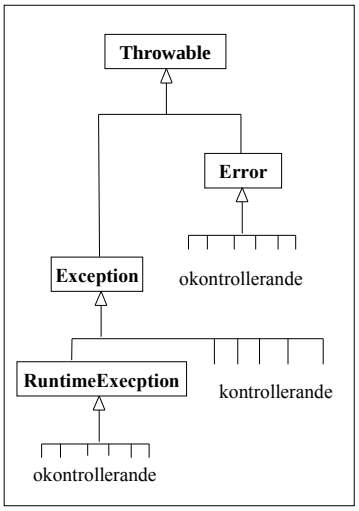
Kontrollerade exceptionella händelser *måste* hanteras i programmet. Om en kontrollerad exceptionell händelse inträffar i en metod och inte fångas måste metoden specificera händelsen genom att ange den i sitt metodhuvud i den s.k. *händelselistan* enligt:

```
public int method() throws CheckedException {
   ...
}
```

Händelselistan kan innehåller flera exceptionella händelser.

De kontrollerade exceptionella händelserna tillhör klassen `Exception` (som också är superklass till den okontrollerade subklassen `RuntimeException`).

Okontrollerade exceptionella händelser behöver inte specificeras i händelselistan. De okontrollerade exceptionella händelserna tillhör klasserna `Error` eller `RuntimeException` eller någon subklass till dessa klasser.



Subklasser till klassen RuntimeException

Exceptionella händelser som härleds ur `RuntimeException` kan sägas vara "programmerarens fel" att de inträffar. Med "bra" kod "borde" de kunna undvikas. Därför är det inget krav på att de fångas. Exempel på fel av detta slag är fel i typomvandling, användning av index som inte finns och försök att använda objekt med värdet **null**.

Bland annat finns följande subklasser till `RuntimeException`:

`ArithmeticException`	`ArrayStoreException`
`CannotRedoException`	`CannotUndoException`
`ClassCastException`	`EmptyStackException`
`IllegalArgumentException`	`IllegalMonitorStateException`
`IllegalPathStateException`	`IllegalStateException`
`ImagingOpException`	`IndexOutOfBoundsException`
`MissingResourceException`	`NegativeArraySizeException`
`NoSuchElementException`	`Null1PointerException`
`ProfileDataException`	`ProviderException`
`SecurityException`	`SystemException`
`UndeclaredThrowableException`	`UnsupportedOperationException`

Kontrollerade exceptionella händelser

Metoden `readLine` i klassen `BufferedReader` är specificerad på följande sätt:

> **public** String readLine() **throws** IOException
> Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
>
> Returns:
> A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
>
> Throws:
> IOException - If an I/O error occurs

`IOException` är en kontrollerad exceptionell händelse, vilket betyder att en metod som anropar `readLine` antingen måste fånga den exceptionella händelsen eller ange den i sin händelselista. Fångas inte den exceptionella händelsen innebär detta att händelsen skickas vidare till den metod som anropade metoden där händelsen inträffade.

Att skicka exceptionella händelser vidare

```
import java.io.*;
public class TestOne {
   public static void main(String[] arg) throws IOException {
      TestOne obj = new TestOne();
      System.out.print("Ange ett heltal: ");
      int value = obj.getInt();
      System.out.println("Talet är: " + value);
   }//main
   public int getInt()  throws IOException {
      BufferedReader myIn = new BufferedReader(
                                        new InputStreamReader(System.in));
      String str = myIn.readLine();
      int i = Integer.parseInt(str);
      return i;
   }//getInt
}//TestOne
```

Måste anges (fångas inte)

Måste anges (fångas inte)

Att skicka exceptionella händelser vidare

Metoden `readLine` i klassen `BufferedReader` kastar ett kontrollerat undantag av typen `IOException`, och metoden `parseInt` i klassen `Integer` kastar ett okontrollerat undantag av typen `NumberFormatException`.

Metod `getInt` anropar `readLine` och `parseInt` utan att fånga dessa undantag. Metoden måste ange `IOException` i sin händelselista.

Metoden `main` anropar `getInt`, som alltså kan kasta två typer av undantag – `IOException` och `NumberFormatException`. Inga undantag fångas i `main`, varför `main` måste ange det kontrollerade undantaget `IOException` i sin händelselista.

Körning av programmet:

```
Ange ett heltal: 12
Talet är: 12

Ange ett heltal: fel

Exception in thread "main" java.lang.NumberFormatException:  For
input string: "fel"
    at java.lang.NumberFormatException.forInputString(
    NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:449)
    at java.lang.Integer.parseInt(Integer.java:499)
    at ExceptionTestOne.getInt(ExceptionTestOne .java:13)
    at ExceptionTestOne.main(ExceptionTestOne .java:7)
```

Att fånga exceptionella händelser (1)

För att fånga exceptionella händelser används **try-catch-finally** konstruktionen.

```
. . .
try {
   . . .
   // anrop av metod(er) som kan kasta undantag
   . . .
}
catch (ExceptionOne e) {
   // hantering av undantag av klassen ExceptionOne
   // eller subklasser till denna klass
}
catch (ExceptionTwo e) {
   // hantering av undantag av klassen ExceptionTwo
   // eller subklasser till denna klass
}
finally {
   // satser som utförs oberoende av om undantag har inträffat eller inte
}
. . .
```

Att fånga exceptionella händelser (2)

När ett undantag inträffar i ett **try**-block avbryts exekveringen och undantaget kastas vidare. Finns det en **catch**-sats som hanterar den typ av undantag som inträffat, fångas undantaget och koden i **catch**-satsen utförs. Därefter fortsätter exekveringen med satserna *efter* **try-catch**-blocket. Man hoppar *aldrig* tillbaks till **try**-blocket.

Man kan ha hur många **catch**-block som helst. Sökningen sker sekventiellt bland **catch**-satserna.

Ett **catch**-block fångar alla undantag som är av specificerad klass eller subklasser till denna klass. Ordningen av **catch**-blocken är således av betydelse.

Finns ett **finally**-block exekveras detta oavsett om ett undantag inträffar eller ej.

Om inget **catch**-block finns för den klass av undantag som inträffat kastas undantaget vidare till den anropande metoden.

Kontrollerade undantag som kastas från en metod måste anges i metodens händelselista.

Anropsstacken

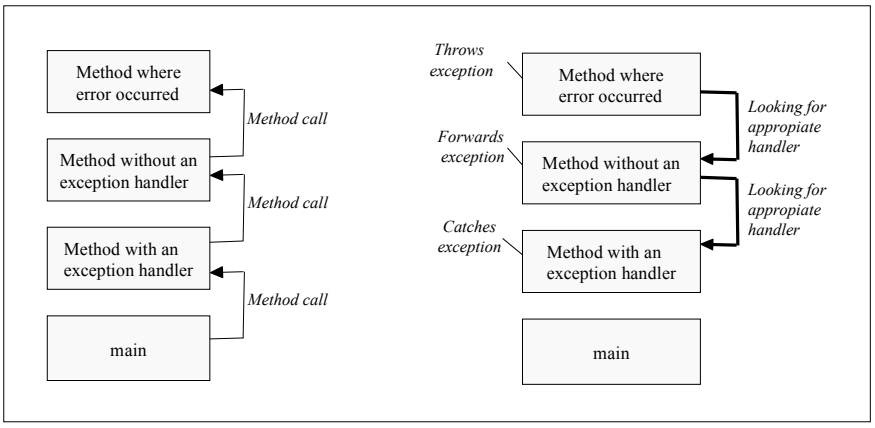
Då ett program körs och en metod anropas används *anropsstacken*. All data som behövs vid anropet, t.ex. parametrar och återhoppsadress, sätts samman till en s.k. *stackframe* och läggs upp på anropsstacken. Då metoden är klar avläses eventuellt resultat varefter stackframen tas bort från stacken. Exekveringen fortsätter vid angiven återhoppsadress.

Antag att metoden a anropar metoden b som anropar metoden c.

- a läggs på stacken, anropar b
- b läggs på stacken, anropar c
- c läggs på stacken
- c klar, c tas bort från stacken, exekveringen fortsätter i b
- b klar, b tas bort från stacken, exekveringen fortsätter i a
- a klar, a tas bort från stacken

Då ett undantag uppstår avbryts det normala flödet och programmet vandrar tillbaks genom anropsstacken. Hanteras inte undantaget någonstans kommer programmet att avbrytas med en felutskrift.

Anropsstacken



Att fånga exceptionella händelser (3)

Nedan har vi skrivit programmet på så sätt att metoden `getInt` fångar undantagen `IOException` och metoden `main` fångar undantaget `NumberFormatException`.

```
import java.io.*;
public class TestTwo {
   public static void main(String[] arg) {
      TestTwo obj = new TestTwo();
      while (true) {
         System.out.print("Ange ett heltal: ");
         try {
            int value = obj.getInt();
            System.out.println("Talet är: " + value);
            break;
         }
         catch (NumberFormatException e) {
            System.out.println("Inget heltal!");
         }
      }
   }//main
```

```
   public int getInt() {
      BufferedReader myIn = new BufferedReader(
                  new InputStreamReader(System.in));
      int i = 0;
      try {
         String str = myIn.readLine();
         i = Integer.parseInt(str);
      }
      catch (IOException e) {
         e.printStackTrace();    // skriv ut "felspåret"
         System.exit(1);  //avbryt exekveringen
      }
      return i;
   }//getInt
}//TestTwo
```

Körning av programmet:

Ange ett heltal: 12
Talet är: 12

Ange ett heltal: fel
Inget helttal!
Ange ett heltal: fel igen
Inget heltal!
Ange ett heltal: 222
Det inlästa talet är: 222

Hur vet programmeraren av `main` att metoden `getInt` kastar ett undantag av typen `NumberFormatException`?

I detta fall endast genom att läsa och förstå koden!

Att kasta egna exceptionella händelser (1)

När man skall kasta ett eget undantag måste man först bestämma om undantaget skall vara kontrollerat eller okontrollerat.

- *Unchecked exceptions*: Används för programmeringsfel, d.v.s. sådant som vi själva är ansvariga för. Det skall smälla så snabbt och så högt som möjligt (undantagen fångas inte). Vi använder typiska objekt från klasserna `NullPointerException`, `IllegalArgumentException`, och `IllegalStateException`.
- *Checked exceptions*: Används för fel som är möjliga att hantera (programmet skall inte terminiera). Till exempel skall inte en databasfråga som resulterar i att man inte hittar det eftersökta göra att programmet avslutas.

Ett undantag kastas med satsen

throw new SomeException("Förklaring till felet");

Man kan kasta undantag av redan existerande klasser eller skapa egna undantagsklasser.

Att kasta egna exceptionella händelser (2)

Egna undantagsklasser erhålls genom att skapa subklasser till klassen `Exception` (kontrollerade undantag) eller till klassen `RuntimeException` (okontrollerade undantag):

```
public class ExceptionName extends Exception {
    public ExceptionName() {
        super();
    }
    public ExceptionName(String str) {
        super(str);
    }
}
```

Konstruktorn `ExceptionName(String str)` används för att beskriva det uppkomna felet. Beskrivningen kan sedan läsas när felet fångas m.h.a. metoden `getMessage()`, som ärvs från superklassen `Throwable`. Om felet inte fångas i programmet kommer den inlagda texten att skrivas ut när programmet avbryts.

Även metoden `printStackTrace()` ärvs från `Throwable`. Denna metod skriver ut var felet inträffade och "spåret" av metoder som sänt felet vidare. Metoden `printStackTrace()` anropas automatiskt när en exceptionell händelse avbryter ett program.

Att kasta egna exceptionella händelser (3)

I vårt tidigare exempel fångade vi inte undantaget `NumberFormatException` i metoden `getInt`. Vi skall nu skriva om metoden på så sätt att den fångar detta undantag och kastar ett eget undantag för att "tala om" att ett fel uppkommit.

Vi beslutar oss för ett kontrollerat undantag och börjar med att definiera en ny klass `ReadIntException` enligt:

```
public class ReadIntException extends Exception {
   public ReadIntException() {
      super();
   }
   public ReadIntException(String s) {
      super(s);
   }
}
```

Ett undantag av klassen `ReadIntException` kastas genom att skriva

```
throw new ReadIntException();
```

eller

```
throw new ReadIntException("En text");
```

Vårt program får följande utseende:

```
import java.io.*;
public class TestThree {
   public static void main(String[] arg) {
      TestThree obj = new TestThree();
      while (true) {
         System.out.print("Ange ett heltal: ");
         try {
            int value = getInt();
            System.out.println("Talet är: " + value);
            break;
         }
         catch (ReadIntException e) {
            System.out.println(e.getMessage());
         }
      }
   }//main
```

```
   public static int getInt() throws ReadIntException {
      BufferedReader myIn =  new BufferedReader(
                        new InputStreamReader(System.in));
       int i = 0;
       try {
          String str = myIn.readLine();
          i = Integer.parseInt(str);
       }
       catch (NumberFormatException e) {
          throw new ReadIntException("Inget heltal!");
       }
       catch (IOException e) {
          e.printStackTrace();
          System.exit(1);
       }
       return i;
   }//getInt
}//TestThree
```

Med användning av Scanner:

```
import java.util.Scanner;
import java.util.InputMismatchException;;
public class TestFour {
   public static void main(String[] arg) {
      TestFour obj = new TestFour();
      while (true) {
         System.out.print("Ange ett heltal: ");
         try {
            int value = getInt();
            System.out.println("Talet är: " + value);
            break;
         }
         catch (ReadIntException e) {
            System.out.println(e.getMessage());
         }
      }
   }//main
```

```
   public static int getInt() throws ReadIntException {
      Scanner myIn =  new Scanner(System.in);
       int i = 0;
       try {
          i = myIn.nextInt();;
       }
       catch (InputMismatchException e) {
          throw new ReadIntException("Inget heltal!");
       }
       return i;
   }//getInt
}//TestFour
```

Körning av programmet:

Ange ett heltal: 12
Talet är: 12

Ange ett heltal: fel
Inget heltal!
Ange ett heltal: fel igen
Inget heltal!
Ange ett heltal: 222
Talet är: 222

Vad har vi vunnit med detta?

Är det lämpligt att, som vi gjort, fånga `IOException` i metoden `getInt`?

finally-blocket

Rätt använt är **finally**-blocket mycket användbart för att frigöra resurser eller återställa objekt till väldefinierade tillstånd då fel inträffar. Det finns ett antal användbara tekniker.

Exempel:

Vid läsning av eller skrivning på en fil skall filen alltid stängas, annars riskerar man att data kan gå förlorat.

public FileReader(String fileName) **throws** FileNotFoundException
kastar FileNotFoundException om filen inte existerar eller inte kan öppnas för läsning

public String readLine() **throws** IOException
kastar IOException om läsningen misslyckas

public void close() **throws** IOException
kastar IOException om stängningen misslyckas

finally-blocket

Metoden skickar alla undantag vidare.

```
import java.io.*;
public final class SimpleFinally {
   public static void main(String[] args) throws IOException {
      simpleFinally("test.txt");
   }//main
   private static void simpleFinally(String aFileName) throws IOException {
      //If this line throws an exception, then neither the
      //try block nor the finally block will execute.
      //That is a good thing, since reader would be null.
     BufferedReader reader = new BufferedReader(new FileReader(aFileName));
      try {
         //Any exception in the try block will cause the finally block to execute
         String line = null;
         while ( (line = reader.readLine()) != null ) {
            //process the line...
         }
      }
         finally {
            //The reader object will never be null here.
            //This finally is only entered after the try block is entered.
            reader.close();
         }
      }//simpleFinally
   }//SimpleFinally
```

finally-blocket

Metoden fångar alla undantag.
try..**finally** nästlad med **try**. .**catch**

```
import java.io.*;
public final class NestedFinally {
   public static void main(String[] args) {
      nestedFinally("test.txt");
   }//main
   private static void nestedFinally(String aFileName) {
      try {
         //If the constructor throws an exception, the finally block will NOT execute
         BufferedReader reader = new BufferedReader(new FileReader(aFileName));
         try {
            String line = null;
            while ( (line = reader.readLine()) != null ) {
               //process the line...
            }
         }
               finally {
                  //no need to check for null, any exceptions thrown here
                  //will be caught by the outer catch block
                  reader.close();
               }
            }
            catch (IOException ex) {
               ex.printStackTrace();
            }
      }//nestedFinally
   }//NestedFinally
```

finally-blocket

Metoden fångar alla undantag.
try..**catch** inuti **finally**

```
import java.io.*;
public final class CatchInsideFinally {
   public static void main(String[] args) {
      catchInsideFinally("test.txt");
   }//main
   private static void catchInsideFinally(String aFileName) {
      //Declared here to be visible from finally block
      BufferedReader reader = null;
      try {
         //if this line fails, finally will be executed,
         //and reader will be null
         reader = new BufferedReader(
                        new FileReader(aFileName));
         String line = null;
         while ((line = reader.readLine()) != null ) {
            //process the line...
         }
      }
         catch(IOException ex){
            ex.printStackTrace();
         }
         finally {
            try {
               //need to check for null
               if ( reader != null ) {
                  reader.close();
               }
            }
            catch(IOException ex) {
               ex.printStackTrace();
            }
         }
      }//catchInsideFinally
   }//CatchInsideFinally
```

Case study: Sortering

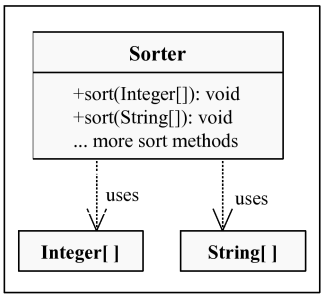
Problem: Skapa en klass `Sorter` som kan sortera alla typer av fält.

En första design:

```
public class Sorter {
   public static void sort(String[] data) {
      // code for sorting String arrays
   }

   public static void sort(Integer[] data) {
      // code for sorting Integer arrays
   }

   // methods for sorting other arrays
}
```

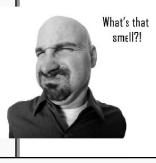


Problem med den första versionen

- För varje specifik typ av array måste det finnas en specifik metod i klassen `Sorter`.
  - Vad händer som vi i framtiden inför nya typer?
- Mycket kod är duplicerad i metoderna.
  - Kan vi undvika detta?
- I bland vill man sortera på något annat sätt än i växande ordning.
  - Hur kan vi åstadkomma detta?

Duplicerad kod

```
public class Sorter {
   public static void sort(Integer[] data) {
      for (int i = data.length-1; i >= 1;  i--) {
         int indexOfMax = 0;
         for (int j = 1; j <= i; j++) {
            if (data[j] > data[indexOfMax])
               indexOfMax = j;
         }
         Integer temp = data[i];
         data[i] = data[indexOfMax];
         data[indexOfMax] = temp;
      }
   }//sort
```




```
   public static void sort(String[] data){
      for (int i = data.length-1; i >= 1;  i--) {
         int indexOfMax = 0;
         for (int j = 1; j <= i; j++) {
            if (data[j].compareTo(data[indexOfMax]) > 0)
               indexOfMax = j;
         }
         String temp = data[i];
         data[i] = data[indexOfMax];
         data[indexOfMax] = temp;
      }
   }//sort
   // methods for sorting other arrays
}//Sorter
```

Skillnader i koden

Vi notera att metoderna är identiska förutom att

- fälten i parameterlistorna har olika typer
- de använder olika sätt att jämföra två värden;
  - operationen > för `Integer`
  - metoden `compareTo` för `String`
- de använder olika typer på den lokala variabeln `temp`.

För att undvika duplicering av kod vill vi skapa en metod som är generisk (allmängiltigt) för alla typer av fält.

Steg 1: Inför en ny klass `Comparator`

För att eliminera skillnaden i hur de båda metoderna jämför objekt inför vi en ny klass `Comparator`, vars uppgift är att utföra jämförelserna åt oss.

Klassen innehåller överlagrade `compare`-metoder för de typer av objekt vi vill hantera.

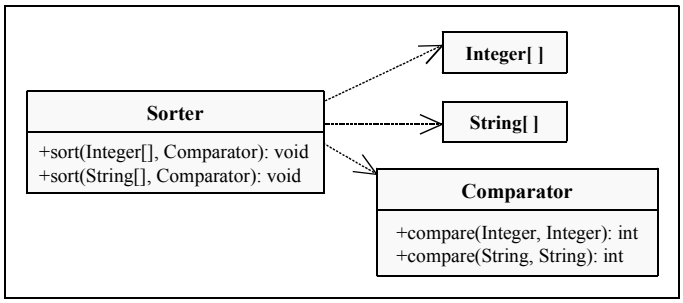
```
public class Comparator {
   public int compare(String o1, String o2) {
         return  o1.compareTo(o2);
   }//compare
   public int compare(Integer o1, Integer o2) {
         return o1 - o2;
   }//compare
    //more compare methods for other types
}//Comparator
```

Steg 2: Använd klassen `Comparator`

```
public class Sorter {
   public static void sort(Integer[] data, Comparator comp) {
      for (int i = data.length-1; i >= 1;  i--) {
         int indexOfMax = 0;
         for (int j = 1; j <= i; j++) {
            if (comp.compare(data[j], data[indexOfMax]) > 0)
               indexOfMax = j;
         }
         Integer temp = data[i];
         data[i] = data[indexOfMax];
         data[indexOfMax] = temp;
      }
   }//sorter
```

```
   public static void sort(String[] data, Comparator comp) {
      for (int i = data.length-1; i >= 1;  i--) {
         int indexOfMax = 0;
         for (int j = 1; j <= i; j++) {
            if (comp.compare(data[j], data[indexOfMax]) > 0)
               indexOfMax = j;
         }
         String temp = data[i];
         data[i] = data[indexOfMax];
         data[indexOfMax] = temp;
      }
   }//sorter
   // methods for sorting other arrays
}//Sorter
```

Steg 2: Använd klassen `Comparator`



Steg 3: Utvidga typtillhörigheten - polymorfism

Det gäller att:

- typerna `Integer` och `String` är subtyper till typen `Object`
- typerna Integer[] och `String[ ]` är subtyper till `Object[ ]`.

Vi kan således utnyttja denna möjlighet till polymorfism för att eliminera den återstående skillnaden i de båda metoderna i klassen `Sorter`.

Vi ersätter typerna `String` och `String[ ]` respektive `Integer` och `Integer[ ]`, med typerna `Objekt` och `Objekt[ ]`.

De båda metoderna blir därmed identiska!

Slutlig version av klassen `Sorter`

```
public class Sorter {
   public static void sort(Object[] data, Comparator comp) {
      for (int i = data.length-1; i >= 1;  i--) {
         int indexOfMax = 0;
         for (int j = 1; j <= i; j++) {
            if (comp.compare(data[j], data[indexOfMax]) > 0)
               indexOfMax = j;
         }
         Object temp = data[i];
         data[i] = data[indexOfMax];
         data[indexOfMax] = temp;
      }
   }//sort
}//Sorter
```

Problem:

Eftersom den statiska typen på de objekt som hanteras i metoden `sort` nu är `Object` inträffar ett kompileringsfel vid anropet av

comp.compare(data[j], data[indexOfMax])

Det finns ingen metod `compare` i klassen `Comparator` med signaturen

compare(Object, Object)

Vi måste införa en sådan metod i `Comparator`.

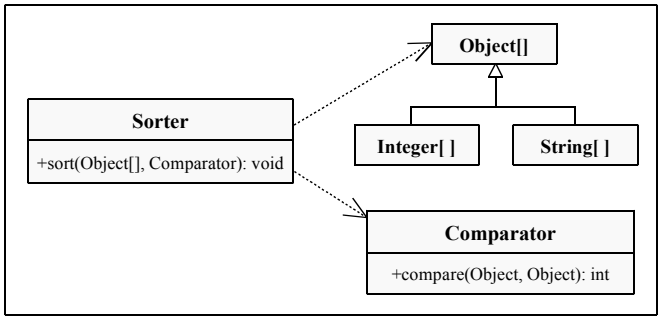
Denna metod blir dock den metod som alltid kommer att väljas!

Därför måste denna nya metod handha alla typer av objekt.

Ny version av klassen `Comparator`

```
public class  Comparator {
   public int compare(Object o1, Object o2) {
      if (o1 instanceof String && o2 instanceof String )
         return ((String) o1).compareTo((String) o2);
      else if (o1 instanceof Integer && o2 instanceof
Integer )
         return (Integer) o1 - (Integer) o2;
      else
         throw new IllegalArgumentException
                        ("Can't handle this type of objects.");
   }//compare
}//Comparator
```

Den nya designen



Brister med klassen `Comparator`

Metoden

compare(Object, Object)

uppvisar flera brister:

- vill vi hantera andra typer av fält än `Integer[]` och `String[]` måste metoden skrivas om
- metoden klarar inte av att sortera ett fält på olika sätt (t.ex. i stigande eller avtagande ordning).

Problemet är att metoden försöker göra för mycket, nämligen att handha jämförelsen av all typer av objekt.

Lösningen är att göra `Comparator` till ett interface och implementera en `Comparator`-klass för varje typ av objekt och sorteringsordning.

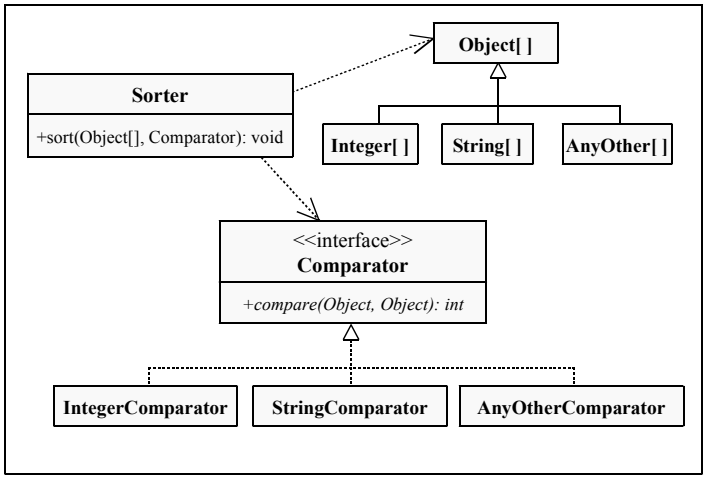
Interfacet `Comparator` och konkreta klasser

```
public interface Comparator {
   public int compare(Object o1, Object o2);
}//Comparator
```

```
public class StringComparator implements Comparator {
   @Override
   public int compare(Object o1, Object o2) {
      String s1 = (String) o1;
      String s2 = (String) o2;
      return s1.compareTo(s2);
   }
}//StringComparator
```

```
public class IntegerComparator implements Comparator {
   @Override
   public int compare(Object o1, Object o2) {
      return (Integer) o1 - (Integer) o2;
   }
}//IntegerComparator
```

Slutlig design

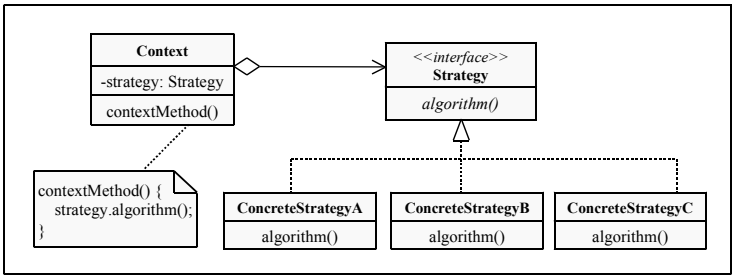


Designmönstret `Strategy`

Vår slutliga lösning utnyttjar designmönstret Strategy.

Designmönstret Strategy är en generell teknik som används när man vill att ett objekt skall ha olika beteenden, d.v.s. utföra olika algoritmer, beroende på vilken klient som använder objektet. Detta åstadkoms genom att objektet och dess beteende separeras och läggs i olika klasser.

När man har flera likartade objekt som endast skiljer sig åt i sitt beteende, är det en bra idé att använda sig av Strategy-mönstret.



Jämförelse av objekt

För att finna ett specifikt objekt i en samling, eller för att sortera en samling av objekt, är det nödvändigt att kunna jämföra två objekt på storlek.

Exempelvis tillhandahåller klasserna `Arrays` och `Collections` statiska metoder för att sortera ett fält respektive en lista.

Hur jämför vi objekt av klassen `Person`?

```
public class Person {
    private String name;
    private int id;
    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }//constructor

    public int getId() {
        return id;
    }//getId

    @Override
    public String toString() {
        return ("Namn: " + name + "    IDnr: " + id);
    }//toString
}//Person
```

Gränssnitten `Comparable<T>`

Om en klass implementerar gränssnittet `Comparable<T>` är objekten i klassen jämförbara på storlek.

Gränssnittet `Comparable` innehåller endast en metod:

```
public interface Comparable <T> {
    public int compareTo(T o);
}
```

Ett anrop

```
a.compareTo(b)
```

skall returnera värdet 0 om a har samma storlek som b, returnera ett negativt tal om a är mindre än b och returnera ett positivt tal om a är större än b.

Gränssnitten `Comparable<T>`

```
public class Person implements Comparable <Person>{
   private String name;
   private int id;
   public Person(String name, int id) {
      this.name = name;
      this.id = id;
   }//constructor
   public int getId() {
      return id;
   }//getId
      @Override
      public String toString() {
         return ("Namn: " + name + "   IDnr: " + id);
      }//toString
      @Override
      public int compareTo(Person otherPerson) {
         return name.compareTo(otherPerson.name);
      }//compareTo
}//Person
```

Total ordning

När man implementerar `compareTo`, måste man förvissa sig om att metoden definierar en *total ordning*, d.v.s. uppfyller följande tre egenskaper:

Antisymmetri: Om `a.compareTo(b) <= 0`, så `b.compareTo(a) >= 0`

Reflexiv: `a.compareTo(a) = 0`

Transitiv: Om `a.compareTo(b) <= 0` och `b.compareTo(c) <=0`, så `a.compareTo(c) <= 0`

När en klass implementerar `Comparable` skall man förvissa sig om att villkoret

x.equals(y) ⇒ (x.compareTo(y) == 0)

gäller, samt eftersträva att villkoret

x.equals(y) ⇐ (x.compareTo(y) == 0)

gäller.

Gränssnitten `Comparator<T>` i Java

Gränsnittet `Comparator<T>` som vi införde tidigare finns att tillgå i Javas API:

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
    public boolean equals(Object obj);
}
```

Ett anrop

```
comp.compare(a, b)
```

skall returnera värdet 0 om `a` är lika med `b`, returnera ett negativt tal om `a` är mindre än `b` och returnera ett positivt tal om `a` är större än `b`.

Metoden `equals` används för att undersöka om två objekt av `Comparator` är lika och behöver normalt inte överskuggas.

Exempel: `Comparator<T>`

Antag att vi vill jämföra objekt av klassen `Person` med avseende på växande ordning på komponenten `id`.

Vi måste då införa en ny klass som vi kallar `IDComparator`, vilken implementerar gränssnittet `Comparator`:

```
import java.util.*;
public class IDComparator implements Comparator <Person> {
    @Override
    public int compare(Person a, Person b) {
        if (a.getId() < b.getId())
            return -1;
        if (a.getId() == b.getId())
            return 0;
        return 1;
    }//compare
}//IDComparator
```

Exempel: `Comparator<T>`

Att sortera fält av godtyckliga typer, som vi gjorde vårt tidigare exempel, finns också att tillgå i Javas API i klassen Arrays.

Att sortera ett fält med objekt av typen Person i växande ordning på id-nummer sker på följande vis:

```
int size = . . .;
Person[] persons = newPerson[size];
// add persons
Arrays.sort(persons, new IDComparator());
```

Fortsättning på exempel:

Nedan ges ett program som lagrar ett antal objekt av klassen `Person` i ett fält och objekten skrivs ut dels i bokstavsordning med avseende på namnen och dels i växande ordning med avseende på id-numret.

```
import java.util.*;
public class PersonTest {
   public static void main(String[] arg) {
      Person[] persons = {new Person("Kalle", 22), new Person("Bertil", 62),
                          new Person("Sture", 42), new Person("Rune", 12),
                          new Person("Allan", 52)};
      Arrays.sort(persons);
      System.out.println("I bokstavordning:");
      for (int i = 0; i < persons.length; i++ )
         System.out.println(persons[i]);
      Arrays.sort(persons, new IDComparator());
      System.out.println("I id-ordning:");
      for (int i = 0; i < persons.length; i++ )
         System.out.println(persons[i]);
   }//main
}//PersonTest
```

Utskriften av programmet blir:

```
I bokstavordning:
Namn: Allan   IDnr: 52
Namn: Bertil   IDnr: 62
Namn: Kalle   IDnr: 22
Namn: Rune   IDnr: 12
Namn: Sture   IDnr: 42

I id-ordning:
Namn: Rune   IDnr: 12
Namn: Kalle   IDnr: 22
Namn: Sture   IDnr: 42
Namn: Allan   IDnr: 52
Namn: Bertil   IDnr: 62
```