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.

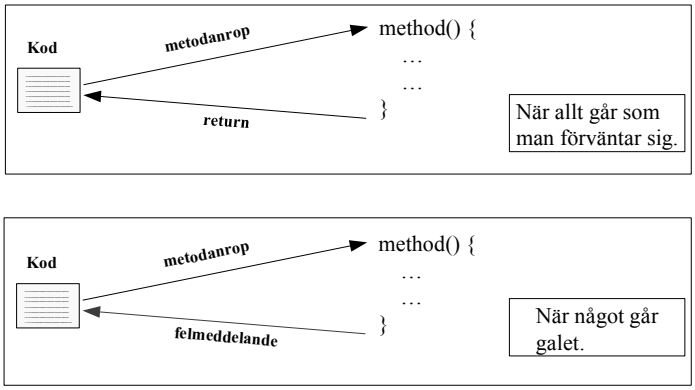
Alla utvecklingsmiljöer (JGrasp, Eclipse, NetBeans, etc) tillhandahåller verktyg för debugging, lär dej att använda dessa i den utvecklingsmiljö du använder.

Orsaker till exekveringsfel

Orsaker till att exekveringsfel uppstår:

- Oförutsägbara externa problem. Nätverket gick ner, hårddisken är full, minnet är slut, databasen har kraschat, DVD:n var inte insatt, osv.
- Felaktig användning. Användaren har inte läst dokumentationen, dvs följer inte uppställda förvillkor.
- Brister i koden. Programmeraren har inte gjort sitt arbete. Koden innehåller buggar (t.ex. refererar till ett objekt som inte finns, adresserar utanför giltiga index i ett fält och representations exponering).

Exekveringsfel i ett program

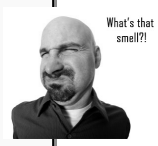


För att hantera felet måste den anropande koden kunna tolka felmeddelandet och vidta lämpliga åtgärder.

Fel i ett program

Har programspråket inga inbyggda mekanismer för felhantering, måste felhanteringen ske via konventioner.

```
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.

Hantering av exekveringsfel i Java

Java använder *exceptions* (undantag) för att hantera exekveringsfel.

Ett exception är ett objekt som påtalar en avvikande eller felaktig situation i ett program.

Java tillhandahåller följande konstruktioner för exceptions:

- konstruktioner för att "kasta" exceptions (konstruktionen **throw**)
- konstruktioner för att "specificera" att en metod kastar eller vidarebefordrar exceptions (konstruktionen **throws**)
- konstruktioner för att fånga exceptions (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.

Kasta exceptions

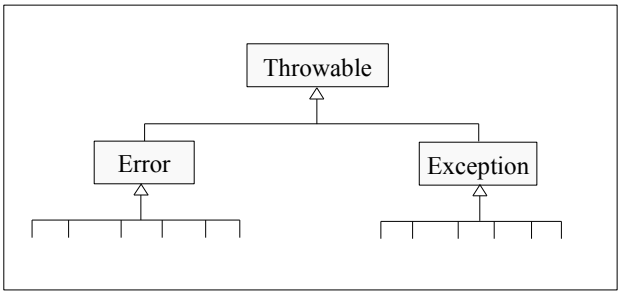
```
public class  PurchaseOrder {
   private Product prod;
   private int quantity;
   . . .
   public double totalCost() throws IllegalQuantityException {
      if (quantity < 0)
         throw new IllegalQuantityException("quantity is < 0");
      else
         return prod.getPrice() * quantity;
   }//totalCost
}//PurchaseOrder
```

Deklarerar att ett exception kastas

Ett exception kastas

Klasshierarkin för exceptions

En exception är ett objekt som tillhör någon subklass till standardklassen Throwable.
Det finns två standardiserade subklasser till Throwable: klasserna Exception och Error.



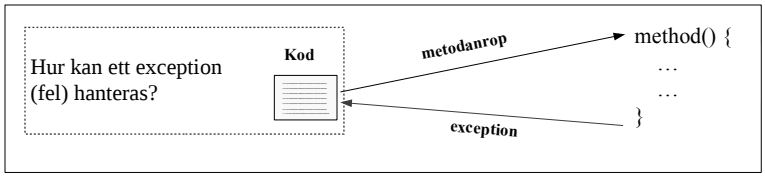
Klasserna Error och Exception

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.

Hantering av exceptions



Man kan ha olika ambitionsnivåer:

- Ta hand om det och försöka vidta någon lämplig åtgärd så att exekveringen kan fortsätta.
- Fånga det, identifiera det och kasta det vidare till anropande programenhet.
- Ignorera det, vilket innebär att programmet avbryts om felet inträffar.

Java skiljer mellan två kategorier av exceptions

- *checked exceptions*
- *unchecked exceptions*

Checked och unchecked exceptions

Checked exceptions *måste* hanteras i programmet:

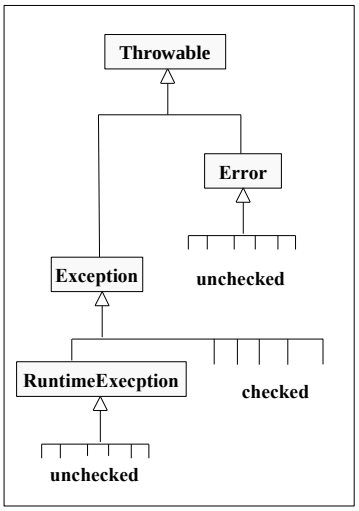
- antingen fånga det eller
- ange i den s.k. *händelselistan* i metodhuvudet att det kastas vidare

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

Händelselistan kan innehåller flera exceptions.

Checked exceptions tillhör klassen Exception eller någon subklass till denna klass (förutom subklassen RuntimeException).

Unchecked exceptions behöver inte specificeras i händelselistan. Unchecked exceptions tillhör klasserna Error eller RuntimeException eller någon subklass till dessa klasser.



Anrop av metoder som kastar exceptions

Från API-dokumentationen kan vi se att metoden parseInt, i klassen java.lang.Integer, kastar en NumberFormatException om parametern s som ges till metoden inte kan översättas till ett heltal.

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Anrop av metoder som kastar exceptions

Från API-dokumentationen kan vi se att konstruktorn Scanner(File source), kastar en FileNotFoundException om filen source som ges som parameter inte finns.

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

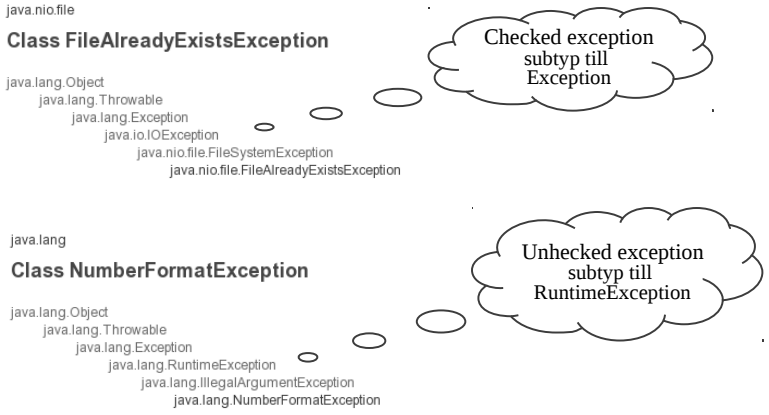
`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

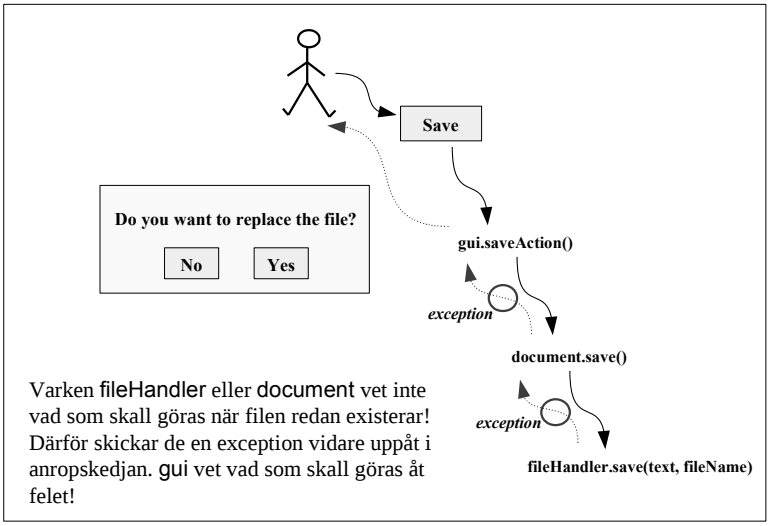
Feltyper i Java

I dokumentationen av en exception, kan man se om den är checked eller unchecked.



Att kasta exceptions vidare

Fel kan oftast inte hanteras där de inträffar!



Varken fileHandler eller document vet inte vad som skall göras när filen redan existerar! Därför skickar de en exception vidare uppåt i anropskedjan. gui vet vad som skall göras åt felet!

Kasta exceptions

```
public class Document {
   private FileHandler fileHandler;
   private String theDocument;
   . . . .
   public void save(String fileName) throws FileAlreadyExistException {
      . . .
      filehandler.save(theDocument, fileName);
      . . .
   }//save
}//Document
```

Kan inte åtgärda felet!

```
public class FileHandler {
   . . . .
   public void save(String text, String fileName) throws FileAlreadyExistException {
      File f = new File(fileName);
      if (f.exists())
         throw new FileAlreadyExistException(fileNamne + " exists.");
      . . .
   }//save
}//Document
```

Detekterar felet, men kan inte åtgärda det!

Fånga exceptions

"An exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer."

"During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception." // JLS 11

Om ingen exception handler finns i terminerar programmet!

Att fånga exceptions

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

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

Att fånga exceptions

När ett exception inträffar i ett **try**-block avbryts exekveringen och exception kastas vidare. Finns det ett **catch**-block som hanterar den typ av exception som inträffat, fångas exception och koden i **catch**-blocket utförs. Därefter fortsätter exekveringen med satserna *efter* **catch**-blocken. 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 exception 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 exception inträffar eller ej.

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

Checked exceptions 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

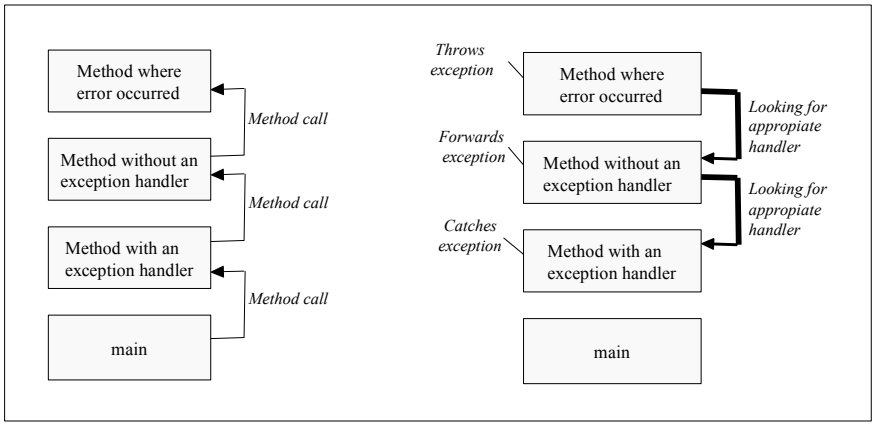
c klar, c tas bort från stacken, exekveringen fortsätter i b

b klar, b tas bort från stacken, exekveringen fortsätter i a

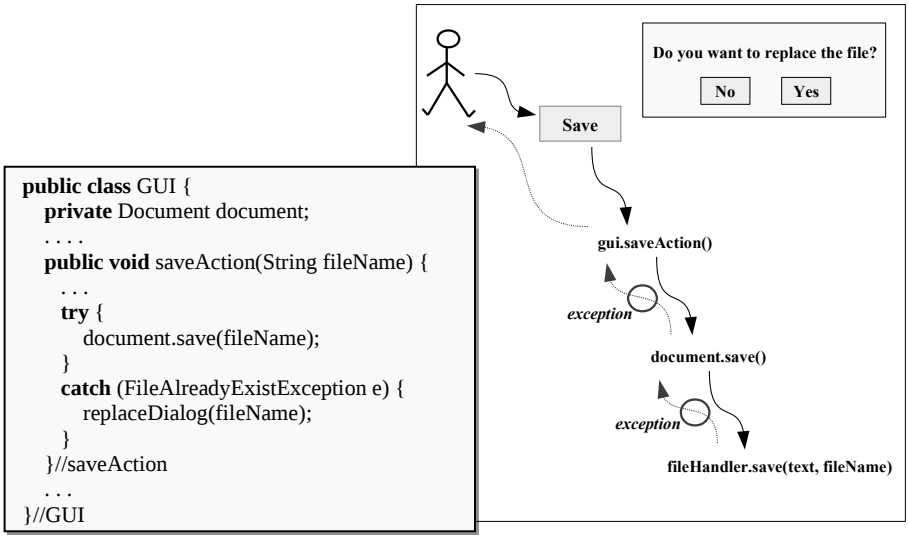
a klar, a tas bort från stacken

Anropsstacken

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.

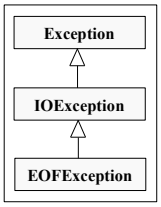


Att fånga exceptions




```
public class GUI {
   private Document document;
   . . . .
   public void saveAction(String fileName) {
      . . .
      try {
         document.save(fileName);
      }
      catch (FileAlreadyExistException e) {
         replaceDialog(fileName);
      }
   }//saveAction
   . . .
}//GUI
```

Flera catch-block




```
public void m1() throws IOException { . . .}
public void m2() throws EOFException { . . .}
```

```
public void p() {
   . . .
   try {
      m1();
      m2();
   } catch (EOFException e) {
      //kör om metoden p kastar  EOFException
   } catch (IOException e) {
      //kör om metoden p kastar  IOException
   }
}//p
```

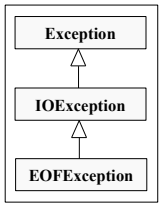
OK

```
public void q() {
   . . .
   try {
      m1();
      m2();
   } catch (IOException e) {
      . . .
   } catch (EOFException e) {
      . . .
   }
}//q
```

WRONG!

Kompileringsfel!
Det andra catch-blocket
är oåtkomligt!

Vilka exceptions kan en metod kasta?




```
public void m1() throws Exception {
   . . .
   throw new IOException();
      . . .
   }
}
```

OK

```
public void m2() throws EOFException {
   . . .
   throw new IOException();
      . . .
   }
}
```

WRONG!

Det är tillåtet att kasta en subtyp till den exceptiontyp som metoden deklarerar.

Att kasta egna exceptions

Det är möjligt att skapa och kasta egendefinierade exceptions. Denna möjlighet skall dock användas restriktivt – återanvändning av existerande exceptiontyper från API:n oftast bättre.

När man skall kasta ett eget exception måste man först bestämma om det skall vara checked eller unchecked.

- *Unchecked exceptions*: Används för programmeringsfel, d.v.s. sådant som vi själva är ansvariga för. Ju snabbare och ju högre det smäller desto bättre (undantagen fångas inte).
- *Checked exceptions*: Används för fel som inträffar vid interaktionen med programmet och 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.

Att kasta egna exceptions

Egna exceptionsklasser erhålls genom att skapa subklasser till klassen Exception (checked) eller till klassen RuntimeException (unchecked):

```
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 exception avbryter ett program.

Kasta exceptions med rätt abstraktionsnivå

Det är förvillande om en metod kastar ett exception som inte har någon uppenbar koppling till den uppgift som metoden utför.

```
try {
   . . .
} catch (LowLevelException e) {
     throw new HighLevelException(s);
}
```

```
//returns: x such that ax^2 + bx + c = 0
//throws: NoRealSolutionsException if no real solutions exist
public double solveQuad(double a, double b, double c)
                                                    throws NoRealSolutionsException {
   try {
      return (-b + Math.sqrt(b *b  - 4 * a * c) / (2 * a);
   } catch (IllegalArgumentException e) {
       throw new NotRealSolutionException("No real solutions exists");
   }
}//solveQuad
```

Exception Swallowing

```
// BAD! BAD! BAD!
try{
      // Possible exception.
} catch (SomeException e) {
      // Empty, nothing here, but exception handled
      // program will continue.
      // So exception unnoticed from now on.
}
```

Detta är det sämsta tänkbara sättet att hantera exceptions!

Endast acceptabelt om det går att bevisa att SomeException aldrig kan inträffa.

Dokumentation av exceptions

Både checked och unchecked exceptions som kastas av en metod bör dokumenteras med javadoc:

```
/**
* . . .
* @throws IllegalArgumentException if parameter args is null
* @throws NoSuchFileException if referencing a non-existing file
*/
public void makeSomething(String args) throws IllegalArgumentException,
                                               NoSuchFileException {
   . . . .
   . . . .
}//makeSomething
```

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
```

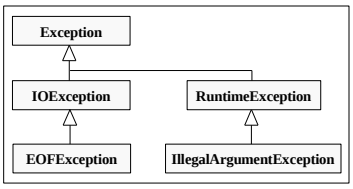
finally-blocket

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
```

Exceptions och överskuggning

Vid överskuggning får metoden i subklassen inte kasta en exception med vidare typtillhörighet.




```
public class Sup {
   public void m1() throws Exception {
   . . .
   }
   public void m2() throws RumtimeException {
   . . .
   }
}
```

```
public class Sub extends Sup {
   @Override
   public void m1() throws IllegalArgumentException {
      . . .
   }

   @Override
   public void m2() throws Exception {
      . . .
   }
}
```

OK

WRONG!

Exceptions: Sammanfattning

- Finns ingen allmänt accepterad *best practices*.
- Använd unchecked exceptions för programmeringsfel, men överväg noga om det borde vara förvillkor och assertions istället.
- Använd checked exception då det kan antas att klienten verkligen kan åtgärda det uppkomna felet.
- Throw early, catch late. Kasta ett exception direkt när felet upptäcks, åtgärda felet där det kan åtgärdas.
- Kasta exceptions som är på rätt abstraktionsnivå.
- Använd företrädesvis fördefinierade exceptions.
- Dokumentera alla exceptions (både checked och unchecked) som en metod kastar.
- Inkludera information som beskriver felorsaken när ett exception kastas.
- Använd inte tomma catch-block (swallowing exceptions).

Case study: Sortering

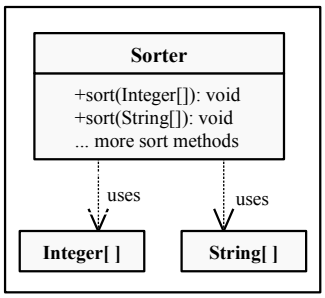
Problem: Skapa en klass `Sorter` som kan sortera alla typer av fält.

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.

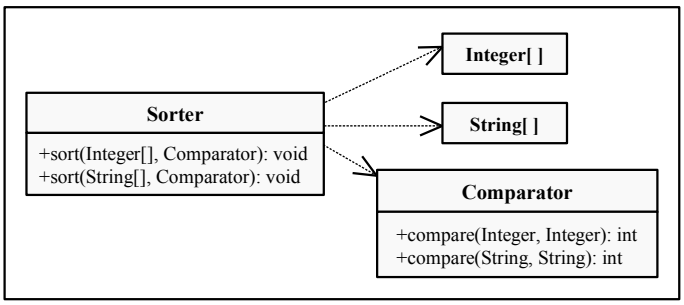
```
public class Comparator {
    public int compare(String o1, String o2) {
          return  o1.compareTo(o2);
    }//compare
    public int compare(Integer o1, Integer o2) {
          return o1 - o2;
    }//compare
     //more compare methods for other types
}//Comparator
```

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`.

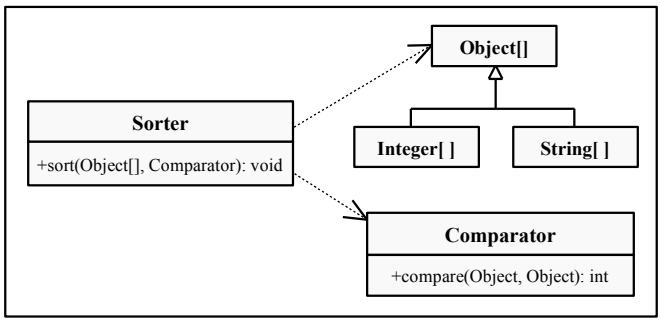
Denna metod blir dock den metod som alltid kommer att väljas!

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.

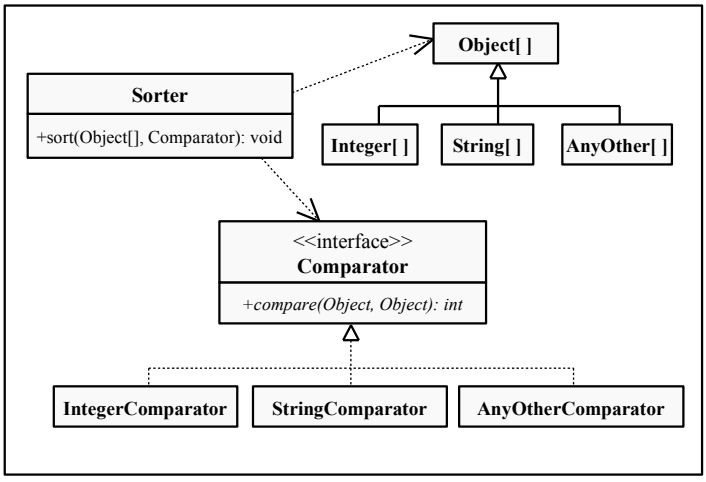
Interfacet `Comparator` och konkreta klasser

```
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



Testprogram

```
public class Main {
   public static void main(String[] args) {
      String[] names = {"Sture", "Malin", "Maja", "Adam","Hilda"};
      Integer[] values = {12, 16, 134, 2, 7};
      Sorter.sort(names, new StringComparator());
      Sorter.sort(values, new IntegerComparator());
      for (int i = 0; i < names.length; i++)
         System.out.println(names[i]);
      System.out.println();
      for (int i = 0; i < values.length; i++)
         System.out.println(values[i]);
   }
}
```

Ger utskriften:

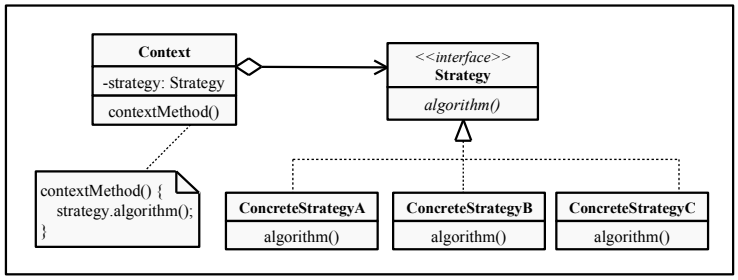
```
Adam Hilda Maja Malin Sture
2 7 12 16 134
```

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
```