Föreläsning 11

Undantag
Testning

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.

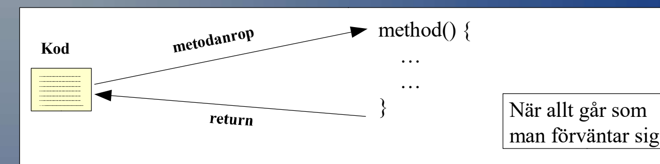
Alla utvecklingsmiljöer (JGrasp, Eclipse, NetBeans, etc) tillhandahåller verktyg för debugging, lär dej att använda dessa i den utvecklingsmiljö du använder.

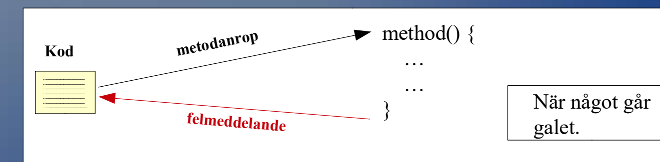
Orsaker till exekveringsfel

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;
   }
}
```

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

Java tillhandahåller följande konstruktioner för undantag:

- konstruktioner för att "kasta" undantag (konstruktionen **throw**)
- konstruktioner för att "specificera" att en metod kastar eller vidarebefordrar undantag (konstruktionen **throws**)
- konstruktioner för att fånga undantag (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;
   }
}
```

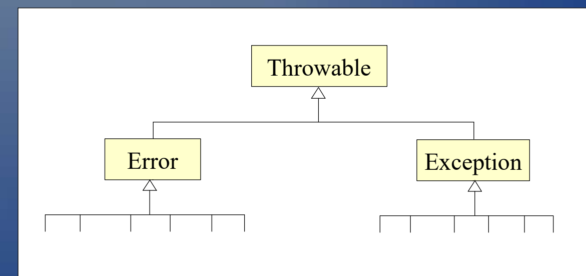
Deklarerar att ett undantag kastas

Ett undantag kastas

Klasshierarkin för exceptions

En undantag är ett objekt som tillhör någon subklass till standardklassen Throwable.

Det finns två standardiserade subklasser till Throwable: klasserna Exception och Error.



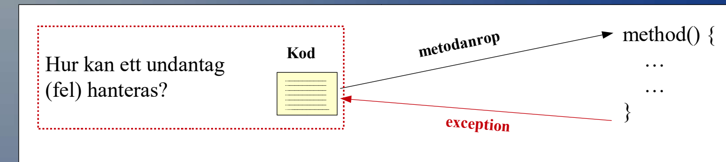
Klasserna Error och Exception

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 undantag



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 undantag

- *kontrollerade undantag*
- *okontrollerade undantag*

Kontrollerade och okontrollerade undantag

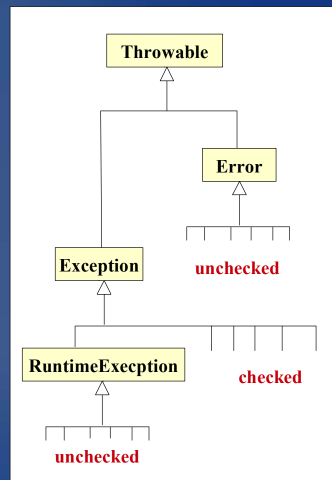
Kontrollerade undantag *måste* hanteras i programmet:

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

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

Kontrollerade undantag tillhör klassen Exception eller någon subklass till denna klass (förutom subklassen RuntimeException).

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



Anrop av metoder som kastar undantag

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 undantag

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 undantagsklass, kan man se om den är kontrollerad eller okontrollerad.

java.nio.file

Class FileAlreadyExistsException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.nio.file.FileSystemException
                    java.nio.file.FileAlreadyExistsException
```

Kontrollerat undantag subtyp till Exception

java.lang

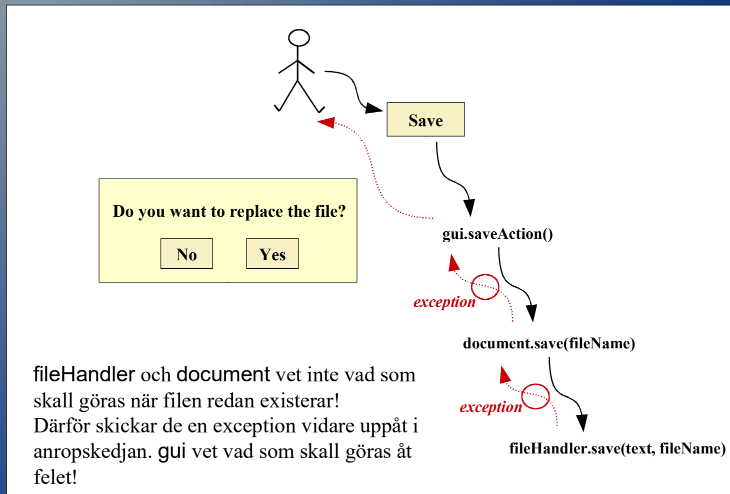
Class NumberFormatException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

Okontrollerat undantag subtyp till RuntimeException

Att kasta undantag vidare

Fel kan oftast inte hanteras där de inträffar!



Kasta undantag

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

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.");
     . . .
   }
}
```

Detekterar felet, men kan inte åtgärda det!

Fånga undantag

"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 undantagshanterare finns terminerar programmet!

Att fånga undantag

För att fånga undantag används **try-catch-finally** konstruktionen.

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


Att fånga undantag

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

Ett godtyckligt antal **catch**-block får finnas. Ett **catch**-block fångar alla undantag som är av specificerad klass eller subklasser till denna. **catch**-blocken genomsöks sekventiellt. 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. *aktiveringspost* (stackframe) och läggs upp på anropsstacken. Då metoden är klar avläses eventuellt resultat varefter aktiveringsposten 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

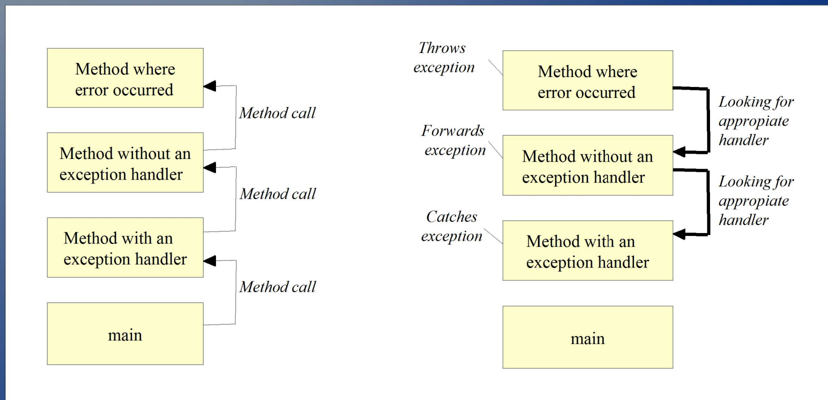
c klar, c tas bort från stacken, exekveringen fortsätter i b

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

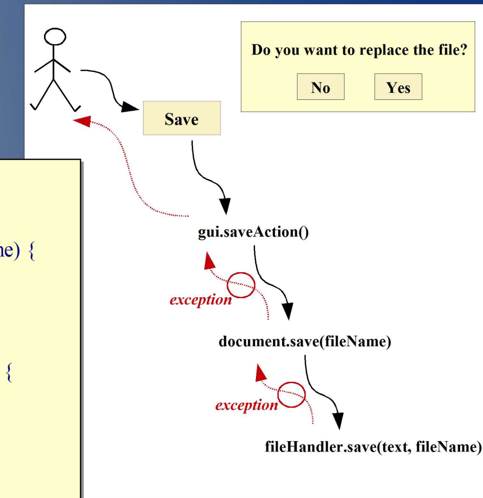
a klar, a tas bort från stacken

Anropsstacken

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

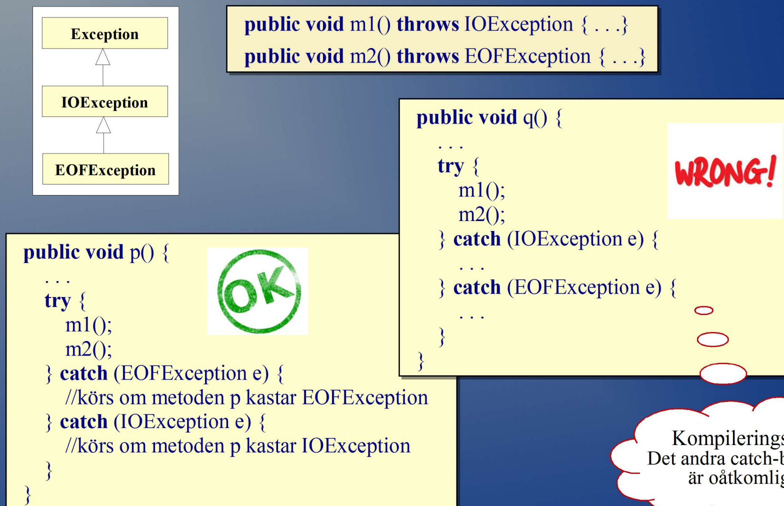


Att fånga undantag




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


Flera catch-block




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

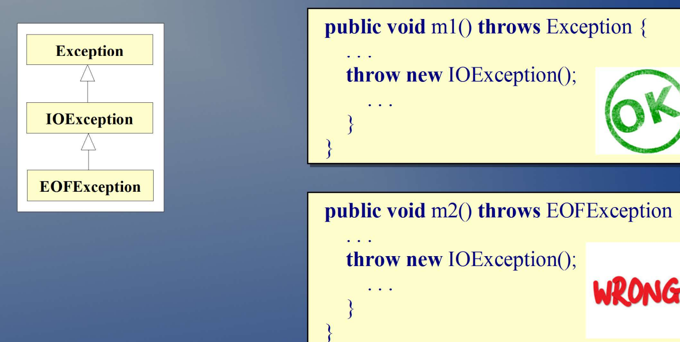
```
public void p() {
   . . .
   try {
      m1();
      m2();
   } catch (EOFException e) {
      //körs om metoden p kastar EOFException
   } catch (IOException e) {
      //körs om metoden p kastar IOException
   }
}
```

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

WRONG!

Kompileringsfel!
Det andra catch-blocket
är oåtkomligt!

Vilka undantag kan en metod kasta?




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

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

Det är tillåtet att kasta ett undantag av subtyp till den undantagstyp som metoden deklarerar.

Att skapa egna undantagsklasser

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

När man skall skapa en egen undantagsklass måste man först bestämma om den skall vara kontrollerad eller okontrollerad.

- *Okontrollerade undantag*: Används för programmeringsfel, d.v.s. fel som programmerarna själva är ansvariga för. Dessa undantag skall inte fångas. Ju snabbare de inträffar desto bättre (*fail fast*).
- *Kontrollerade undantag*: Används för fel som inträffar vid interaktionen med programmet och som är möjliga att hantera (programmet skall inte terminera). Till exempel skall inte en databasfråga som resulterar i att man inte hittar det eftersökta göra att programmet avslutas.

Att skapa egna undantagsklasser

Egna undantagsklasser erhålls genom att skapa subklasser till Exception (checked) eller till 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 metodanrop som sänt felet vidare. Metoden printStackTrace() anropas automatiskt när ett undantag avbryter ett program.

Kasta undantag med rätt abstraktionsnivå

Det är förvillande om en metod kastar ett undantag 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 NoRealSolutionException("No real solutions exists");
   }
}
```


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 undantag!

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

Dokumentation av undantag

Både kontrollerade och okontrollerade undantag som kastas är en del av metodens kontrakt och skall 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 {
   . . . .
   . . . .
}
```


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");
   }
   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();
      }
   }
```


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");
   }
   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();
      }
   }
}
```

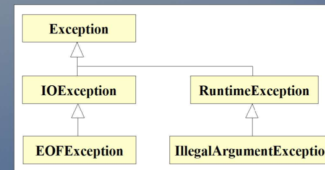
finally-blocket

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

```
import java.io.*;
public final class CatchInsideFinally {
  public static void main(String[] args) {
    catchInsideFinally("test.txt");
  }
  private static void catchInsideFinally(String aFileName) {
    //Declared here to be visible to 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();
      }
    }
  }
}
```


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 RuntimeException {
    . . .
  }
}
```

WRONG!

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

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


Exceptions: Sammanfattning

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

Testning

Översikt

– Testning, verifiering och avlusning
– Olika typer av testning
– Black box och White box
– Testplan, testsvit och testfall
– Enhets- och regressionstestning med JUnit
– Kodtäckningsanalys med EclEmma

Testning, verifiering, avlusning – vad är skillnaden?

- Testning kan påvisa *närvaron* av fel, men kan i allmänhet inte bevisa frånvaron av fel
  - – de flesta algoritmer har en oändlig indatamängd vilket omöjliggör uttömmande testning.
- Verifiering kan *bevisa* frånvaron av fel
  - – kräver formell symbolisk analys av programmet.
  - – svårare att automatisera än testning.
  - – testning *är inte* verifiering!
- Avlusning söker efter *orsaken* till fel avslöjade av *misslyckade test*.
  - – avlusning *är inte* testning!
  - – att planlöst rätta uppkomna fel utan en genomtänkt testplan kallas *Trial-And-Error* – inte testning.

Ickefunktionell testning – mot ickefunktionella krav

Ickefunktionell testning undersöker övergripande kvalitetsaspekter som inte direkt avser innehållet i systemets tjänster, utan:

• Användbarhet (usability)	är systemet lätt att använda?
• Robusthet (robustness)	klarar systemet av oförutsedda situationer utan att krascha?
• Tillförlitlighet (reliability)	systemet är tillförlitligt om det är korrekt och robust.
• Tillgänglighet (availability)	hur ofta går systemet ner?
• Effektivitet (efficiency)	klarar systemet av att ge efterfrågad service inom rimlig tid och med rimliga resurser?
• Flexibilitet (flexibility)	kan systemet anpassas för individuella behov?
• Skalbarhet (scalability)	kommer systemet att fungera korrekt även om om det problem som handhas skalas upp en magnitud?
• Säkerhet (safety)	orsakar systemet skada (autonoma fordon ...)?
• Säkerhet (security)	är systemet skyddat mot intrång av obehöriga?

Några testbegrepp – från låg till hög nivå

Enhetstestning
Testning av funktionaliteten hos en metod, klass, modul, ...

Regressionstestning
Upprepning av (relevanta) test efter förändring. Vanligen enhetstest.

Integrationstestning
Testning av funktion, prestanda och pålitlighet vid sammansättning av delsystem.

Systemtestning
Testning av systemets kvalitet mot *beställarens kravspecifikation*. Funktion, prestanda, tillförlitlighet, stresstålighet, skalbarhet, ...

Acceptanstestning
Uppfyller systemet *verksamhetskrav* för leverans till beställaren?

Användbarhetstestning
Uppskattning av systemets *nytta för slutanvändaren*. Referensgrupper, intervjuer, m.m. kan användas för att värdera kognitionspsykologiska aspekter på systemets användbarhet, inlärningskurva, m.m.

Regressionstestning

Att *regressionstesta* innebär att upprepa test efter varje förändring.

I en utvecklingscykel med gradvisa utvecklingssteg varvade med refaktoreringar är det viktigt att momenten utförs i en genomtänkt ordning.

Lägg inte till ny funktionalitet i en dåligt strukturerad design! Refaktorera först, bygg ut sen och testa löpande under hela processen.

Black Box och White Box

Specifikationsbaserad testning (black box testing)

Denna form av testning undersöker systemets *funktionalitet* med utgångspunkt från dess specifikation, utan kunskap om implementationsdetaljer, och utförs av beställaren.

Implementationsbaserad testning (white box testing)

Denna form av testning kräver insyn i implementationsdetaljer och utförs av implementatören.

Ex. Olika former av *kodtäckningsanalys*

- *Satstäckning*: i vilken grad alla satser exekveras.
- *Grentäckning*: i vilken grad alla grenar i valsisituationer exekveras.
- *Spårtäckning*: i vilken grad alla möjliga exekveringsspår exekveras.

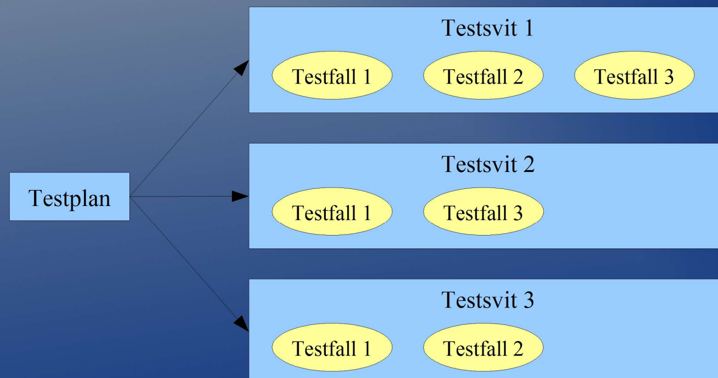
Anm. I detta sammanhang kan *rollerna* beställare och implementatör spelas av samma aktör. Beställaren kan vara en aktör i ett oberoende testteam, eller utvecklaren själv.

Testbegrepp – Testplan – testsvit - testfall

Ett *testfall* beskriver en uppsättning testdata, förvillkor, eftervillkor och förväntade resultat i syfte att visa att ett specifikationskrav är uppfyllt.

En *testsvit* är en samling testfall. Ett testfall kan ingå i flera testsviter.

En *testplan* är en samling testsviter.



Avlusningstekniker – handexekvering och tabellering

Avlusning av programkod (*debugging*) kan göras med olika grad av verktygsstöd. Nyttan av manuella metoder skall ej underskattas.

Manuell ”handexekvering” av koden på papper ger ofta nya insikter.

Tabellering av objekttillstånd kan vara ett effektivt sätt att hitta fel när inte andra metoder gett resultat. Dokumentera förändringar i instansvariablernas värden efter varje metodanrop – eller oftare.

	x_1	x_2	…	x_n
$steg_1$				
$steg_2$				
…				
$steg_m$				

Avlusningstekniker – Verbala genomgångar

Förklara för någon vad programmet gör!

- En annan person kan se problemet med ”friska ögon” och hitta felet.
- Processen att förklara innebär en bearbetning som kan leda till att du själv kommer till insikt om problemet.

Gruppbaserade aktiviteter för *kodrevision*

- Agila metoder
- Extreme Programming (XP)

Enhetstestning

Varje enhet i en applikation kan testas:

- metod
- klass
- modul (paket i Java)

Enhetstestning planeras tidigt i utvecklingen, innan implementeringen av systemet påbörjas.

Testplanen är ett viktigt komplement till specifikationen.

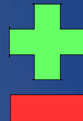
Ex. En TCK (Technology Compatibility Kit) för en JSR (Java Specifikation Request) består av en serie testsviter för att kontrollera om en komponent beter sig enligt specifikationen. En TCK kan innehålla tusentals testfall.

Ex. En mobiltelefonutvecklare konstruerar blåtandsapplikationer. En underleverantör av blåtandskomponenter använder TCK JSR82, Bluetooth för att testa att den levererade komponenten konformerar med beställarens krav.

Enhetstestning – Att konstruera testfall

De flesta algoritmer har en oändlig indatamängd och vi kan då inte testa alla tänkbara indatavärden och göra en *uttömmande* testning. Det vi *kan* göra är att försöka välja testfall som avslöjar vanliga programmeringsmisstag.

- testa *positivt* – att kontraktet är uppfyllt
- testa *negativt* – kan kontraktet brytas?
- testa gränsfall, t.ex.
  - 0, 1, n
  - sök i en tom samling
  - sätt in i en full samling
- leta efter *off-by-one-errors*



Enhetstestning med JUnit

JUnit är ett *ramverk* för automatiserad enhetstestning av javakod. De flesta utvecklingsverktyg för Java har stöd för testning med JUnit.

Ett vanligt scenario är att man testar en klass genom att skriva en tillhörande testklass som testar klassens konstruktorer och metoder. Test kan utfalla på tre olika sätt:

- lyckat test
- misslyckat test p.g.a. att något uppställt villkor ej uppfylls
- exekveringsfel

Utfallen rapporteras av verktyget i tabellform. Orsaken bakom misslyckade test kan därefter analyseras med verktygets avlusare.

Enhetstestning med JUnit

Annotationstyper i paketet org.junit sätts framför metoder i testklassen:

- @BeforeClass
- @Before
- @AfterClass
- @After
- @Test

```
@Test
public void testSomeProperty() {
    SomeClassToBeTested obj = new SomeClassToBeTested();
    obj.someMutator(someValue);
    assertEquals(42,obj.getValue());
}
```

För att testa olika egenskaper används klassmetoder i klassen org.junit.Assert.

Utvecklingsverktyg som stödjer JUnit registrerar utfallen av Assert-anrop.

Om inget Assert-anrop fallerar i en testmetod rapporteras testet som lyckat i verktygets grafiska gränssnitt.

Om något Assert-anrop fallerar rapporteras testet som misslyckat i verktygets grafiska gränssnitt.

Enhetstestning med JUnit - syntaxexempel

```
import org.junit.*;
public class TestWhatever {
  @BeforeClass
  public static void setUpClass() {
    // Exekveras en gång innan testen (t.e.x. uppkoppling mot databas, etc)
  }
  @Before
  public void setUp() {
    // Exekveras alltid innan exekvering av en testmetod.
  }
  @Test
  public void testSomeProperty() {
    // Kod som testar egenskaper hos den testade klassen.
  }
  …

  @After
  public void tearDown() {
    // Exekveras alltid efter exekvering av en testmetod.
  }
  @AfterClass
  public void tearDown() {
    // Exekveras en gång efter testen (t.ex. för att frigöra resurser).
  }
```


Enhetstestning med JUnit – klassen Assert

Ett urval av metoder i klassen org.junit.Assert.
– Metoderna med **double** som parametrar testar om |(un)expected – actual| ≤ delta
– Flera av metoderna finns även i en version som tar ett textmeddelande som parameter
public static void assert*Something*(String message, ...)

```
public static void assertTrue(boolean condition)
public static void assertFalse(boolean condition)
public static void assertEquals(long expected, long actual)
public static void assertNotEquals(long unexpected, long actual)
public static void assertEquals(double expected, double actual, double delta)
public static void assertNotEquals(double unexpected, double actual, double delta)
public static void assertSame(Object expected, Object actual)
public static void assertNotSame(Object unexpected, Object actual)
public static void assertNull(Object object)
public static void assertNotNull(Object object)
```


Enhetstestning med Junit – Ex. testmetod

```
import org.junit.Test;
import static org.junit.Assert.*;   // importera och synliggör alla klassmetoderna i Assert
public class TestWhatever {
  …
  @Test
  public void testSomeProperty() {
    SomeClassToBeTested obj = new SomeClassToBeTested();
    obj.someMutator(someValue);
    assertEquals(42,obj.getValue());
  }

  @Test
  public void testSomeOtherProperty() {
    SomeClassToBeTested obj = new SomeClassToBeTested();
    obj.someOtherMutator(someOtherValue);
    assertNotEquals(97,obj.getValue());
  }
}
```

Kodduplicering!

Kodduplicering!

Enhetstestning med JUnit – Testfixtur

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestWhatever {
   private SomeClassToBeTested obj;

   @Before
   public void setUp() {
      obj = new SomeClassToBeTested();
   }

   @Test
   public void testSomeProperty() {
      obj.someMutator(someValue);
      assertEquals(42,obj.getValue());
   }

   @Test
   public void testSomeOtherProperty() {
      obj.someOtherMutator(someOtherValue);
      assertNotEquals(97,obj.getValue());
   }
}
```

Testfixtur

Exekveras *alltid* *innan* exekvering av en testmetod

Varje test får ett nytt fräscht objekt oavsett i vilken ordning testen körs

Enhetstestning med JUnit – Exempel

Test skall kunna skrivas med utgångspunkt från gränssnitt och specifikationer.

```
/**
 * Keeps track of the lengths of the shortest
 * and the longest non null string seen.
 */
public interface IMinMaxMemory {
   void remember(String s) throws IllegalArgumentException;
   int getMinLength();
   int getMaxLength();
}
```

Problem:

- I en implementation av IMinMaxMemory finns *oändligt* många möjliga tillstånd.
- Kan vi ändå definiera ett litet antal test som avslöjar de troligaste misstagen?

Enhetstestning med JUnit – Exempel

Vi skriver en testklass med en enkel fixtur samt fyra testmetoder.
De valda värdena är av underordnad betydelse, men inte deras *inbördes storleksförhållanden*.

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class MinMaxMemoryTest {
   private IMinMaxMemory memory;
   @Before
   public void setUp() {
      memory = new MinMaxMemory();
      memory.remember("333");
      memory.remember("55555");
   }
   @Test(expected=IllegalArgumentException.class)
   public void testRememberNull() {
      memory.remember(null);
   }
```

```
@Test
public void testOrderSmall() {
   memory.remember("1");
   assertEquals(1,memory.getMinLength());
   assertEquals(5,memory.getMaxLength());
}
```

```
@Test
public void testOrderMiddle() {
   memory.remember("4444");
   assertEquals(3,memory.getMinLength());
   assertEquals(5,memory.getMaxLength());
}
```

```
@Test
public void testOrderLarge() {
   memory.remember("7777777");
   assertEquals(3,memory.getMinLength());
   assertEquals(7,memory.getMaxLength());
}
```

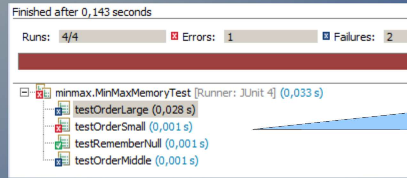

Enhetstestning med JUnit – Exempel

En möjlig implementering av IMinMaxMemory

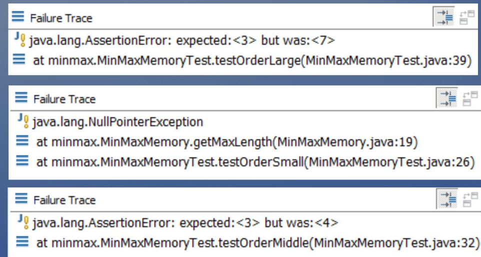
```
public class MinMaxMemory implements IMinMaxMemory {
   private String shortest = null;
   private String longest = null;

   public void remember(String s) {
      if ( s == null )
         throw new IllegalArgumentException("null string passed to remember");
      if ( shortest == null || s.length() <= shortest.length() )
         shortest = s;
      if ( longest == null || s.length() >= longest.length() )
         shortest = s;
   }
   public int getMinLength() {
      return shortest.length();
   }
   public int getMaxLength() {
      return longest.length();
   }
}
```

Är implementeringen korrekt?

Enhetstestning med JUnit i eclipse – Exempel



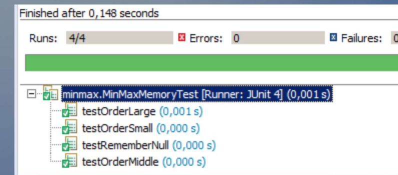
– ett misslyckat test
– ett exekveringsfel
– ett lyckat test
– ett misslyckat test




```
...
if ( longest == null || s.length() >= longest.length() )
    shortest = s;
}
...
```

copy paste bug?

Enhetstestning med JUnit i eclipse – Exempel



Wohooo!

```
...
if ( longest == null || s.length() >= longest.length() )
    longest = s;
}
...
```

Felet rättat!

- *Kan vi slå oss till ro nu och betrakta klassen som* korrekt?
- *Är all kod testad?*
- *Är testen relevanta?*
- *Vilka slutsatser kan vi dra resp.inte dra?*

EclEmma – en eclipse-plugin för kodtäckningsanalys

EclEmma beräknar och visualiserar täckningsgraden hos JUnit-test med färgmarkeringar i källkoden.

```
28   @SuppressWarnings("unchecked")
29   public SortedBuffer(Comparator<? super T> comp,
30       assert capacity > 0;
31 //        buffer = (T[])new Comparable[capacity];
32       buffer = (T[])new Object[capacity];
33       size = 0;
34       if ( comp != null ) {
35           this.comp = comp;
36           useNaturalOrderComparator = false;
37       } else {
38           this.comp = (Comparator<? super T>)Compa
39           useNaturalOrderComparator = true;
40       }
41       assert invariant();
42   }
```

1 av 2 grenar otestade

testat

delvis testat

otestat

Fotnot. EclEmma = Emma för eclipse.

EclEmma – en eclipse-plugin för kodtäckningsanalys

EclEmma visar även statistik över testens täckningsgrad per metod i den testade klassen.
Exemplet demonstreras under kommande övningspass.

SortedBufferTest (2016-nov-30 17:02:55)

Element	Coverage	Covered Instructions	Missed Instru...	Total Instructions
SortedBuffer<T>	74,6 %	249	85	334
isOrdered()	0,0 %	0	28	28
SortedBuffer(Comparator<? super T>, int)	64,3 %	27	15	42
checkIfComparable(T)	33,3 %	7	14	21
insert(T)	70,4 %	19	8	27
remove(int)	85,5 %	47	8	55
get(int)	73,3 %	11	4	15
invariant()	84,6 %	11	2	13
isEmpty()	71,4 %	5	2	7
isFull()	80,0 %	8	2	10
SortedBuffer(int)	100,0 %	5	0	5
checkForSpace()	100,0 %	11	0	11
doTheInsertion(T)	100,0 %	40	0	40
getCapacity()	100,0 %	4	0	4
getSize()	100,0 %	3	0	3
toString()	100,0 %	45	0	45

Enhetstestning med Junit i eclipse - konfiguration

Några adresser:
http://junit.org/junit4/index.html
http://junit.org/junit4/javadoc/latest/index.html
http://www.eclemma.org/

Lägga till JUnit-biblioteket till ett projekt i eclipse
Högerklicka på projektnoden och välj
Build Path > Add Libraries > JUnit > JUnit4 > FINISH

Skapa testklass i eclipse
Högerklicka på klassens nod i Package Explorer-fliken och välj
New > JUnit Test Case > *gör diverse val* > Next >
> *välj metoder att testa* > FINISH

Installera EclEmma-plugin i eclipse (kräver uppkoppling)
Help > Eclipse Marketplace > Find: *eclemma* > Install > FINISH