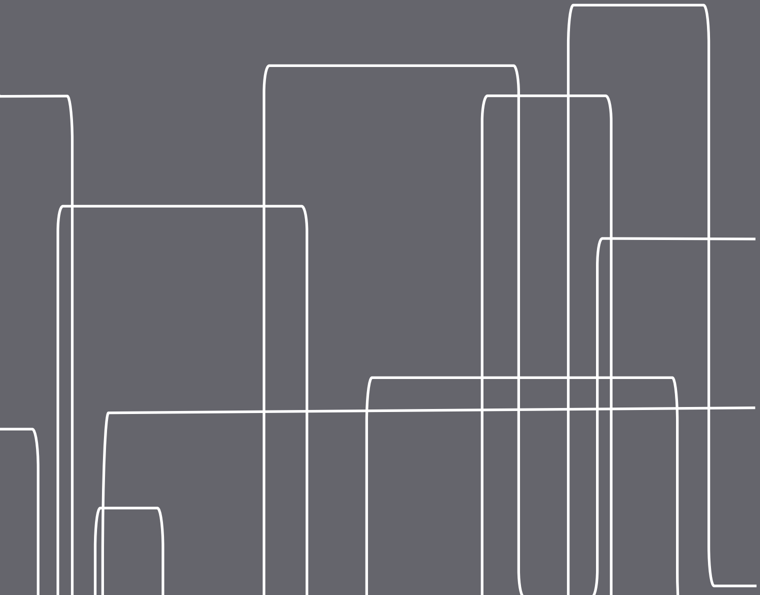
Objektorienterad Programkonstruktion

Föreläsning 4

8 nov 2016



Nästade klasser

I Java går det att deklarera en klass inuti en annan klass. Vi kallar detta för att en **yttre klass** innehåller en **inre klass**.

Det finns fyra sorters inre klasser

- en **statisk nästad klass** fungerar som en vanlig klass definierad i samma paket, men finns i samma fil
- en **inre klass** kan bara instansieras i en instans av den yttre klassen, och har tillgång till alla delar av denna instans, inklusive privata fält och metoder
- en **lokal inre klass** definieras inuti ett annat kodblock, och kan sedan användas fritt inuti detta block
- en **anonym inre klass** definieras och instansieras vid ett specifikt tillfälle och kan inte användas någon annanstans

Static Nested Class

- Ett sätt att packetera klassdefinitioner i en fil, om man tycker att de hör samman.
- Den har inte tillgång till instansvariabler eller -metoder från den yttre klassen.

```
public class OuterClass {
    static class StaticNestedClass {
    ...
    }
}

OuterClass.StaticNestedClass nestedObject =
                   new OuterClass.StaticNestedClass();
```

Inner Class

- Kan bara instansieras i en instans av den yttre klassen
- Har tillgång till alla delar av denna instans, inklusive privata fält och metoder
- Kan användas var som helst i den yttre klassen
- Lämplig om klassen bara används av den yttre klassen

```
class OuterClass {
    class InnerClass {
        ...
    }
}
```

- Kan instansieras utifrån, om man använder en instans av den yttre klassen:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Local Class

- Definieras inuti ett kodblock, finns bara i detta block
- Är oftast väldigt enkel och/eller specifik
- Har tillgång till alla instansvariabler och -metoder

```
class OuterClass {
    public void someMethod(){
        class LocalClass{
            ...
        }
        LocalClass myLocalClass = new LocalClass();
        ...
    }
}
```

Anonymous (sub-)class

- Deklareras och instansieras direkt. Existerar bara där den deklareras, kan ej användas igen, ens I samma kodblock (om man inte deklarerar den igen, förstås)
- Måste ärva en annan klass eller implementera ett gränssnitt
- Har tillgång till instanvariabler och -metoder

```
class OuterClass ... {
    ...
    // inside a method in OuterClass :
    myButton.addActionListener(new ActionListener(){
        public void actionPerformed( ActionEvent e ){
            System.out.println(“Anonymous!”);
        }
        });
    ...
}
```

Abstrakta Klasser

- Deklareras med nyckelordet `abstract`
- Går inte att instansiera, utan är en mall att ärva från
- En klass som har minst en metod som är deklarerad som abstract måste själv vara abstrakt
- För att en ärvande klass ska gå att instansiera måste den explicit deklarera alla metoder som är abstrakta I superklassen, annars blir den ärvande klassen också abstrakt
- Gränssnitt och deras metoder är implicit abstrakta
- Abstrakta klasser behöver inte implementera alla metoder ur gränssnitt
- Abstrakta klasser kan ha statiska fält och metoder

```
public abstract class MyAbstractClass {

    abstract void doAbstractStuff();

}
```

Abstrakta klasser eller gränssnitt?

- Om man deklarerar några instansvariabler och konkreta metoder, men inte vill specificera alla metoder konkret, bör man använda en **abstrakt klass**
- Om man inte har några instansvariabler, och alla metoder är abstrakta, bör man använda ett **gränsnitt**
- Som exempel kan vi tänka oss
  - den abstrakta klassen **Kort**, som innehåller information om storlek, och har en gemensam metod för att lägga korten i en hög, med konkreta subklasser som är t.ex spelkort, visitkort, mm
  - gränsnittet **Flippable**, som gäller för alla kort (och andra föremål) som går att vända på

Polymorfism

- Man kan deklarera variabler av en superklass och instansiera dem med en subklass

```
Human putte = new SimTek();
```

- Java kommer att behandla variabeln som den **deklararerade** typen för att avgöra om en metod eller ett fält finns.
- Java kommer att behandla de som den **instansierade** klassen för att avgöra icke-statiska metoder eller fält skall tolkas, sk **dynamisk bindning**.
- **Statiska** fält och variabler tas från variabeltypen, oavsett instansiering

Polymorfism, exempel

```
Human putte = new Human(25, "Putte");
Human anna = new Fysiker(2009,25,"Anna");
```

Denna rad anropar `toString()`-metoden från Human:

```
System.out.println(putte.toString());
```

Denna rad anropar `toString()`-metoden från Fysiker:

```
System.out.println(anna.toString());
```

Denna rad går **inte** att kompilera (metoden finns ej i Human):

```
System.out.println(anna.getYear());
```

Deklarera med gränssnitt

- Man kan deklarera en variabel med typ som är ett gränssnitt

```
Comparable putte = new Human();
```

- man kan nu anropa:

```
putte.compareTo(anna);
```

- eftersom Comparable inte känner till de metoder som specifikt har deklarerats i Human går det inte att skriva:

```
putte.getAge(); // Går inte!
```

- men följande går förstås bra!

```
((Human)putte).getAge();
```

Deklarera med gränssnitt, exempel

```
public boolean isSame(Differing obj1, Differing obj2) {
    if ( obj1.getDifferenceFrom(obj2) == 0)
        return true;
    else
        return false;
}
```

Här antar vi att vi har deklarerat följande gränssnitt:

```
public interface Differing{
    Public int getDifferenceFrom(Differing d);
}
```

Exceptions (Särfall)

- Javas sätt att hantera interna felmeddelanden.
- En metod som träffar på ett problem skickar ett felmeddelande uppåt i anropsstacken:

Metod **A** anropar metod **B**. **B** anropar i sin tur **C**. **C** kan inte utföra sin uppgift som förväntat, utan kastar en **exception** (undantag) som fångas av metod **B**. **B** har nu information om att **C** inte fungerade som tänkt, och kan åtgärda felet. Om **B** inte tar hand om felet skickas det vidare till **A**, som nu kan anta att **B** inte fungerade som tänkt.

Att kasta särfall (Throwing Exceptions)

- Metoden som vill kasta ett undantag gör detta med kommandot throw. Särfall ska vara av typen `Throwable` (eller någon av dess underklasser, oftast `Exception`)

```
throw new IOException("No such file!");
```

- En metod som inte vill fånga ett särfall, utan skickar detta vidare uppåt i hierarkin måste deklarera denna möjlighet I förväg, genom att använda nyckelordet `throws`, t.ex:

```
public void doErrorproneOperation() throws IOException{
    doErrorProneIO();
}
```

Att fånga särfall (Catching Exceptions)

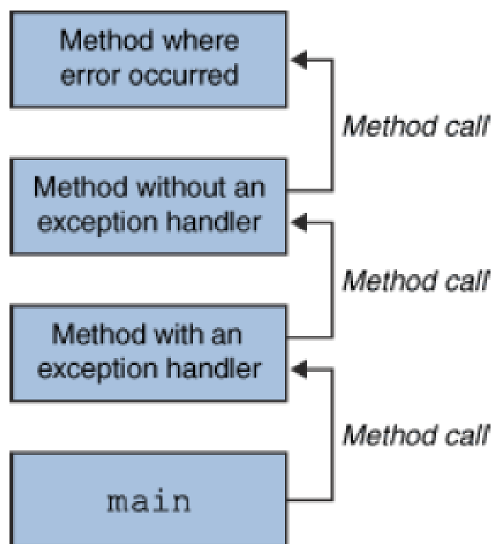
- För att hantera särfall som kastas av en annan metod använder man en så kallad **try/catch-**formulering, t.ex:

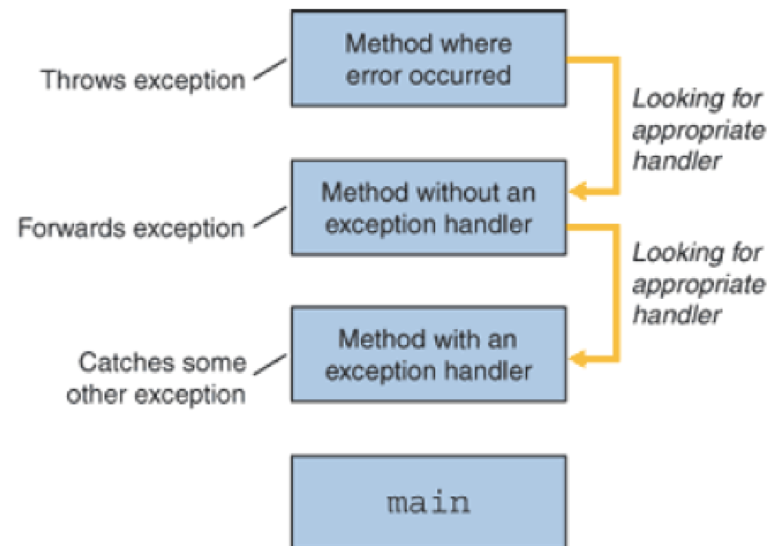
```
try{
    rawResult = doFileIO(myFile); //throws IO related Exceptions
    result = parse(rawResult); // Only run if no exception
occured
}catch(FileNotFoundException e){
    System.err.println("FileNotFoundException: " +
e.getMessage());
    return defaultEmptyString;
}catch (IOException|SQLException ex){
    System.err.println("Caught IO/SQL-Exception:"+
ex.getMessage());
    result = generateDefaultResult();
}finally{ // This is always called!
    doCleanup();
}

doSomeMoreStuff();
return result;
```

Exceptions

- En metod som träffar på ett problem skickar ett felmeddelande uppåt i anropsstacken
- Den metod som fångar exceptions av rätt sort anses kunna hantera dem, om ingen fångar rätt exception stoppas programmet





Kedjor

- En metod kan hantera ett särfall genom att skicka ett nytt, t.ex:

```
try {
    doErrorproneIO();
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

- Man kan sedan komma åt det underliggande felet med

```
Throwable getCause() // Finns i alla Throwable
Throwable initCause(Throwable) // -"-

Throwable(String, Throwable) // Konstruktorer
Throwable(Throwable)
```

Feltyper

- **Checked Exceptions** - de flesta typer av Exceptions. Dessa måste hanteras med **try-catch** eller **throws**
- **Unchecked Exceptions** - behöver inte hanteras. Dessa är eller ärver från klasserna **Error** och **RuntimeException.**
  - **Error** och dess underklasser används för externa fel som är svåra att förutse eller göra något åt, t.ex om en fil slutar att existera under pågående inläsnen för att disken gick sönder.
  - **RuntimeException** och dess underklasser används för att rapportera interna fel, som division med noll, pekare till objekt som inte finns, mm
- Om man inte hanterar unchecked exceptions så kommer programmet att avbrytas och ett felmeddelande skrivs ut på skärmen. Man kan välja att hantera även dessa fel, i mer avancerad kod.