Objektorienterad Programmering (TDDC77)

Föreläsning XIV: Undantag, Design

Ahmed Rezine

IDA, Linköpings Universitet

Hösttermin 2017

Outline

Hashing

Undantag

Design

Outline

En frukt har ett namn

```
class Fruit {
    private String name;

    public Fruit(String str){this.name = str;}

    public String getName() {return name;}

    public void setName(String str) {this.name = str;}

    public String toString(){return name;}

}
```

Man kan lägga en frukt i en korg ...

```
import java.util.*;

//Basket päron melon apelsin melon
public class Basket {

    public static void main(String[] args) {
     Set<Fruit> set = new HashSet<Fruit>();
     for(String in: args)
         set.add(new Fruit(in));

     System.out.println("Mängden av olika input är: " + set);
    }
}
```

En frukt med bättre hash funktion

```
class Fruit {
    private String name;

    public Fruit(String str){this.name = str;}

    public String getName() {return name;}

    public void setName(String str) {this.name = str;}

    public String toString(){return name;}

    public int hashCode(){return name.hashCode();}

    public boolean equals(Object other){
     if(other==null || !(other instanceof Fruit))
         return false;
     return name.equals(((Fruit)other).name);
    }
}
```

Hashing

- Alla klasser som implementerar “Collection” eller “Map” som innehåller Hash i namnet (HashSet, LinkedHashSet, HashMap, HashTable ...etc) använder något som heter hashing för att öka hastigheten på vissa operationer, exempelvis när man ska göra snabba sökningar i en lista
- Det ingår inte i kursen att förstå exakt hur det fungerar, men ni måste förstå tillräckligt för att kunna använda det
- Idéen är att man beräknar för varje objekt ett så kallad hash (heltal)

Hashing

- Heltalet används sedan för att dela upp mängden i flera mindre mängder, vill man veta vilken del man ska leta i behöver man bara hasha värdet man letar efter
- Hashfunktionen måste man alltid returnera samma värde för samma objekt, och för alla andra objekt där **equals()** returnerar sant
- För er ordlistlaboration så räcker det bra att returnera värdet för textsrängens **haschCode()**

Outline

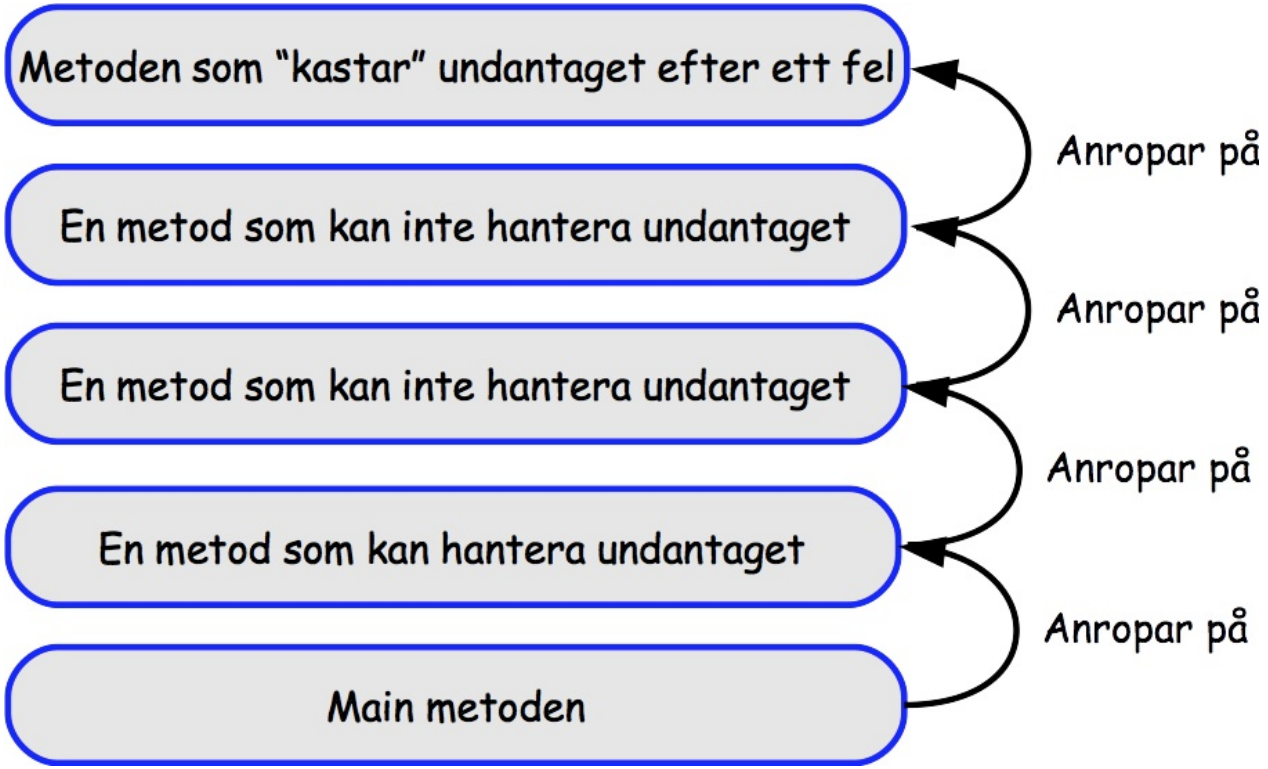
Undantag

```
class SGradeParsing{

    private static int getGradeValue(String input){
     int val = Integer.parseInt(input);
     return val;
    }

    private static String getGradeName(String input)
    {
     int grade = getGradeValue(input);
     switch(grade){
     case 5: return "\"Mycket väl godkänd\"";
     case 4: return "\"Väl godjänd\"";
     default: return "\"Godkänd\"";
     }
    }

    public static void main(String[] args) {
     String errorMsg= "Translates numerical grade\nSyntax: GradeParsing x\nwhere x is a
     if(args.length != 1){
         System.out.println(errorMsg);
     }else{
         System.out.println("The grade " + args[0] + " corresponds to " + getGradeName(a
     }
    }
}
```

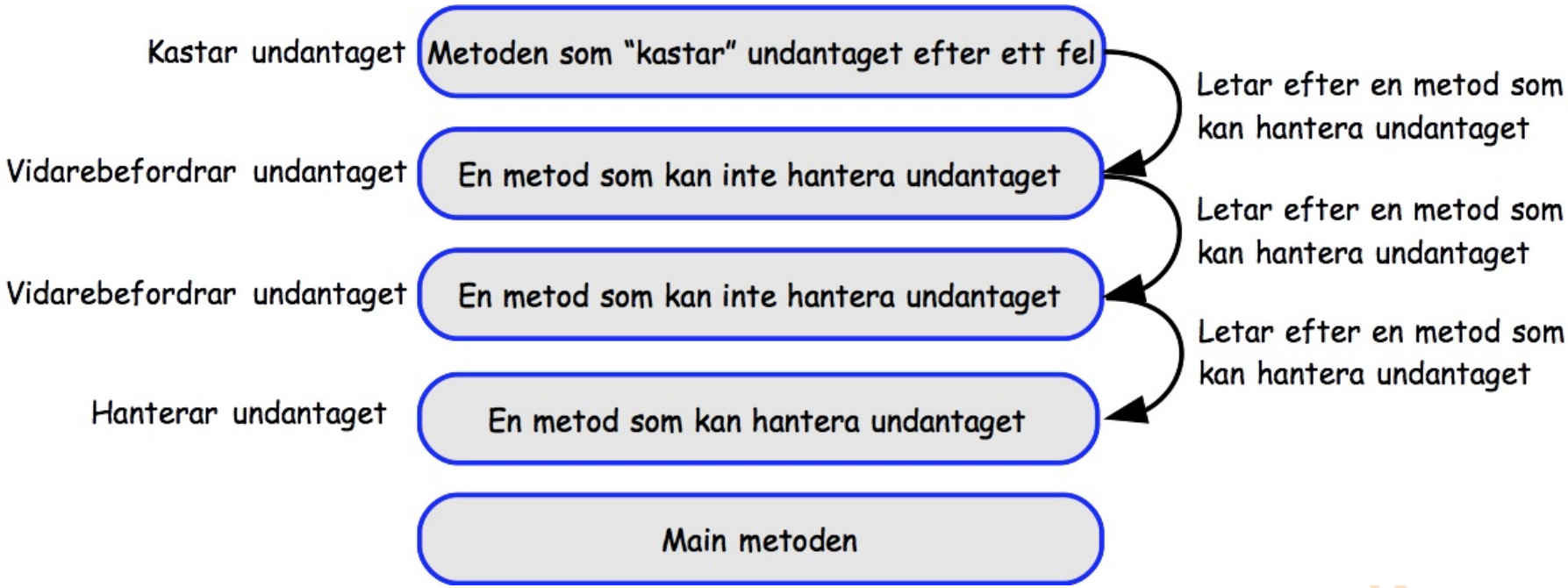
Undantag - Exceptions

- Fel som uppstår under körning kallas undantag (exceptions)
- Metoder kan kasta undantag (throw)
- Metoder kan fånga undantag (catch)

Undantag - Exceptions

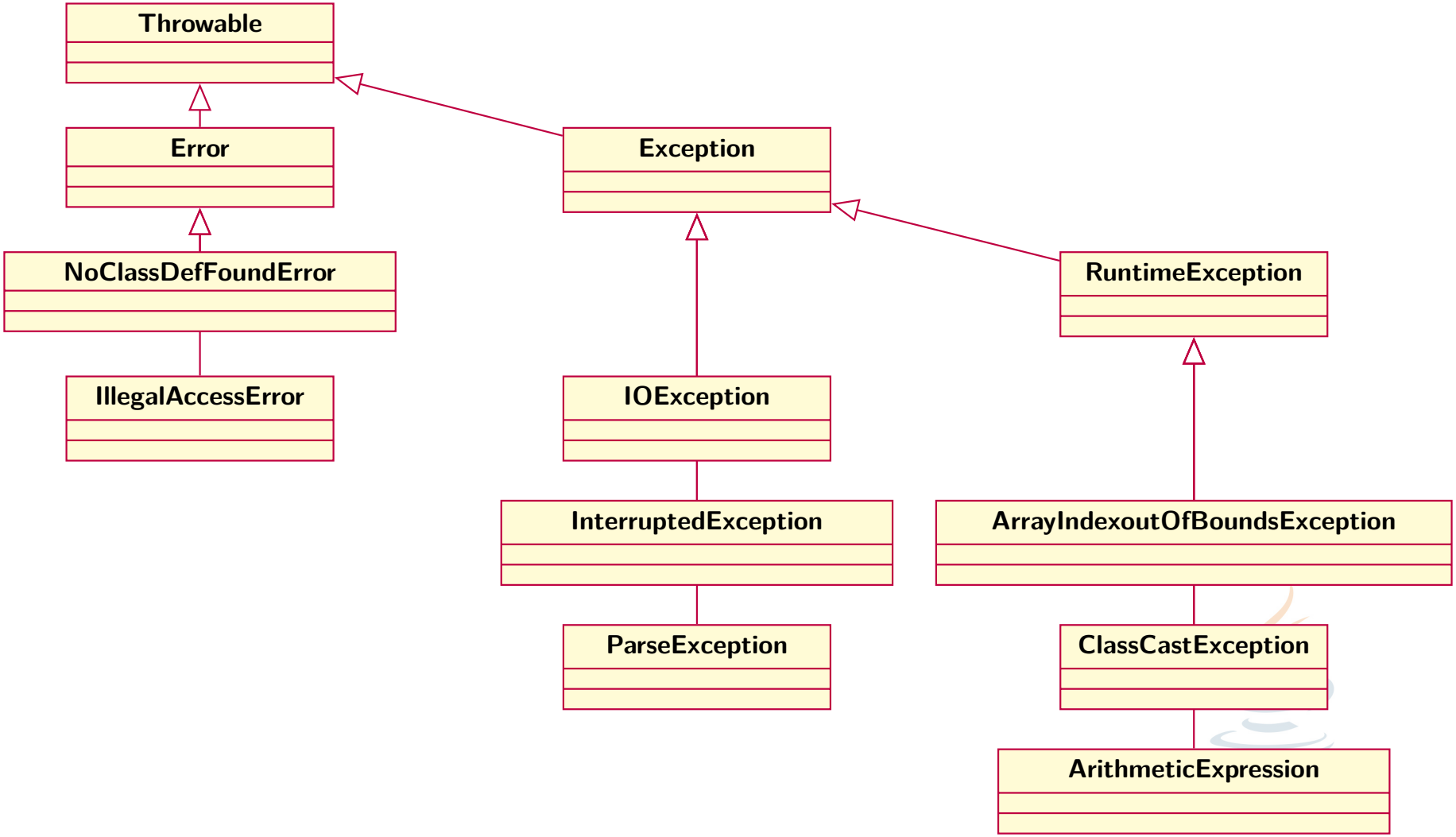


Undantag - Exceptions



Typer av undantag

Vissa undantag måste man förvarna för, medan andra kan uppstå när och var som helst



Typer av undantag

- **Throwable** - Allt som kan kastas och fångas
- **Errors** - Jättealvarliga fel i själva java. Normalt sett bör dessa inte hanteras av programmeraren.
- **Exceptions** (utom **RuntimeException**) - Förutsedda undantag, dessa är del av normal funktion och måste hanteras
- **RuntimeExceptions** - Undantag som kan inträffa under normal körning, men normal sett beror på att programmeraren gjort fel. Behöver inte fångas

Typer av undantag

- Checked exception:
  - e.g. **Exception, ParseException, IOException,** ...
  - Måste hanteras (dvs. antigen med **try-catch** eller genom att skriva **throws** i metodens deklaration)
  - När undantaget förväntas uppstå
  - När det går att rädda systemet
- Unchecked exception:
  - e.g. **RuntimeException, ArrayIndexOutOfBoundsException,** ...
  - Måste inte hanteras ((dvs. behöver inte ha en **try-catch** eller skriva **throws** i metodens deklaration))
  - När undantaget inte förväntas uppstå
  - Men ifall det ske, “Something went horribly wrong!”

Kasta vidare undantag

- En kedja av metoder kan ha throws i sin signatur
- Lämnar över ansvaret att fånga högre upp i anroskedjan

Undantag

```
class GradeParsing{

    private static int getGradeValue(String input) throws GradeValueException{
     int val = Integer.parseInt(input);
     if(!(3 <= val && val <= 5)){
         throw new GradeValueException("Grades should be between 3 and 5. You entered "
     }
     return val;
    }

    private static String getGradeName(String input) throws GradeValueException
    {
     int grade = getGradeValue(input);
     switch(grade){
     case 5: return "\"Mycket väl godkänd\"";
     case 4: return "\"Väl godjänd\"";
     default: return "\"Godkänd\"";
     }
    }

    public static void main(String[] args) {
     String errorMsg= "Translates numerical grade\nSyntax: GradeParsing x\nwhere x is a
     if(args.length != 1){
         System.out.println(errorMsg);
     }else{
         try{
          System.out.println("The grade " + args[0] + " corresponds to " + getGradeName(
         }catch(NumberFormatException e){
          System.out.println(errorMsg);
         }catch(GradeValueException e){
          System.out.println(e.getMessage());
         }
     }
    }
```

Egna undantag

```
class GradeValueException extends Exception{
    public GradeValueException (String message){
     super(message);
    }
}
```

Stack trace

```
Exception in thread "main" java.lang.NumberFormatException: For in
      at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDeci
      at java.lang.Double.parseDouble(Double.java:510)
      at lincalc.LinCalcJohn.calc(LinCalcJohn.java:139)
      at lincalc.LinCalc.calc(LinCalc.java:35)
      at lincalc.LinCalc.evaluate(LinCalc.java:45)
      at lincalc.LinCalc.main(LinCalc.java:28)
```

```
class Undantag {

  public static void main(String[] args) {
    System.out.println("Prog börjar");
    foo1();
    System.out.println("Prog slutar");
  }

  public static void foo1(){
    System.out.println("foo1 börjar");
    try{
     foo2();
    }
    catch (ArithmeticException e){
      System.out.println("foo1 catch1");
    }
    catch(NumberFormatException e){
      System.out.println("foo1 catch2");
     }
    finally{
      System.out.println("foo1 finally");
    }
    System.out.println("foo1 slutar");
  }



  public static void foo2(){
    System.out.println("foo2 börjar");
    try{
      System.out.println(Integer.parseInt("a"));
    }
    catch(ArithmeticException e){
      System.out.println("foo2 catch1");
    }
    /*catch(NumberFormatException e){
      System.out.println("foo2 catch2");
      }*/
```

Outline

Design

- Klasser: substantiv, t.ex. Item, DatabaseInitializer
- Gränssnitt: brukar beskriver en egenskap, adjektiv, t.ex. Serializable, Cloneable
- Metoder: verb, t.ex. calculateSum(), initialize()
- Metoder som returnera ett booleskt värde: fråga, t.ex. isBiggerThan(), hasElement()
- Metoder som hämtar eller sätter en attribut, börja med get, respektive set.

Isberg

- Man brukar liknar en väldesignad klass med ett isberg: 80% av massan ligger dold under vattenytan,
- På samma sätt bör klasser bara ha några få synliga metoder

En uppgift

"En klass ska göra en sak, och göra den bra"

- Man ska alltså undvika att ha med funktionalitet som inte logiskt hör ihop med klassen

Undvika överraskningar

"Principle of least surprise"

- Om någonting kan lösas på flera olika sätt, välj den lösning som skulle överraska användaren minst

Dokumentera alltid

- Dokumentera alltid alla publika metoder och variabler
- Normal sett bör man även dokumentera privata metoder och variabler

Bottom up

- Börja bygga från grunden
- Man bygger flera smådelar som successivt sammanfogas till ett större system
- Risk för att helheten blir virrig och “ogenomtänkt”
- Men ofta kan man få körbar, (delvis) fungerande kod snabbt

Top down

- Börja från ett tänkt färdigt system
- Bryt ned systemet i fler mer detaljerade delar
- Risk för att man gör uppdelningar som inte blir bra när man skriver koden
- men ofta enklare att genomföra systematiskt

Båda

- Vilken man väljer är helt upp till en själv
- Vissa gillar att blanda