Objektorienterad Programmering DAT043

Föreläsning 5
29/1 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Repetition: Arv

- En **subklass** ärver *alla variabler och metoder* från sin **superklass**. Vi kan sedan lägga till extra variabler och metoder.

- Alla klasser i Java ärver implicit från klassen `Object`. Javas klasser bildar en hierarki.

```
class Atlet extends Person {
   // utöver de instansvariabler vi ärver från Person
  private double syreupptagning;
  ...
}
```

Klassen Atlet äver alltså alla instansvariabler som fanns i klassen Person (namn, längd, vikt, ålder) och lägger till ytterligare en: syreupptag. Atlet ärver också metoden bmi() från Person.

Eftersom alla klasser implicit ärver från Object, så “innehåller” Person även metoder från klassen Object (toString etc). Även dessa ärvs (indirekt) av Persons subklass Atlet.

Överrida metoder

- Ibland gör inte metoder vi ärvt från en superklass det vi önskar.
- Vi kan då skriva en ny metod med samma namn som ***överlagrar (overrides)*** den ärvda metoden i subklassen.
- Det är t.ex. god programmeringspraktik att överlagra `toString()` och vissa andra metoder som ärvs från `Object`.

```
Person p1 = new Person("Alice", "Alicedotter", 60, 1.60, 60.5);
System.out.println(p1.toString());
```

```
$ Person@511baa65
```


Om vi försöker “skriva ut” ett objekt med System.out.println så kallas metoden toString, som ärvts av superklassen Object. Den skriver ut objektets s.k. hash-kod, vilker förmodligen inte var vad vi förväntade oss!

I Haskell kan man ju ofta skriva t.ex. “deriving show” efter en ny datatyp, och få en vetting funktion för att skriva ut en representation för datatypen. I Java ärver man bara toString-metoden från Objekt, och den gör inte alltid det vi förväntar oss.

Det är god programmeringspraktik att överlagra fler av metoderna som ärvs från Object, med mer specifika versioner. Se Javas API för klassen Object.

Övning: Skriv en toString() metod för klassen Person, som returnerar en enligt dig lämpligare strängrepresentation. Skriv även en metod equals(Person other) som överlagrar equals-metoden som ärvts från Object.

Notering: En metod som är deklarerad `final` i en klass kan inte överlagras av en subklass.

Repetition: Interfaces

- Specificerar enbart vilka metoder som ska finnas (headers). Påminner of type-classes i Haskell.
- Klasser kan *implementera* ett eller flera interface.

```
interface Runnable{
  // Exempel-interface från Javas bibliotek för multithreading
 void run()
 }
```

```
class GUIStart implements Runnable{
 ...
 void run(){...}//kod för att sätta upp och köra GUI-applikation.
```

Interfaces innehåller bara signaturer för metoder (deras namn, returtyp och argumenttyp). Metoderna definieras inte och inga klassvariabler anges. Ett interface specificerar alltså bara gränssnittet, inte hur representationen av objekten ska implementeras. För att kunna använda ett gränssnitt måste det finnas en klass som implementerar det.

Runnable är ett interface med en metod (run). När en objekt av en klass some implementerar Runnable startar en ny tråd (för parallell exekvering) körs metoden run.

Abstrakta klasser

- Innehåller en **blandning** av vanliga **konkreta metoder** och **abstrakta metode**r som saknar implementation.
- Mellanting mellan interfaces och vanliga klasser.
- Går ej att skapa objekt av abstrakt klass.
- (Konkreta) subklasser är tänka att implementera metoderna.

```
//JavaFX applikationer ärver från denna klass.
public abstract class Application{
 abstract void start(Stage primaryStage);
 . . .
 }
```

```
public class HelloWorldFX extends Application {
  // Måste här ge implementation för metoden start.
   public void start(Stage primaryStage){
      // Kod för att starta applikationen }
}
```

Abstrakta klasser kan innehålla dels vanliga metoder, men även abstrakta metoder, dvs enbart metodens signatur (som i interfaces). Abstrakta klasser kan alltså sägas vara ett mellanting mellan ett Interface och en klass. En klass måste deklareras abstract ifall den innehåller någon abstrakt metod.

Man kan inte skapa objekt av abstrakta klasser. Däremot kan man ärva från dem. Då måste subklassen ge en implementation av de abstrakta klasserna i föräldraklassen.

Exemplet kommer från biblioteket JavaFX, som används för att skapa GUI-applikationer. Vi kommer se lite mer av det senare.

Samanfattning: klasser, interface, abstrakta klasser

- En vanlig (konkret) klass innehåller variabler och metoder.
- Ett **interface** innehåller enbart **signaturer för metoder** (abstrakta metoder). Inga implementationer.
  - En klass kan **implementera ett interface**. Den måste då innehålla implementationer av interfacets metod-signaturer.
- En **abstrakt klass** kan innehålla **både konkreta metoder och signaturer** för metoder som saknar implementation (abstrakta metoder).
  - En klass kan **ärva (extends) en abstrakt klass**, och måste då implementera de abstrakta metoderna.

Demo: Interface och Klass

- Anta att vi har fått **ett interface `IntList` som specificerar en lista av heltal**.
- Här väljer vi att implementera detta som en **dynamisk array**.
  - En array som kan vara större än aktuellt antal element i listan.
  - Förstorar arrayen vid behov.
  - Från början allokeras en array med ett visst antal element. Arrayens verkliga storlek brukar kallas kapacitet.

```
public interface IntList {
  void add(int e);             // lägg till element e på slutet
  void add(int index, int e); // sätt in element e på plats index
  void remove(int index);      // ta bort element på plats index
  void clear();                // töm listan
  int get(int index);          // läs element på plats index
  void set(int index, int e); // uppdatera element på plats index
till e
  int size();                  // tala om listans längd
}
```

Det finns många olika sätt vi skulle kunna implementerar en lista. Här väljer vi en dynamisk array (en ganska vanlig datastruktur). Vi implementerar därefter de givna metoderna.

Ett interface säger inget om vilka konstruerare som ska finnas i en klass som implementerar det.

DEMO: DynArray.java

Grafiska gränssnitt (GUI)

- Flera färdiga paket för GUIs i Java:
  - awt (äldst)
  - **Swing** (färdiga komponenter, liknande utseende över olika platformar)
  - JavaFX (nyast)

Det finns olika paket i Java:s API för att skriva program med grafiska gränssnitt. Det ursprungliga var awt, sedan kom Swing med fler färdiga komponenter och ett utseende som var mer likt mellan olika plattformar. Den senaste varianten heter javafx och den har funnits några år. Den har vissa fördelar framför de äldre paketen när det gäller hårdvaruacceleration, användar-input på touchskärmar, animationer, 3D-grafik, html-innehåll.

Eftersom det är ett modernare och mer kraftfullt system så är också abstraktionsnivån högre. I kursen fokuserar vi på Swing för att det är så etablerat och koncepten är konkreta. När man kan Swing är det lätt att gå vidare och lära sig javafx. Även om javafx används för mycket nyutveckling så finns många applikationer som använder Swing kvar, så därför är det nyttigt att känna till båda två. I lab 3 kan man välja om man vill använda Swing eller JavaFX. I det senare fallet får man dock söka mer information själv samt anpassa den givna test-koden.

Händelsestyrda program

- **Initieringsfas**: bygger GUI
- **Huvudfas**: Väntar på händelser.
  - Eg. musklick
- Både Swing och JavaFX består av hierarki av klasser, som representerar olika komponenter.
  - Fönster, knappar, textrutor....
  - Hierarkin av komponenter i Swing finns här. Exempel:
    - Basklass i Swing: JComponent
    - JFrame: Fönster med ram
    - JButton: En knapp

Metodiken när man skriver program med GUI är densamma mellan javafx och Swing (och motsvarande system för andra programspråk). För det första så är interaktionen med användaren händelsestyrd, till skillnad från att vara programstyrd som den är i applikationer som interagerar via kommandoprompten Att den är händelsestyrd innebär att programmet har en initieringsfas där användargränssnittet byggs upp och att det sedan går in huvudfasen där den väntar på att händelser ska uppstå, ofta genom att användaren ger någon input, t.ex. klickar med musen.

Ett annat gemensamt drag är att kärnan i paketen består av en hierarki av klasser som representerar olika grafiska komponenter, t.ex. ett fönster med ram eller en knapp.

Swing bygger på awt så man behöver använda klasser även från detta paket här och var.

Trådar och GUI-program (swing)

- Trådar: för parallell exekvering.
- Javaprogram startar genom att skapa en tråd som kör main-metoden.
- Avslutar när alla trådar är klara (inte bara main).
- I Swing: *event dispatching thread* väntar på händelser e.g. (knapptryck). Här bör även initiering av GUI ske.

I java-program kan koden, liksom i andra moderna programspråk såsom Haskell och C#, köras i olika trådar (threads). Om ett program har flera trådar igång så exekveras dessa ofta faktiskt eller virtuellt parallellt. När ett javaprogram börjar exekveras skapas en tråd som kör main-metoden. Ett java-program fortsätter exekveras tills alla trådar avslutats. Så även om huvudtråden blir klar med main-metoden så avslutas inte programmet om det skapats andra trådar som fortfarande är aktiva.

Trådar kan också användas för att utföra olika typer av uppgifter. I Swing och awt används en event dispatching thread som är den som väntar på händelser och utför det ska hända när en sådan uppstår. Det rekommenderas att man också utför initieringen av GUIt i denna tråd. Standardsättet att starta denna tråd är att anropa SwingUtilities.invokeLater. Detta innebär att event dispatching thread startas när huvudtråden är klar. Som argument tar denna metod en klass som implementerar interfacet Runnable. Detta deklarerar en metod
void run()
som körs i event dispatching thread. I denna metod (eller i hjälpmetoder) utför man initieringen av GUIt.

Trådar och GUI-program (swing)

```
import java.awt.*;
import javax.swing.*;

public class GUIStart implements Runnable {
    public void run() {
      JFrame frame = new JFrame("GUIStart");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // create frame contents here

      frame.pack();
      frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new GUIStart());
    }
}
```

Standardsättet att starta Event Dispatching Thread denna tråd är att anropa SwingUtilities.invokeLater. Detta innebär att event dispatching thread startas när huvudtråden är klar. Som argument tar denna metod en klass som implementerar interfacet Runnable. Detta deklarerar en metod
void run()
som körs i event dispatching thread. I denna metod (eller i hjälpmetoder) utför man initieringen av GUIt.

Här har vi valt att låta huvudklassen implementera Runnable. Det är vanligt men inte nödvändigt. Det kan göras av en hjälpklass istället.
Metoden setDefaultCloseOperation bestämmer vad som ska ske man använder stänger fönstret. Alternativet EXIT_ON_CLOSE innebär helt enkelt att hela programmet avslutas.

DEMO: GUIStart.java

Lite Innehåll: JLabel och JButton

```
import java.awt.*;
import javax.swing.*;

public class GUIStart implements Runnable {
    public void run() {
      JFrame frame = new JFrame("GUIStart");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JLabel label = new JLabel("JLabel");
      frame.add(label);

      JButton button = new JButton("JButton");
      frame.add(button);

      frame.pack();
      frame.setVisible(true);
    }
```

Vi kan lägga in en knapp i ramen så här:

```
    JButton button = new JButton("JButton");

button = new JButton("Count");
frame.add(button);
```

En JFrame har en ram och titel. Själva ytan som innehållet befinner sig i kan man nå genom
frame.getContentPane()
Förr var man tvungen att explicit lägga till komponenter till denna, men nu kan man anropa add direkt för ramen.

JLabel används för att visa en kort text. Den har en konstruktor JLabel(String s) som väljer vad det ska stå och en metod setText(String s) som man senare kan ändra texten med.

Layout

```
JButton button = new JButton(“JButton");
frame.add(button, BorderLayout.SOUTH);
```

```
frame.setLayout(new FlowLayout()); // från vänster -> höger
```

```
//horisontellt eller vertikalt
frame.setLayout(new BoxLayout(frame.getContentPane(),
BoxLayout.Y_AXIS));
```

Filosofin hos moderna system för GUI är att i möjligaste mån låta den som skriver programmet att uttrycka layout utan att ange faktiska positioner och storlekar i pixlar. Man gör detta genom att istället t.ex. säga att en viss komponent befinner sig ovanför eller till vänster om en annan.

Det finns några olika layout-modeller i java. Förvald är BoarderLayout. Den består av fem delar, en central och fyra stycken vid sidan om som bildar en ram. add(c) gör c till den centrala komponenten och den överlagrade varianten add(c, BorderLayout.SOUTH) (NORTH, WEST, EAST) ställer in komponenterna vid sidan om.

En annan är FlowLayout där man med add(c) kan lägga till godtyckligt många komponenter som placerar sig från vänster till höger, uppifrån och ner.
frame.setLayout(new FlowLayout());

En tredje är BoxLayout där man anger om komponenterna ska placeras horisontellt eller vertikalt efter varandra.
frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));

https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Lyssnare

- Metoder som blir anropade när något händer
  - tryckt på en knapp i GUIt (med musen).
  - flyttat musen, minimierat fönstret, tryckt på tangent...
- I Swing: Implementera en `ActionListener`. Ofta som en s.k. inre eller lokal klass.

```
private class ButtonActionListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
      System.out.println("Hello World!");
      }
   }
```

```
button.addActionListener(new ButtonActionListener());
```

Ett centralt begrepp i ett händelsestyrt användargränssnitt är lyssnare (listeners). De kan man se som metoder (som implementeras av någon klass) som blir anropade då en händelse har inträffat. Det kan vara att användaren har tryckt på en knapp, klickat eller flyttat på musen, minimerat fönstret, tryck på tangentbordet etc.

Många händelser gör man lyssnare till genom att implementera en klass (oftast görs detta som en s.k. privat inre klass, inuti en annan klass där den ska användas) som implementerar interfacet ActionListener och dess metod:

```
void actionPerformed(ActionEvent e)
```

Ett ActionEvent är en komponent som innehåller information bl.a. om för vilken komponent händelsen inträffade och vilken typ av händelse det är.

För att lägga till en lyssnare (det kan finnas flera) använder main addActionListener(ActionListener l) för den komponent det gäller.

Man kan skapa en ActionListener för varje komponent (genom t.ex. lokala klasser), men man kan också ta hand om flera komponenters händelser på samma ställe genom att anropa getSource() och på så sätt avgöra vilken komponent det gäller.

DEMO: SwingHelloWorld.java