Objektorienterad Programkonstruktion

Föreläsning 3

7 nov 2016

Klass/instans

Med hjälp av nyckelordet **static** kan vi bestämma att en metod eller ett fält ska tillhöra själva klassen i stället för en specifik instans av klassen.

Klassvariabler och klassmetoder laddas in tillsammans med klassen när man startar JVM, och kräver inte att några objekt har instansierats.

När bör man använda klassmetoder/-fält?

- Egenskaper som är gemensamma för alla objekt
  - ex: räknare för antalet instanser
- Konstanter (kan deklareras static final för att förhindra att de ändras av misstag)
  - ex: `Math.PI, Math.E`
- Metoder som utför operationer med primitiva variabler
  - ex: `Math.cos(double a), Math.round(double a)`
- Metoder som fungerar oberoende av om objekt av en viss klass har skapats
  - ex: `Map.convertKmToMiles(double km)`

Klass eller instans?

Med hjälp av nyckelordet static kan vi bestämma att en metod eller ett fält ska tillhöra själva klassen i stället för en specifik instans av klassen (ett objekt)

- Instansmetoder kan komma åt instansvariabler och -metoder direkt
- Instansmetoder kan komma åt klassvariabler och -metoder direkt
- Klassmetoder kan komma åt klassvariabler och -metoder direkt

men

- Klassmetoder kan inte komma åt instansvariabler och -metoder direkt

Klass/instans, exempel

```
public class Exempel{

    private int a = 5;
    static int b = 3;

    public int getA(){
        return a;
    }

    public int getB(){
        return b;
    }

    public static int getStatB(){
        return b;
    }
}
```

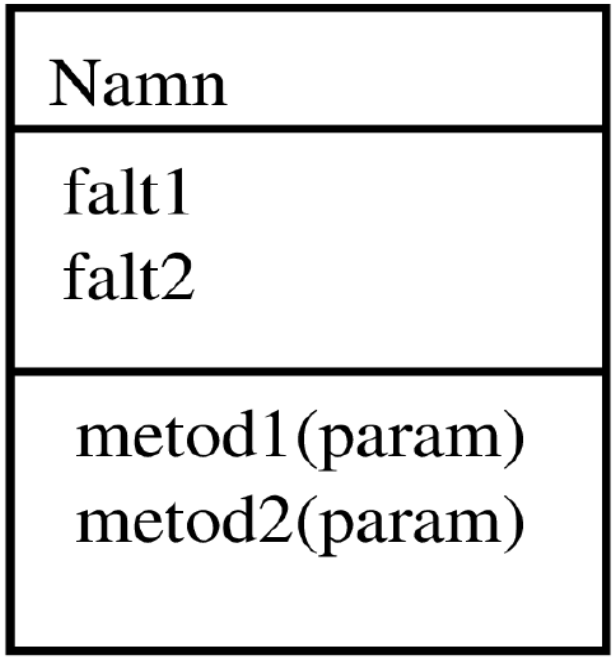
```
public class FelaktigtExempel{

    // OBS! denna kod är
    // felaktig !!
    private int a = 5;
    static int b = 3;

    public static int getA(){
        return a;
    }
}
```

non-static variable a cannot be referenced from a static context

UML: Klassdiagram



UML: Relationer

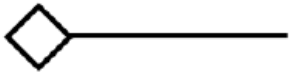
Ärver från superklass

Implementerar gränssnitt

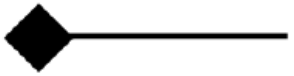
Dubbelriktad eller oriktad relation

Riktad relation (har, känner till)

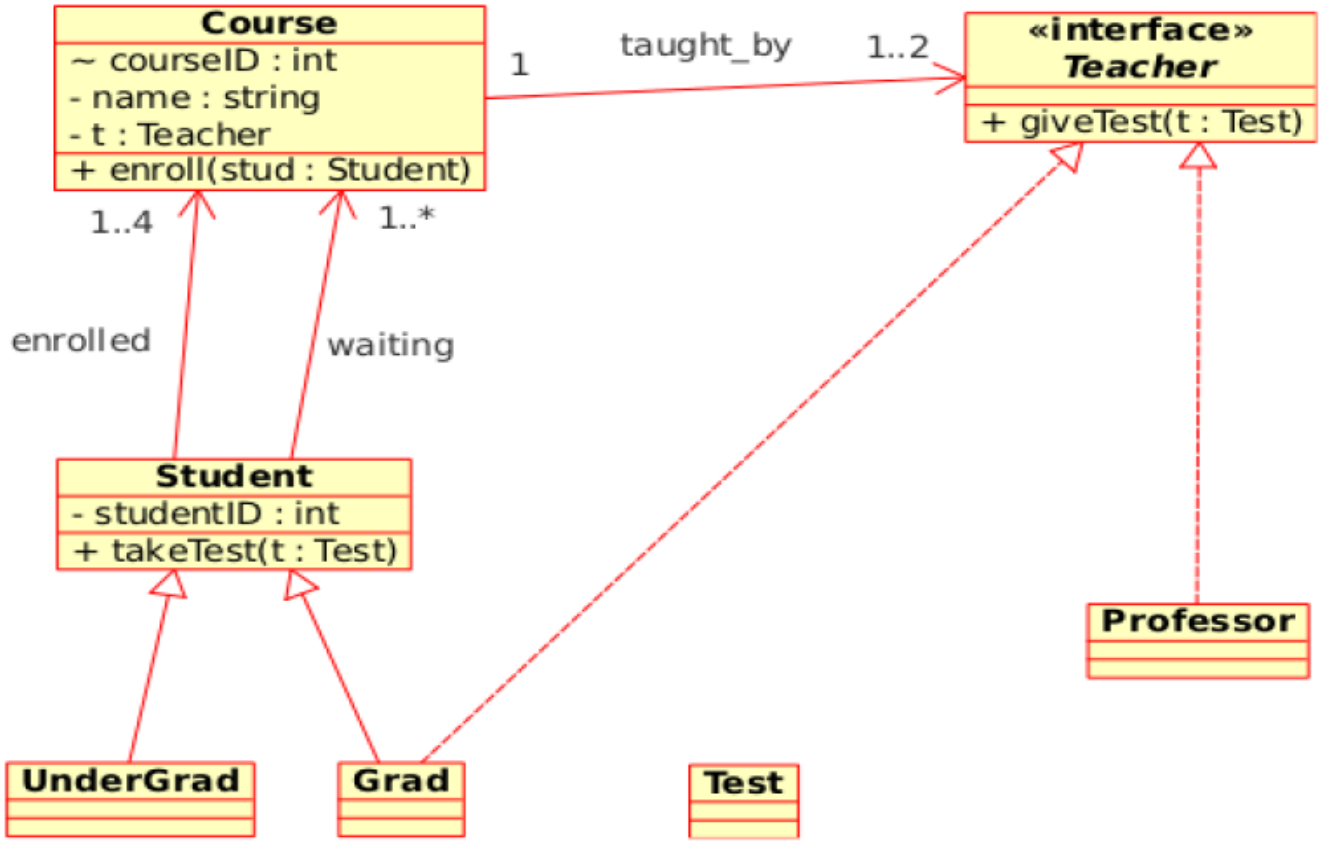
Aggregat (har, består av)



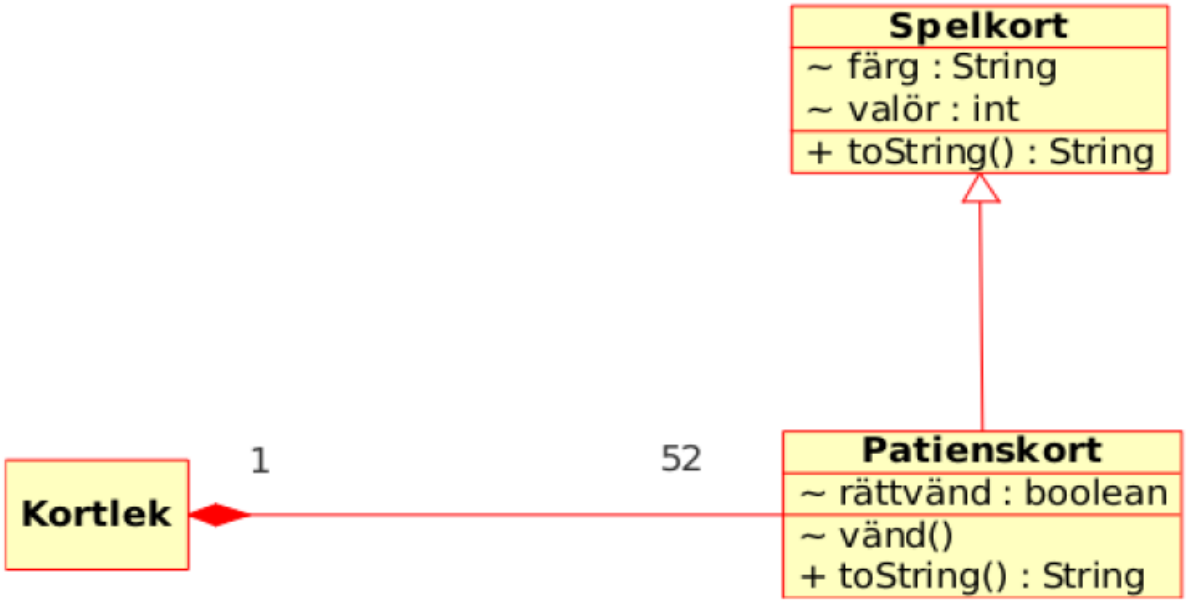
Komposition (-"-, äger)



UML: Relationsexempel



UML: Relationsexempel 2




```
class Kortlek {
   Patienskort[] kortlek = new PatiensKort[52];
      ...
}
```

Grafik

Java har ett antal olika abstraktionsnivåer för att arbeta med grafik:

Graphics - lågnivå, för att skapa bilder, figurer, effekter

AWT – (Abstract Window Toolkit) Platformsspecifik implementation av fönster och GUI-komponenter

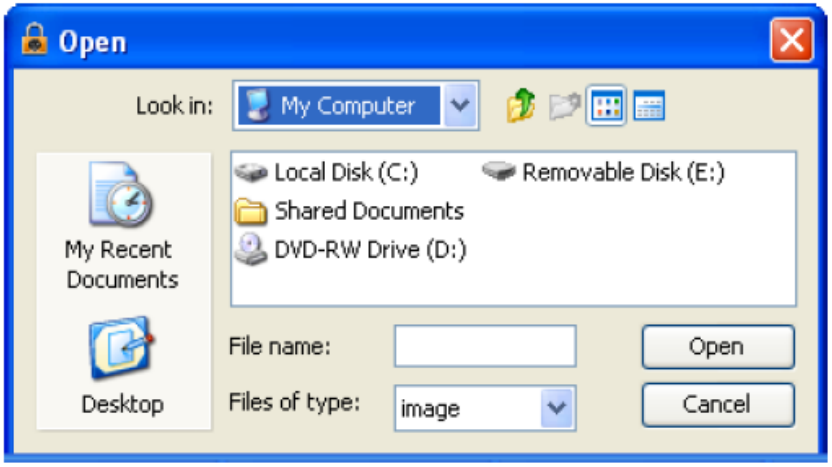
Swing - Plattformsoberoende högnivåversion av GUI-komponenter

I denna kurs kommer vi främst att använda oss av Swinggränssnittet, men för vissa delar kommer vi att behöva arbeta direkt med de två andra

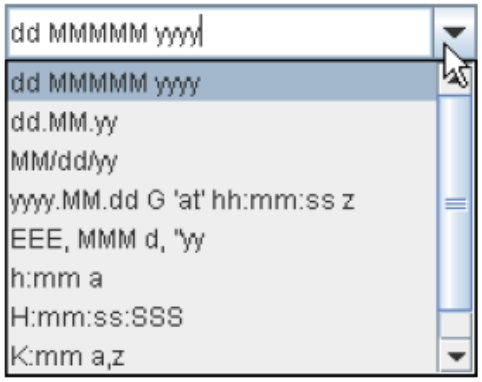
Swing

- Använder utseende och "känsla" från det omgivande systemet
- Fonter, fönsterramar och annat plockas t.ex från systemet
- På Mac kan man t.ex välja att låta lägga menyerna högst upp på skrivbordet i stället för högst upp i fönstret.
- Det går även att välja ett "java-specifikt" utseende
- Det kan gå väldigt fort och enkelt att skapa ett GUI, eftersom alla vanliga beståndsdelar redan finns som färdiga mallar

Exempel på Swing-komponenter

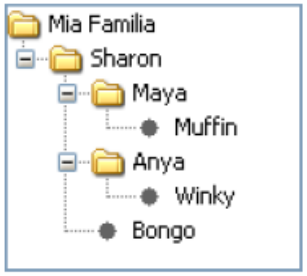


JFileChooser

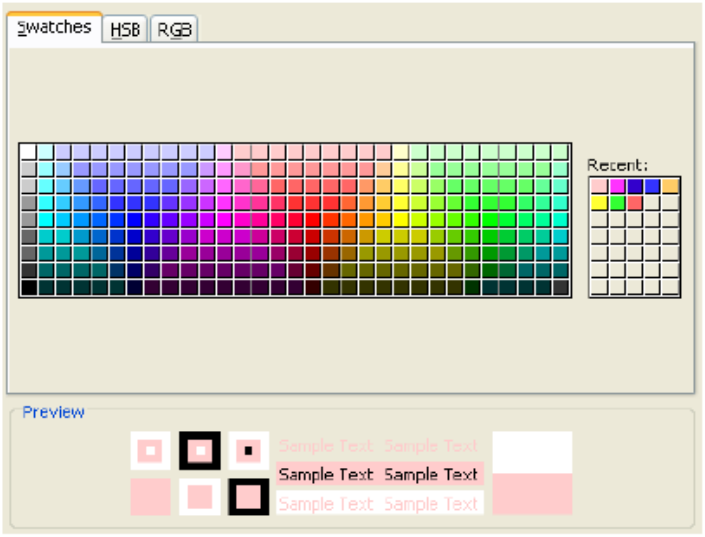


JComboBox

JProgressBar



JTree

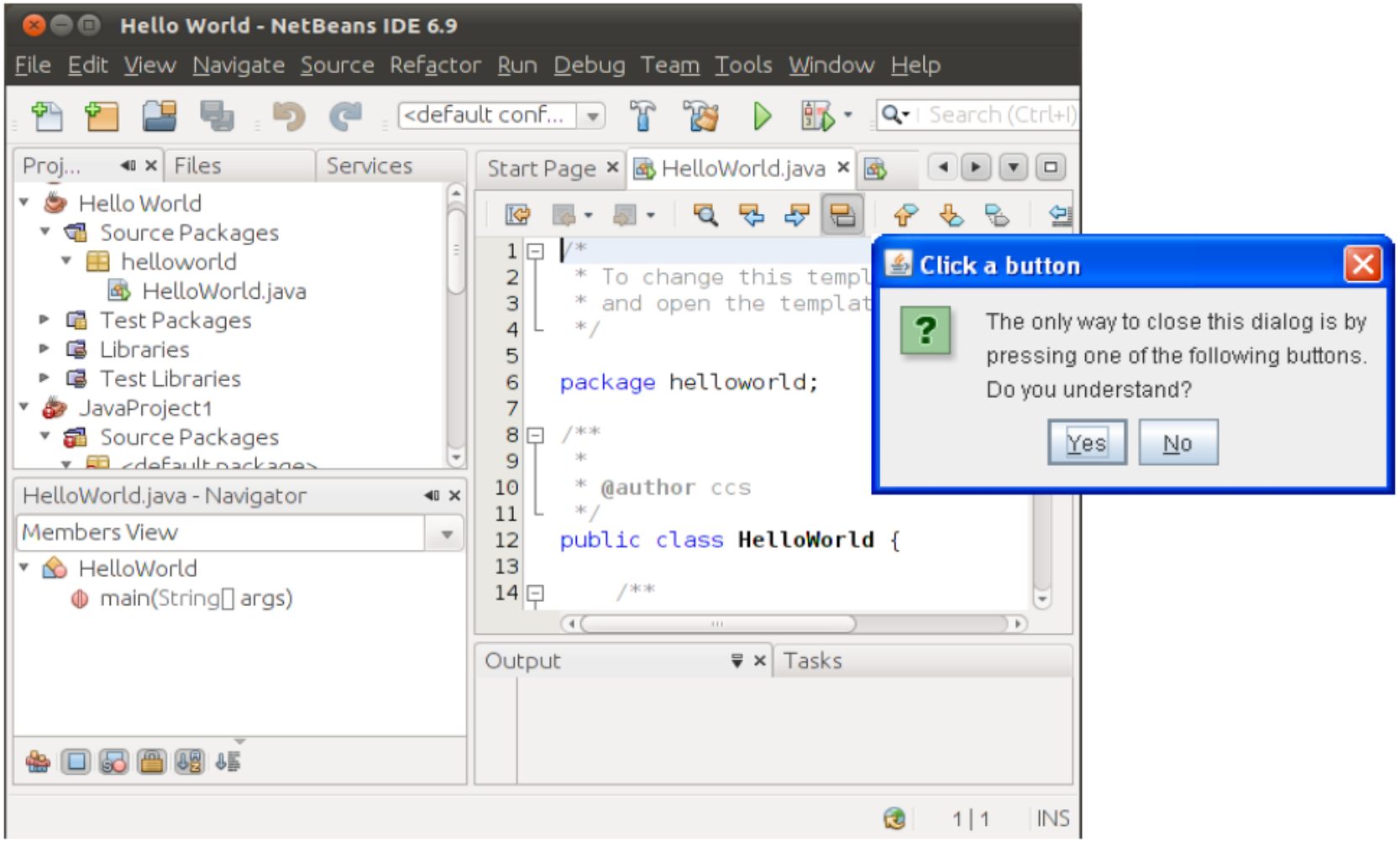


JColorChooser

Grafikhierarkier i Swing

- I Java skapas grafik som trädformade hierarkier
- I roten på varje träd finns en top-level container, som kan vara någon av:
  - **JFrame** - ett fönster på skrivbordet som kan fyllas med godtyckligt innehåll
  - **JDialog** - ett litet fönster som är kopplat till en applikation. Vanligaste subklassen är **JOptionPane**
  - **JApplet** - ett grafiskt fönster som bäddas in i ett annat program, vanligtvis på en hemsida som visas i en webläsare

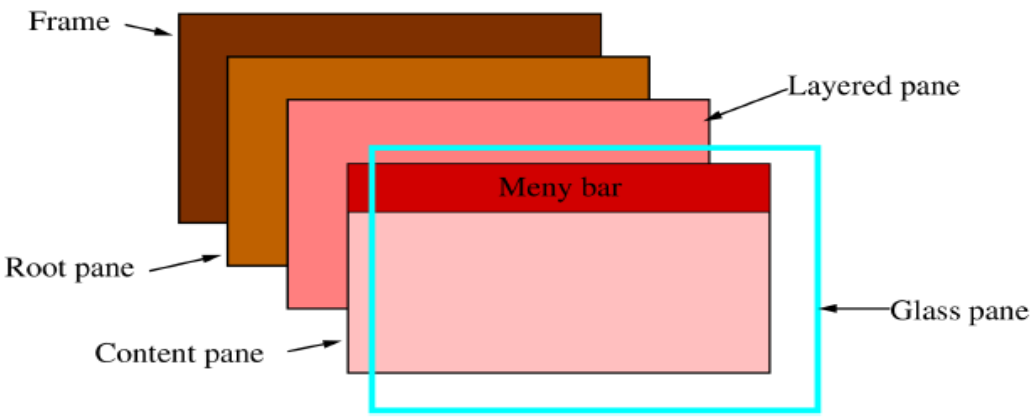
JFrame vs JDialog



Grafikhierarkier I Swing

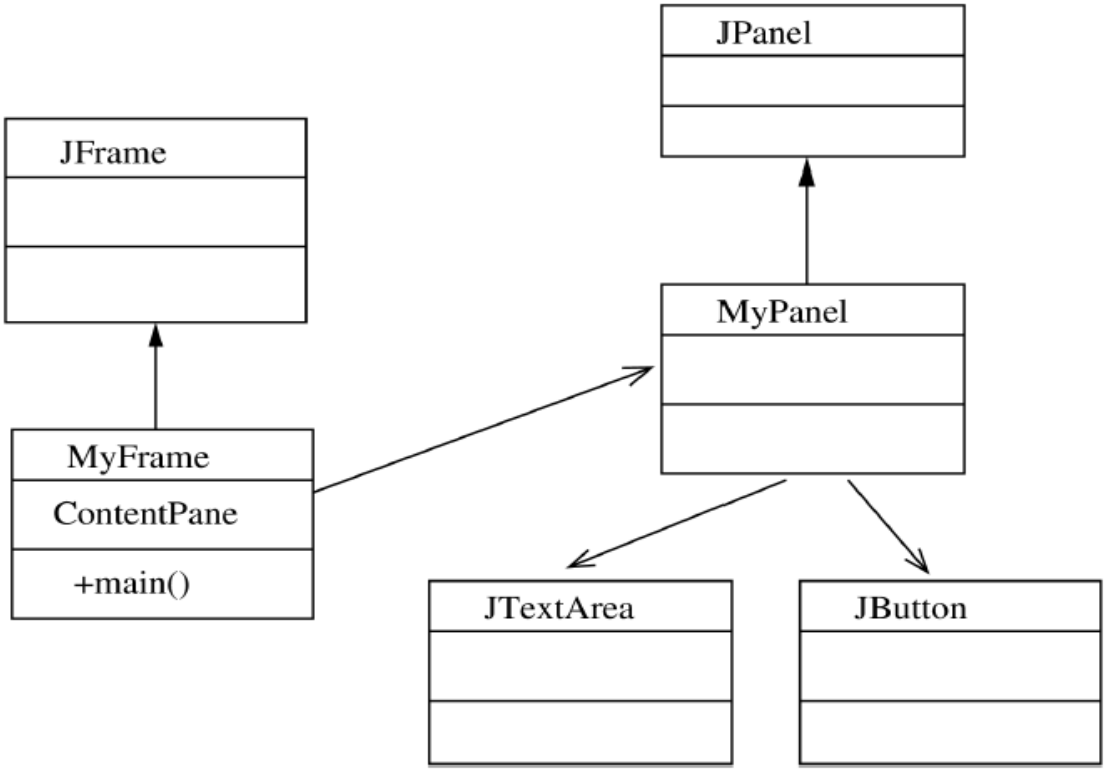
En top-level container innehåller:

- **root pane**, som innehåller
  - **layered pane**, som innehåller
    - (**menu**)
    - **content pane**, som kan innehålla
      - Diverse komponenter
- **glass pane**, som ligger utanpå andra komponenter



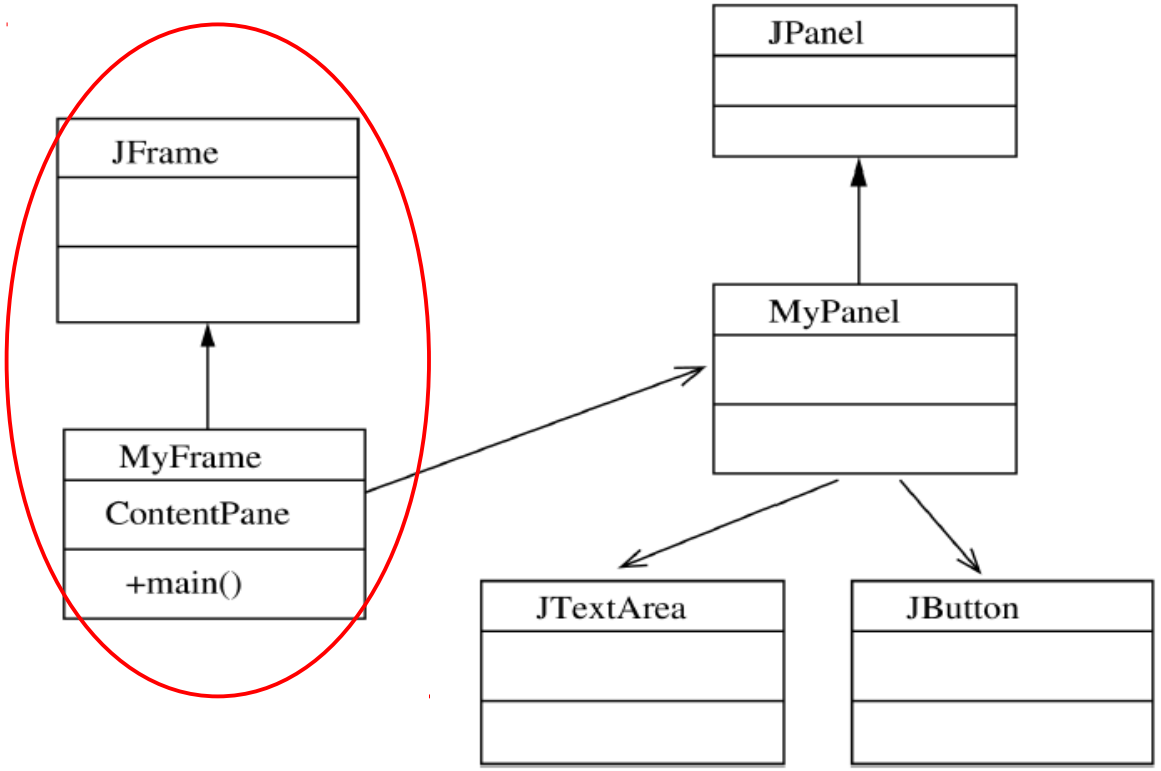
Grafikhierarkier (Swing)

- Som användare behöver man i många fall bara ha koll på sin top-level container och det innehåll man lägger i dess content pane.



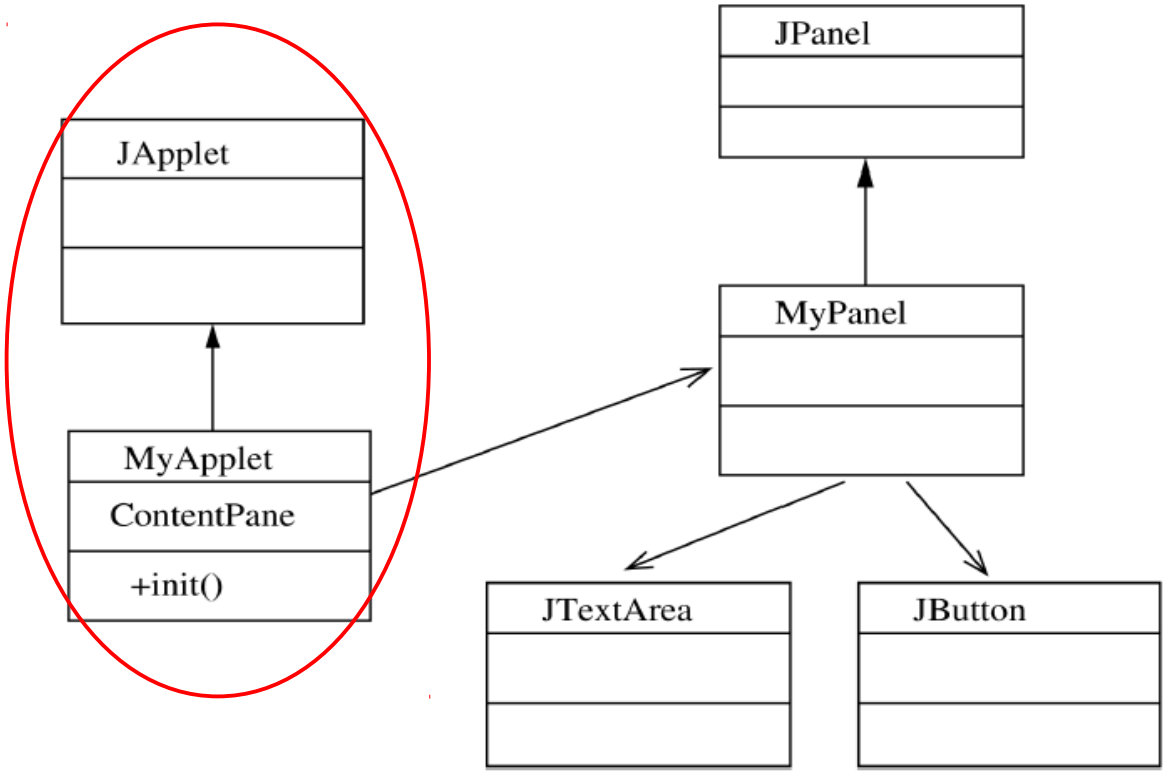
Grafikhierarkier (Swing)

- Som användare behöver man i många fall bara ha koll på sin top-level container och det innehåll man lägger i dess content pane.



Grafikhierarkier (Swing)

- Som användare behöver man i många fall bara ha koll på sin top-level container och det innehåll man lägger i dess content pane.



Minimalt Swing-exempel

```
import javax.swing.*;
public class GrafikExempel extends JFrame{
    public GrafikExempel(){
        JLabel myTextLabel = new JLabel("Hello World");
        add(myTextLabel);
        pack();
        setVisible(true);
    }

    public static void main(String[] args){
        GrafikExempel myGExempel = new GrafikExempel();
    }
}
```

Applets

- **Applets** är program som kan köras som en del av ett annat program, typiskt en webläsare.
- För att fungera måste en en applet ärva från klassen `java.applet.Applet`, vilket t.ex **JApplet** gör
- En applet har ingen `main(String[] args)`-metod, utan motsvarande funktion fylls istället av `init()`
- För att kunna köras i en webbläsare (eller i appletviewer) måste en applet bäddas in i html:

```
<applet code = MyApplet.class
 width = 200 height = 100></applet>
```

Minimalt Appletexempel

```
import javax.swing.*;
public class HelloApplet extends JApplet{
    public void init(){
        JLabel myTextLabel =
            new JLabel("Hello World!");
        add(myTextLabel);
        setVisible(true);
    }
}
```

Interaktiva komponenter

- I ett användargränssnitt vill man att användaren ska kunna göra olika saker - trycka på knappar, skriva in text eller flytta på skjutreglage - som programmet ska reagera på
- I Swing hittar vi t.ex
  - **JButton** - en knapp man kan trycka på
  - **JRadioButton** - knappar för att välja ett alternativ
  - **JCheckBox** - knappar för att välja flera alternativ
  - **JSlider** - ett skjutreglage för att välja ett numeriskt värde
  - **JTextField** - ett fält där man kan skriva in en textrad
  - **JPasswordField** - ett fält som döljer vad man skriver in
  - **JTextArea** - ett fält där man kan skriva längre texter

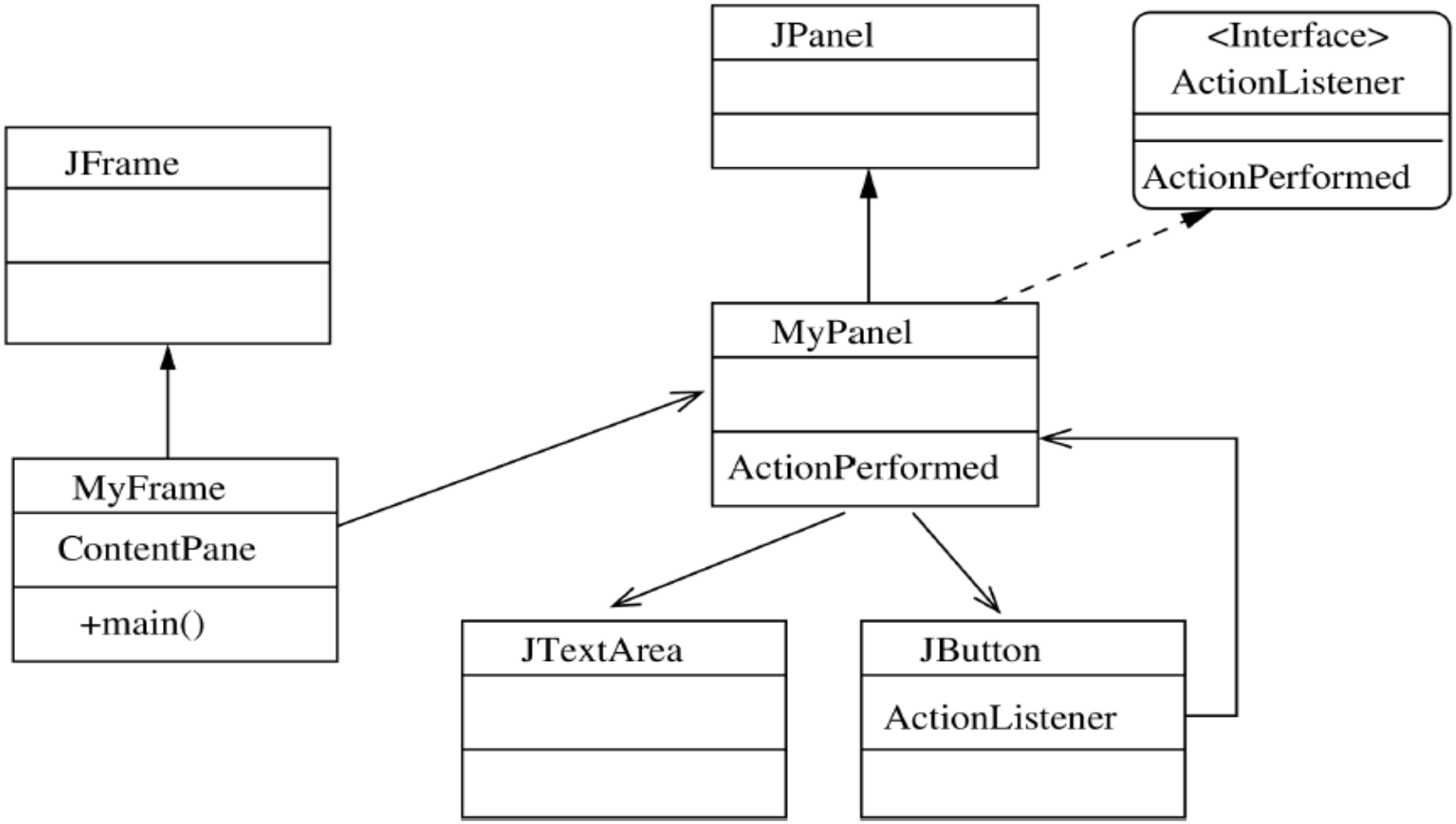
Lyssnare

- För att få ett program att reagera på en interaktiv komponent behövs det en lyssnare
- Den interaktiva komponenten genererar en händelse (event) som gör att en viss metod i lyssnaren anropas
- Klassen som lyssnar måste implementera ett lyssnargränssnitt, vilket varierar med typen av interaktiv komponent.
- Ett objekt kan lyssna på flera andra objekt
- Flera objekt kan lyssna på samma objekt

Lyssnare

- Exempel:
  - en lyssnare till en **JButton** måste implementera gränssnittet `java.awt.event.ActionListener`, och därmed tillhandahålla en metod som heter `actionPerformed(ActionEvent e)` och som anropas när man trycker på knappen
  - `e` ovan innehåller information om vilket objekt som skickade det
- Det finns många olika möjlighter för att bestämma vilket objekt som skall lyssna på ett annat, beslutet bör baseras på vad som ska hända när man trycker på knappen. T.ex kan det vara vettigt att låta det objekt som ska påverkas av knapptryckningen lyssna, eller att ha ett centralt debug-objekt som lysnar på alla knappar och skriver ut någon text på skärmen

Grafikhierarkier med lyssnare



Kodexempel Lyssnare

```
public class MyListener implements ActionListener{
    public void actionPerformed(ActionEvent e){
        System.out.println("Du tryckte på knappen!");
    }
}

public class MyFrame extends JFrame{
    public MyFrame(){
        JButton myButton = new JButton("Tryck mig!");
        myButton.addActionListener(new MyListener());
        ... // add(), pack(), etc
    }
    public static void main(String[] args){...}
}
```

•

Javas API-dokumentation

- För att få specifik information om hur befintliga klasser i Java fungerar, kan man läsa API-dokumentationen
- För alla klasser finns här information om vilka paket de finns i, vilka gränssnitt de implementerar, vilka fält de har, vilka metoder de har och en kort beskrivning av hur man använder dem.
- Se:

```
http://download.oracle.com/javase/7/docs/api/
```

Javadoc

- Med hjälp av **JavaDoc** kan man automatiskt skapa egen API-dokumentation
- Om vi skriver beskrivande kommentarer som kommer precis före en fält- eller metoddeklaration och som inleds med `/**` kommer detta att generera en förklarande text I dokumentationen.
- Från kommandoraden kan man anropa `javadoc *.java` för att skapa dokumentation för samtliga java-filer
- I NetBeans kan man högerklicka på ett kommando man har skrivit och välja "show JavaDoc" för att få upp ett fönster med dokumentation