Objektorienterad Programmering DAT043

Föreläsning 6
30/1 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Repetion: En GUI-applikation med Swing

- Huvudklass implementerar `Runnable` - metoden `run()`.
- main-metoden kallar `SwingUtilities.invokeLater`
- Trådsäkerhet - parallell exekvering.

```
public class MyGUI implements Runnable {
    public void run() {
     // Kod för att sätta upp GUI
    }

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


När vi implementerar en GUI-applikation med Swing bör vi först se till att den är trådsäker -dvs fungerar i miljöer där man använder parallell exekvering, vilket är vanligt i moderna program och system. Vår huvudklass ska därför implementera interfacet Runnable, och dess metoden run(). I main-metoden kan vi sedan kalla metoden SwingUtilities.invokeLater på ett objekt i vår GUI-klass. Denna metod ser till att GUI:t sätts upp i en lämplig tråd (event dispatching thread).

Repetion: En GUI-applikation med Swing

- Nästa steg är att bestämma vad som ska vara i GUI:t
- Vi utgår från en JFrame (representerar ett fönster)
- Därefter lägger vi till knappar, textrutor, osv.

```
JFrame frame = new JFrame("Hello World!");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JButton button = new JButton("Say 'Hello World'");

frame.setPreferredSize(new Dimension(300, 250));
frame.add(button, BorderLayout.SOUTH);

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


Nu kan vi fylla run-metoden med kod för att sätta upp vårt GUI. Vi börjar med att skapa en JFrame, vilket är ett objekt som representerar ett fönster. Vi kan ange önskade dimensioner på en JFrame, ange önskad layout osv (se Javas API).

Vi kan lägga till komponenter (här en knapp) på vår JFrame, med metoden add. Här anger vi även en Layout för var vi vill att knappen ska vara (Längst ner i fönstret).

Metoden pack() justerar storleken på fönstret så att komponenterna i möjligaste mån får sin önskade storlek. Vi kan ange en önskad storlek på fönstret med setPreferredSize(), vilket pack då försöker ta hänsyn till (om det går, beroende på andra komponenters önskade storlek).

Avslutningsvis måste vi göra vårt fönster synligt, så det är viktigt att inte glömma sista raden och metoden setVisible(true).

Repetion: En GUI-applikation med Swing

- För att något ska hända när vi t.ex. trycker på en knapp måste vi lägga till en s.k. *Lyssnare*.
- Implementeras ofta som en privat, inre klass.
- Lyssnare implementerar ett interface `ActionListener`, och metoden `actionPerformed`.

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

```
JButton button = new JButton("Say 'Hello World'");
button.addActionListener(new ButtonActionListener());
```


Nästa steg är att få något att hända när vi trycker på knappen. Vi måste då lägga till en s.k. lyssnare till knappen. Det är helt enkelt ett objekt av typen ButtonActionListener. Klassen ButtonActionListener kommer bara användas internt i vårt GUI, så vi kan definiera den som en s.k. privat, inre klass inuti huvudklassen.

Det finns även andra typer av lyssnare för andra händelser (t.ex. musrörelser, keybord tryck osv). Konsultera Javas API!

Fler Swing-komponenter

```
JPanel panel2 = new JPanel();
panel2.setLayout(new BoxLayout(panel2, BoxLayout.Y_AXIS));
frame.add(panel2);
```

```
checkBox = new JCheckBox("CheckBox");
frame.add(checkBox);
...
if (checkBox.isSelected()){...}
```

```
textField = new JTextField(10); // Antal tecken
frame.add(textField);
...
String s = textField.getText();
```

Här ger vi exempel på ytterligare några komponenter som vi kan lägga till på vår JFrame från förra exemplet.

JPanel: Inget eget innehåll, men vi kan fästa andra komponenter på den. Man kan ställa in en layoutmodell för JPanel som skiljer sig från huvudfönstrets. Med JPanel kan man skapa en hierarki av grafiska element där olika delar är organiserade enligt olika layoutmodeller.

JCheckBox: är en kryssruta som antingen kan vara vald eller inte. Metoden isSelected() används för att avgöra dess status.

JTextField: är en komponent för inmatning av enradig text. En ActionListener till en sådan komponent anropas då användaren har tryckt på enter. getText() ger aktuellt innehåll i rutan.

DEMO: GUIDemo.java

Egna komponenter och enkel grafik

```
public class OlympicRingComponent extends JPanel{

    //A wider pen stroke for drawing the rings.
    final static BasicStroke wideStroke = new BasicStroke(5.0f);

    public void paintComponent(Graphics g0){
        Graphics2D g = (Graphics2D) g0;

    //Draw with a wider pen.
        g.setStroke(wideStroke);
        int diam = 50; // Ring diameter

        // Draw first ring.
        Ellipse2D.Double ring1 =
                  new Ellipse2D.Double(50,50,diam,diam);
        g.setColor(Color.BLUE);
        g.draw(ring1);
       .......
```

Genom att ärva existerade komponenter kan man modifiera dem, t.ex. förändra utseendet. Vill man rita grafik, t.ex. diagram eller bilder så är det bra att utgå från JPanel eftersom man vill bestämma innehållet helt själv.

Alla komponenter har en metod som ritar innehållet varje gång detta behövs, paintComponent. Denna tar som argument ett objekt av typen Graphics som är en miljö för komponentens yta som tillåter grafiska operationer.
Man överrider (definierar en ny) uppritningsmetod

```
    @Override
    public void paintComponent(Graphics g) {
...
}
```

@Override är inte nödvändigt men rekommenderat för att göra det tydligt att man ersätter en metod i någon superklass. Mer om detta vid ett annat tillfälle. Argumenttypen är Graphics, men det är “egentligen” fråga om ett objekt av typen Graphics2D (som har något fler metoder). Alltså är det säkert att göra en typecast till Graphics2D i det här fallet.

I Graphics finns metoder för att rita linjer (drawLine), cirklar och andra figurer, bilder och text.
Om lyssnare till händelser ändrar på något som gör att en egen komponent behöver ritas om så anropar man repaint(). Man ska aldrig direkt anropa paintComponent. Detta görs av Swings egen loop som hanterar händelser, lyssnare och omritning av komponenter. Exempelvis kan detta ske när man ändrar fönstrets storlek.

DEMO: OlympicRingComponent.java, OlympicRingViewer.java, OlympicRingViewerTimer.java

Input från Mus och Tangentbord

```
private class MyMouseListener extends MouseAdapter {
        private JComponent comp;

        public MyMouseListener(JComponent comp) {
            this.comp = comp;
        }

        @Override
        public void mouseClicked(MouseEvent e) {
            comp.setVisible(!comp.isShowing());
        }
    }
```

```
// Kommer gömma/visa rings när man klickar på frame.
frame.addMouseListener(new MyMouseListener(rings));
```

Om man vill hantera input från mus använder man en speciell lyssnare, MouseListener, och händelser, MouseEvent. En lyssnare kopplas till den komponent som ska hantera mushändelser. Positionen för musen anges relativt denna komponent. MouseListener är ett gränssnitt med många metoder för olika typer av händelser; mouseClicked, mouseMoved etc. Är man bara intresserad av att hantera några få typer så kan man för bekvämlighets skull utgår från motsvarande adapter, MouseAdapter. Det är en klass där alla metoder har getts en tom implementation och man behöver bara överskugga dem man vill använda.

För tangentbordshändelser finns på motsvarande sätt interfacet KeyListener och KeyEvent.

Tidsstyrda händelser

```
// Retro flashing images.
// This will look truly horrible!
int delay = 100; //milliseconds
ActionListener taskPerformer = new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                rings.setVisible(!rings.isShowing());
            }
        };
new Timer(delay, taskPerformer).start();
```

Ibland, t.ex. i spel, vill man att saker ska förändras bara av att tiden går, inte att användaren ger något input. För program som använder Swing kan man använda klassen Timer. Den har en konstruktor som tar en heltal som anger i millisekunder hur ofta den ska generera en händelse och en ActionListener som anropas.

OBS! Det finns flera klasser som heter Timer i Javas bibliotek. Notera att detta är klassen Timer som hör till paketet Swing!

Mer om Swing...

- Menyer: `JMenuBar, JMenu, JMenuItem`.
- `Action`: Interface som abstraherar bort hur användaren startat det. E.g. knapp, meny osv.
- Dialogfönster: `JFileChooser, JOptionPane, JDialog`.
- Konsultera kursboken och Javas API!

Menyer
Till en JFrame kan man koppla en JMenuBar med metoden setJMenuBar. Till en JMenuBar kan man addera flera JMenu. Till en JMenu kan man addera flera JMenuItem.
Det går att skapa undermenyer, kortkommandon som automatiskt visas, ikoner, checkbox-alternativ i menyer och det mesta annat man ser i applikationer.

Actions
Ofta finns det flera sätt i GUIt att utföra ett visst kommando, t.ex. både en knapp och i menyerna. Vi kan tänka att flera olika komponenter utför samma handling (e.g. spara en fil).
Det finns ett koncept Action som abstraherar ett kommando/handling bort från hur användaren startat det. Samma Action kan startas på flera olika sätt. Action är ett interface som, liksom MouseListener, innehåller många metoder medan man ofta bara vill implementera en. Liksom MouseAdapter finns AbstractAction, en klass som låter definiera bara det nödvändiga.
För att säga ett en komponent, Button eller MenuItem, ska utföra en viss Action, använd metoden setAction. Knappar och menyalternativ som har en Action satt tar sin titel från denna, så tänk på att ställa in titeln på Action. Det finns en konstruerare för detta i AbstractAction.
DEMO: ActionDemo.java

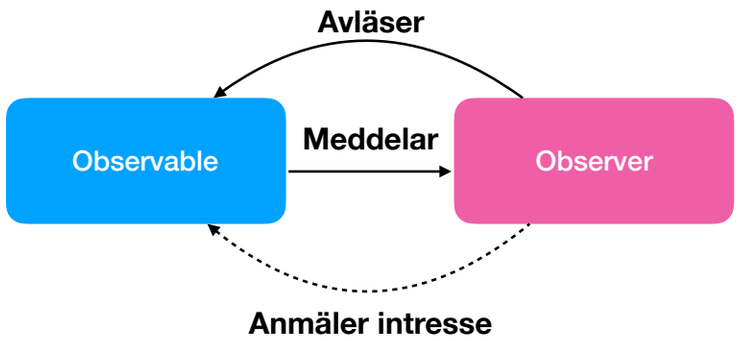
Dialogfönster
Det finns några klasser och metoder i Swing för att enkelt visa dialogfönster, t.ex.:

- JFileChooser – filväljardialog
- JOptionPane.showMessageDialog – visa meddelande i liten ruta

Man kan också bygga egna dialogfönster genom att utgå från klassen JDialog.

Designmönstret Observer (model-view)

- **Observable-object:** värden vars förändring är intressanta för observern. T.ex. en komponent i ett GUI.
- **Observer-objekt:** registrerar intresse, metod anropas vid förändring, nytt värde läses av. Ofta flera observers av samma observable.
- Designmönster hjälper oss strukturera kod.
- Avgränsa och skapa lämpliga klasser.



Sättet att implementera interaktionen i ett GUI med händelser och lyssnare är besläktat med ett designmönster (design pattern) för OOP, nämligen Observer. Den går ut på att objekt kan ha två roller, Observable eller Observer. Observable-objekt representerar värden vars förändringar är intressant för Observer-objekt. Därför har Observable-objekt en lista med Observer-objekt. I denna lista kan objekt som vill veta när något ändras i Observable-objektets värde/tillstånd registrera sig. När Observable-objektet ändrar sitt tillstånd så anropas en metod i de registrerade Observer-objekten som hanterar förändringen.

Detta kan man implementera enkelt genom ett interface för Observer som specificerar en metod som ska anropas vid förändring, på samma sätt som t.ex. ActionEventListener som vi sett tidigare. Varje klass som ska vara Observable tillhandahåller metoder för att lägga till sig till och ta bort sig från listan av Observers.

Detta är ett vanligt möster i GUI-komponenter. Här kallas det ibland Model-View. T.ex. kanske man har en klass som representerar ett antal mätningar från experiment och vill visa dem grafiskt på olika sätt. Man kan hålla reda på mätningarna i en klass (modellen) och sköta om det grafiska i en annan klass (view). När något ändras i modellen (e.g. ett nytt mätvärde har registrerats) så måste klassen som sköter grafiken notifieras så att den kan uppdatera presentationen av resultatet. Vy-objektet motsvarar alltså Observer-objektet i bilden ovan, medan Modell-objektet motsvarar Observable-objektet.

Ett annat exempel är t.ex. grafik på en valvaka, ofta har man många olika diagram som ska uppdateras allteftersom rösträkningen pågår.

Ett trivialt (men konkret exempel) är exemplet i denna föreläsning där vi ändrade OlymicRingViewer, men aldrig OlympicRingComponent.

Property Change

- Exempel på Observer-mönstret i Swing
- Om en egenskap hos `frame` ändras körs metoden `propertyChange`.

```
import java.beans.*;
....
....
PropertyChangeListener p = new PropertyChangeListener(){
          public void propertyChange(PropertyChangeEvent e){
            System.out.println("Egenskapen " + e.getPropertyName() +
                               " ändrades från " + e.getOldValue() +
                               " till " + e.getNewValue());
          }
        };
frame.addPropertyChangeListener(p);
```

Ett exempel på designmönstret Observer är en mekanism som finns implementerad i de olika komponentklasserna i Swing. De egenskaper objekt av dessa klasser har, t.ex. bakgrundsfärgen, har enhetliga setters och getters, setBackground, getBackground.

Men man kan också i andra objekt få reda på när någon egenskap har förändrats. Detta genom att klassen som vill ha informationen implementerar PropertyChangeListener som har en metod propertyChange. Alla komponentklasser har metoder för att registrera och avregistrera lyssnare/observers som implementerar detta interface. De metoderna heter addPropertyChangeListener och removePropertyChangeListener.

När någon egenskap i komponent-objekt förändras så skapas en PropertyChangeEvent och alla lyssnare/observers anropas. En PropertyChangeEvent innehåller information om namnet på egenskapen som förändrats, samt egenskapens gamla och nya värde.
DEMO: OlympicRingViewerPropChange.java (om tid finns).

Kort om JavaFX

- Ett modernare bibliotek för GUIs.
- Applikationer ärver från klassen `javafx.application.Application`.

```
import javafx.*;
public class HelloWorldFX extends Application {

   public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!”);
        primaryStage.show();
    }

   public static void main(String[] args) {
        launch(args);
    }
```

Ett alternativ till Swing är JavaFX. Om man vill får man använda JavaFX istället för Swing på Lab 3. Dock är instruktionerna skrivna för Swing, så man får i så fall söka information själv.

En GUI-applikation som använder JavaFX ärver från klassen Application. Denna är en abstrakt klass, och innehåller den abstrakta metoden start() som vi måste implementera (jämför med hur vi startade Swing-applikationer, med en run() metod).

start() tar ett objekt av typ Stage som argument. Detta skapar vi aldrig själva, det kommer från JavaFXs runtime. Stage representerar fönstret i vilken vårt GUI ska synas (jämför med JFrame). Metoden show() ska alltid kallas så att fönstret görs synligt (jämför med frame.setVisible(true)).

Applikationen startas naturligtvis som vanligt från main-metoden. Här kallar vi metoden launch(args), vilket är en statisk metod ärvd från Application. Den kommer sedermera att kalla metoden start() som vi implementerade ovan.

Kort om JavaFX

- Metaforen Stage - Scene (teater).
  - En (fysisk) scen där många teaterscener kan utspela sig.
- Innehåll läggs till genom att objekt av typen Scene läggs till vårt Stage objekt.

```
import javafx.*;
public class HelloWorldFX extends Application {

   public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!”);
        Label l = new Label("Hello World!”);
         Scene scene = new Scene(label, 400, 200);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

   public static void main(String[] args) {
        launch(args);
    }
```

För att visa något i ett fönster måste vi lägga till ett objekt av typen Scene till vår Stage. Metaforen är lättare att förstå på engelska: på en teaterscen (stage) kan många olika scener(ur pjäser) utspela sig.

Allt som ska visas i en JavaFX applikation måste tillhöra en Scene. I exemplet lägger vi till en text-etikett (Label). Man kan definiera många olika Scene-objekt som man kan alternera genom att visa.

Den första parametern i konstruktorn för Scene är roten i den s.k. “scene graph”. Denna är en trädliknade struktur som innehåller allt som ska synas, t.ex. GUI-komponenterna. Den intresserade kan läsa vidare om detta på egen hand.

Här finns en t.ex. en tutorial om JavaFX: http://tutorials.jenkov.com/javafx/index.html

DEMO: HelloWorldFX.java

Kort om JavaFX

- Lyssnare implementerar interfaces EventHandler<T>.
- För knappar: EventHandler<ActionEvent>

```
Button btn = new Button();
btn.setText("Say 'Hello World'");

btn.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
```

```
btn.setOnAction(actionEvent -> {
           System.out.println("Hello World! (using lambda)”);
         });
```

Här använder vi en s.k. anonym klass som lyssnare (I JavaFX kallad EventHandler). Vi ska bara använda den en gång, så vi struntar i att ge den ett namn. Notera att EventHandler<T> är en interface-typ, där T är någon typ av event som ska hanteras (här ett ActionEvent - knapptryck). Det är tillåtet att använda interface-typer med new när vi skapar anonyma klasser, eftersom den anonyma klassen inte har något namn.

Alt. kan vi använda ett lambda expression (funktion utan namn)
I API:et kan vi se att EventHandler är ett s.k. "funktionellt interface".
Det betyder att vi kan använda det i lambda-uttryck.