Föreläsning 7

Innehåll

Grafiska användargränsnitt i Java

- Orientering om JavaFX (ett ramverk för grafiska användargränssnitt)
  - Komponenter (fönster, knappar, ...)
  - Layout
- Händelsehantering (Hur man får någonting att hända när användaren t.ex. klickar på en knapp.)
  - Callback-metoder (som vi skriver, men som anropas av ramverket)
  - Anonyma klasser, lambdauttryck

JavaFX

Ett ramverk för grafiska användargränssnitt

- JavaFX är ett ramverk för grafiska användargränssnitt (eng. Graphical User Interface – GUI).
- Standard från Java 8. Vanligast i nya projekt.
- Andra ramverk
  - Android
  - Swing
  - AWT (Abstract Window Toolkit)
    - Delar av det används fortfarande i Swing (händelsehantering och layout).

Använda JavaFX

Kommentar

- Ramverket JavaFX innehåller många klasser. Ambitionen här är att visa hur man skriver en liten applikation i JavaFX samt ge smakprov på några olika slags komponenter.
- På nätet finns tutorials och dokumentation. Några länkar når du via kursens webbsida.

En JavaFX-applikation

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MyApp extends Application {

    public void start(Stage stage) {
        HBox root = new HBox();  // eller annan komponent
        // Skapa ytterligare komponenter och lägg till dem i root

        Scene scene = new Scene(root, 200, 80);
        stage.setScene(scene);
        stage.setTitle("Hello World");
        stage.show();
    }

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

En JavaFX-applikation

Metoden `start`

- Vår egen klass `MyApp` är en subklass till `Application`.
- `start` är en abstrakt metod i `Application` och måste implementeras:
  - Via parametern `stage` får man tillgång till ett `Stage`-objekt. `Stage`-objektet instansieras av ramverket och är en "top level"-behållare som innehåller det som ska visas i fönstret.
  - En komponent (rot) som i sin tur kan innehålla komponenter skapas.
  - Ett `Scene`-objekt kopplad till rot-komponenten skapas.
  - Metoden `setScene` anropas för att koppla `Scene`-objektet till `stage`.
  - `stage.show()` anropas för att visa fönstret.
- Analogi med teater: På ett scengolv (`stage`) kan en eller flera scener (`scene`) visas.

En JavaFX-applikation

Metoderna `init` och `stop`

- I klassen `Application` finns också metoderna `init` och `stop`.
  - De ärvs från `Application` och behöver inte implementeras.
  - `init` anropas innan JavaFX har satt upp ramverket.
  - `stop` anropas när programmet avslutas. Här kan du t.ex. spara data på fil.

```
public class MyApp extends Application {
    public void init() { ... }

    public void start(Stage stage) { ... }

    public void stop() { ... }

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


En JavaFX-applikation

Metoden `launch`

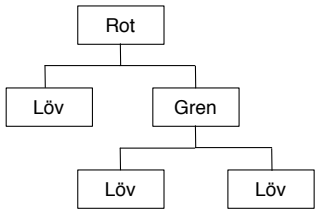
- I `main`-metoden ska metoden `Application.launch()` anropas.
  - main behöver inte ligga i `MyApp`. Du kan även starta JavaFX med `Application.launch(MyApp.class, args)`.
- `launch` gör följande:
  - skapar en instans av `MyApp`
  - anropar `init`
  - anropar `start`
  - väntar på att JavaFX-applikationen ska avslutas vilket inträffar när något av följande inträffar
    - `Platform.exit()` anropas
    - sista fönstret stängts
  - anropar `stop`

Trådar

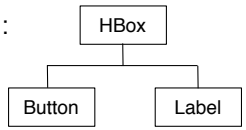
- Trådar används om ett program ska utföra olika uppgifter samtidigt. Processorn exekverar var och en av trådarna en kort stund i taget. För användaren ser det ut som om processorn utför de olika uppgifterna samtidigt.
- JavaFX använder flera trådar.
- Alla JavaFX-objekt måste skapas i `Application.start()` eller i en händelsehanterare (beskrivs senare).

Scengraf

- De grafiska komponenterna som visas på skärmen representeras av objekt.
- Objekten arrangeras i en hierarkisk, trädformad struktur - scengrafen.
- Scengrafen består av olika noder:



Exempel:



- Scene-objektet är behållare för allt innehåll i scengrafen.
- JavaFX ritar automatiskt alla komponenter, uppdaterar vid behov, t.ex. när fönstrets storlek ändras.

Grafiska komponenter

Exempel, fler finns

- Styrkomponenter
  - Används för att interagera med användare.
  - Finns i paketet `javafx.scene.control`
  - Ex: knappar, menyer, textrutor
  - Reagerar på händelser (musklick, menyval, . . .).
- Former
  - Geometriska former som ritas, men som användaren inte kan påverka.
  - Ex. rektangel, linje och cirkel
  - Finns i paketet `javafx.scene.shape`
- Behållarkomponenter
  - Kan innehålla andra komponenter och används för att organisera komponenter.
  - Ofta osynliga
  - Lägg till barn med metoderna `getChildren().addAll()` eller `getChildren().add()`

Behållarkomponenter

javafx.scene.control.Control

- Objekt som användaren interagerar med.
- Exempel MenuBar, ToolBar, TabPane.

javafx.scene.layout.Region

- anpassar storlek när fönstret ändras.
- Subklasser till Region placerar barnen enligt en layout
  - StackPane, HBox, VBox, TilePane, FlowPane, BorderPane, GridPane, och AnchorPane.

javafx.scene.Group

- Fix storlek, ändras inte när fönstret ändras.

Avsluta program

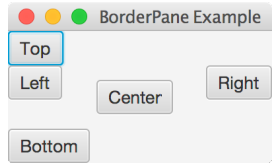
- Ett JavaFX-program avslutas när alla fönster stängts, av användaren eller programmet.
- Om du vill avsluta programmet, t.ex. från en händelsehanterare, anropar du `Platform.exit()`.
- `stop` i din `Application`-klass anropas i båda fallen ovan.
- Om du anropar `System.exit(status)` kommer `stop` inte att anropas.

Behållarkomponenter med layout

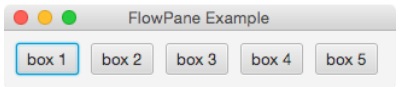
- Vissa behållarkomponenter har en layout som bestämmer storlek och läge för komponenterna i behållaren.
- De räknar också ut nya positioner och storlekar om fönstrets storlek ändras.
- Olika typer av behållarkomponenter har olika strategier för placering av komponenterna. Ex:
  - `BorderPane` – delar utrymmet i fem delar; norr, söder, öster, väster och mitten.
  - `FlowPane` – komponenterna placeras i en rad efter varandra.
  - `HBox` – horisontell rad.
  - `VBox` – vertikal rad.
  - `GridPane` – rutnät.
  - `TilePane` – rutnät, alla rutor lika stora.
  - `AnchorPane` – ankrat till sida/hörn.

Exempel: BorderPane

```
BorderPane root = new BorderPane();
root.setTop(new Button("Top"));
root.setLeft(new Button("Left"));
root.setCenter(new Button("Center"));
root.setRight(new Button("Right"));
root.setBottom(new Button("Bottom"));
root.setPrefSize(200, 100);
```

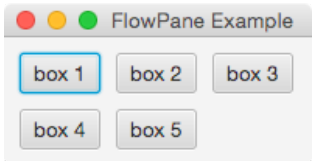


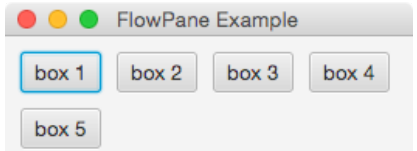
Exempel: FlowPane

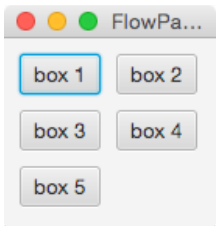



```
FlowPane root = new FlowPane();
root.setHgap(10);
root.setVgap(10);
root.setPadding(new Insets(10, 10, 10, 10));
for(int i = 1;  i<= 5; i++){
    root.getChildren().add(new Button("box " + i));
}
```


FlowPane - ändring av layouten när fönstret ändrar storlek

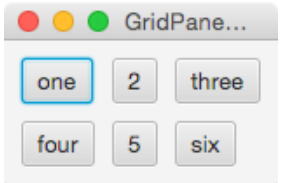






Exempel: GridPane

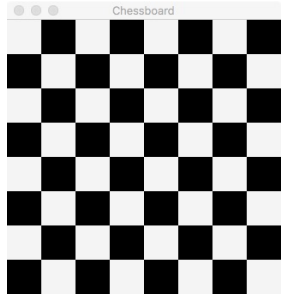
Rader och kolumner kan ha olika storlek.




```
GridPane root = new GridPane();
root.setHgap(10);
root.setVgap(10);
root.setPadding(new Insets(10, 10, 10, 10));
root.add(new Button("one"), 0, 0);
root.add(new Button("2"), 1, 0);
root.add(new Button("three"), 2, 0);
root.add(new Button("four"), 0, 1);
root.add(new Button("5"), 1, 1);
root.add(new Button("six"), 2, 1);
```


Exempel: TilePane

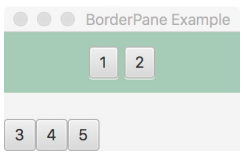
Som GridPane, men cellerna har samma storlek.




```
TilePane root = new TilePane();
root.setPrefColumns(8);
root.setPrefRows(8);
final int SIZE = 40;
for (int i = 0; i < 8; i++) {
    for (int k = 0; k < 8; k++) {
        Label label = new Label();
        label.setPrefSize(SIZE, SIZE);
        if ((i + k) % 2 != 0) {
            label.setStyle("-fx-background-color: #000000;");
        }
        root.getChildren().add(label);
    }
}
stage.setResizable(false);
```


Exempel: Nästlade layouthanterare

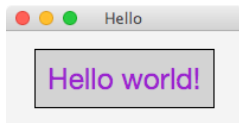
```
HBox topBox = new HBox();
topBox.setPadding(new Insets(12, 12, 12, 12));
topBox.setSpacing(5);
topBox.setAlignment(Pos.CENTER);
topBox.setStyle("-fx-background-color: #adcab8;");
topBox.getChildren().addAll(new Button("1"), new Button("2"));

HBox bottomBox = new HBox();
bottomBox.getChildren().addAll(new Button("3"), new Button("4"),
                               new Button("5"));

BorderPane root = new BorderPane();
root.setTop(topBox);
root.setBottom(bottomBox);
root.setPrefSize(200, 100);
```



JavaFX och CSS

- Man kan ändra utseende på det grafiska användargränssnittet med hjälp av stilmallar, CSS (Cascading Style Sheets).
- Exempel:

```
public void start(Stage stage) {
    Label label = new Label("Hello world!");
    label.setStyle("-fx-background-color: lightGrey;"
                 + "-fx-text-fill: #ff7f50;"
                 + "-fx-font-size: 24; -fx-label-padding: 10;"
                 + "-fx-border-color: black;");

    Pane root = new StackPane();
    root.getChildren().add(label);

    Scene scene = new Scene(root, 200, 80);
    stage.setScene(scene);
    stage.setTitle("Hello");
    stage.show();
}
```



JavaFX och CSS

Style sheet

- Alternativt kan man lägga CSS-koden i en fil.
- Antag att filen `scene.css` innehåller:

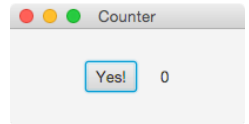
```
.my-label {
    -fx-background-color: lightGrey;
    -fx-text-fill: #ff7f50;
    -fx-font-size: 24;
    -fx-label-padding: 10;
    -fx-border-color: black;
}
```

Då kan man skriva så här i `start`-metoden:

```
...
label.getStyleClass().add("my-label");
// Ger etiketten en identifierare som används i stilmalle
...
scene.getStylesheets().add("scene.css");
// Laddar stilmallen i filen scene.css
...
```


Exempel: Fönster med en knapp

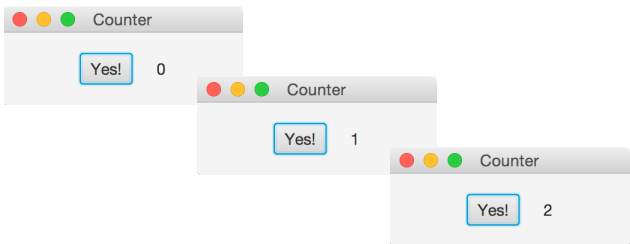
```
public class CounterView extends Application {
    public void start(Stage stage) {
        Button button = new Button("Yes!");
        Label label = new Label("0");

        HBox root = new HBox();
        root.setSpacing(20);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(button, label);

        Scene scene = new Scene(root, 200, 80);
        stage.setScene(scene);
        stage.setTitle("Counter");
        stage.show();
    }

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



Händelsehantering

När användaren klickar på en knapp, väljer ett menyalternativ . . . händer följande:

1. **Ett händelseobjekt skapas**.
2. JavaFX-systemet **anropar en callback-metod på de lyssnarobjekt** som är knutna till komponenten.



Lyssnarobjekt

- Ett lyssnarobjekt är ett objekt av en klass som implementerar interfacet:

```
public interface EventHandler<T extends Event> {
    void handle(T event);
}
```

  - Ett lyssnarobjekt kan knytas till en eller flera komponenter (t.ex. en knapp).
  - När komponentens händelse inträffar (t.ex. när någon klickar på knappen) anropas callback-metoden `handle`.
    - Det är alltså i metoden `handle` vi ska skriva vad som ska hända när man klickar på knappen.
    - En komponent kan ha flera lyssnarobjekt, alla anropas.
  - `Event` är superklass för händelseklasserna i JavaFX. Den händelseklass som ska användas här är `ActionEvent`.

Händelsehantering

För att få någonting att hända när användaren klickat på en knapp måste man:

- Skriva en klass som implementerar interfacet `EventHandler`.
  - I callback-metoden `handle` skriver man det man vill ska hända när användaren klickar på knappen.
- Koppla lyssnar-objektet till knappen genom att anropa metoden `setOnAction`.

Exempel på händelsehantering

Klassen `CounterView`

```
public class CounterView extends Application {
    private Button button;
    private Label label;
    private int counter;

    private class ButtonHandler implements
                                    EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            counter++;
            label.setText(Integer.toString(counter));
        }
    }

    public void start(Stage stage) {
        /** se nästa sida **/ }
```


Exempel på händelsehantering

Klassen `CounterView`, start-metoden

```
    public void start(Stage stage) {
        button = new Button("Yes!");
        button.setOnAction(new ButtonHandler());
        label = new Label("0");

        HBox root = new HBox();
        root.setSpacing(20);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(button, label);

        Scene scene = new Scene(root, 200, 80);
        stage.setScene(scene);
        stage.setTitle("Counter");
        stage.show();
    }

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


Händelsehantering

Kommentar

- Tidigare har vi vant oss vid att det är vi som styr vad som ska hända i programmet genom att skriva kod för att fråga användaren om olika indata etc.
- Nu vänder vi på det hela. Det är användaren som styr vad som ska hända genom att klicka på en viss knapp, välja ett menyalternativ etc.
- Vi ”fyller i” vad som ska hända i de olika fallen genom att implementera callback-metoder i lyssnarklasser.

Anonyma klasser

Klasser som bara instansieras en gång behöver inte namnges och kan skapas direkt ("inline") vid `new`.

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        counter++;
        label.setText(Integer.toString(counter));
    }
});
```

Det blir nu tydligare vilken funktionalitet som kopplas till knappen.

Funktionella interface

- Ett interface med en enda abstrakt metod kallas **funktionellt interface**.
  - Interfacet `EventHandler` har bara den abstrakta metoden `handle` och är därför ett funktionellt interface:

    ```
    public Interface EventHandler<T extends Event> {
       void handle(T event);
    }
    ```

  - (Ett funktionellt interface kan innehålla godtyckligt många statiska- och default-metoder.)
- Ett annat exempel på ett funktionella interface i Java är `Comparator<T>`.

Lambdauttryck

- Vi såg tidigare hur man kunde skapa ett lyssnarobjekt av typen `EventHandler` genom att skriva en anonym klass:

  ```
  button.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent event) {
          counter++;
          label.setText(Integer.toString(counter));
      }
  });
  ```

  Eftersom interfacet `EventHandler` är ett funktionellt interface kan man förenkla ytterligare genom att använda ett lambdauttryck:

  ```
  button.setOnAction(event -> {
      counter++;
      label.setText(Integer.toString(counter));
  });
  ```


Lambdauttryck

- Lambdauttryck infördes i Java 8.
- Förenklat sett är lambdauttryck anonyma klasser där man bara skriver innehållet i en metod. Kompilatorn listar ut vilken metod och alla typer från sammanhanget.

  ```
  button.setOnAction(event -> {
      counter++;
      label.setText(Integer.toString(yesCounter));
  });
  ```

- Lambdauttryck skapar objekt, d.v.s. används normalt istället för `new`.

Lambdauttryck
Syntax

- Lambdauttryck skrivs på formen

```
(parametrar) -> metodkropp
```

- Exempel:

```
(ActionEvent event) -> {
    counter++;
    label.setText(Integer.toString(counter));
}
```

  - Typer på parametrar behöver inte anges om de är otvetydiga:

```
(event) -> ...
```

  - Är det bara en parameter kan även `()` utelämnas:

```
event -> ...
```


Lambdauttryck
Syntax, forts

- Metodkroppen består av ett *block*:

```
(a, b) ->  {
    return a.length() - b.length();
}
```

  eller ett *uttryck*:

```
(a, b) -> a.length() - b.length()
```

  - Om metodkroppen består av ett uttryck ska även `return` utelämnas.

- Exempel:

```
String[] words = {"AA", "EEE","B", "CCCCC", "DDDD"};
Arrays.sort(words, (a, b) -> a.length() - b.length());
```


Lambdauttryck
Ytterligare exempel - överkurs

Interfacet `List` har en metod `removeIf` som tar bort element givet ett predikat. Följande predikat tar bort alla jämna heltal:

```
List<Integer> list = ...
list.removeIf(new Predicate<Integer>() {
   public boolean test(Integer x) {
      return x % 2 == 0;
   }
});
```

Ovanstående anonyma klass kan skrivas om till ett lambdauttryck:

```
list.removeIf((Integer x) -> {
   return x % 2 == 0;
});
```

Som i sin tur kan skrivas om till formen där metodkroppen är ett uttryck:

```
list.removeIf(x -> x % 2 == 0);
```


Lambdauttryck
Restriktioner i Java

- I ett lambda-uttryck kan man använda
  - attribut i den omgivande klassen
  - variabler som är deklarerade `final` eller som ej ändras ("effectively final") i den omgivande metoden.

- Så här kan man göra:

```
int n = 10;
list.forEach(e -> System.out.println(e + n));
```

- Men inte så här:

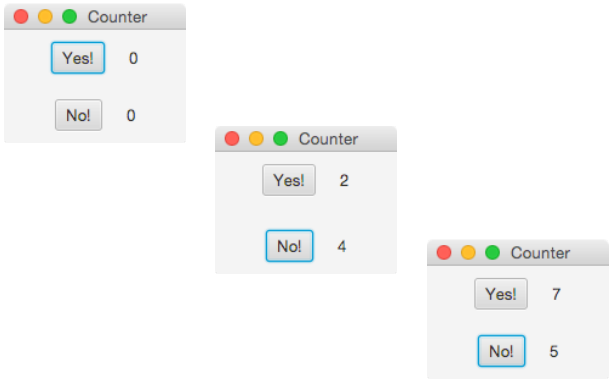
```
int n = 10;
n = 20;  // värdet på n ändras här
list.forEach(e -> System.out.println(e + n));
```

  och inte heller så här:

```
int n = 0;
list.forEach(e -> n += e);  // värdet på n ändras här
```

Händelsehantering - flera komponenter

- Man kan ha flera komponenter som genererar händelser.
  - T.ex. olika knappar, textfält, menyer, ...
- Olika saker ska hända beroende på vilken av komponenterna som genererade händelsen.



Flera komponenter med var sin lyssnare

Klassen `CounterView`

```
public class CounterView extends Application {
    private Button yesButton, noButton;
    private Label yesLabel, noLabel;
    private int yesCounter, noCounter;
    ...
    public void start(Stage stage) {
        yesButton = new Button("Yes!");
        ...
    }

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


Flera komponenter med var sin lyssnare

Klassen `CounterView`, metoden start

```
    public void start(Stage stage) {
        yesButton = new Button("Yes!");
        yesButton.setOnAction(event -> {
                yesCounter ++;
                yesLabel.setText(Integer.toString(yesCounter));
            }
        );
        yesLabel = new Label("0");

        noButton = new Button("No!");
        noButton.setOnAction(event -> {
                noCounter ++;
                noLabel.setText(Integer.toString(noCounter));
            }
        );
        noLabel = new Label("0");
        ...
    }
}
```


Diskutera

Duplicerad kod

- Den lösning vi visat innehåller en hel del duplicerad kod, två knappar, två etiketter, två räknare ...
  - Den enda skillnaden är namnet på knapparna.
- Koden blir onödigt lång och förändringar innebär att man måste göra samma ändring på flera ställen i koden.
- Ännu värre blir det om vi lägger till en tredje knapp.
- Hur ska vi möblera om i koden för att lösa det?

Undvik duplicerad kod

Klassen `CounterPane` med räknare, knapp och etikett

```
public class CounterPane extends HBox  {
    private int counter = 0;
    private Button button;
    private Label label;

    public CounterPane(String s) {
        button = new Button(s);
        button.setOnAction(event -> {
                counter++;
                label.setText(Integer.toString(counter));
            }
        );
        label = new Label("0");
        setPadding(new Insets(10, 10, 10, 10));
        setSpacing(20);
        setAlignment(Pos.CENTER_LEFT);
        getChildren().addAll(button, label);
    }
}
```

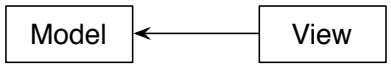

Undvik duplicerad kod

Kortare metod `start` i klassen `CountersView`

```
public class CountersView extends Application {

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        root.getChildren().add(new CounterPane("Yes"));
        root.getChildren().add(new CounterPane("No"));
        root.getChildren().add(new CounterPane("Neutral"));
        root.setPrefSize(150, 100);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Counters");
        stage.show();
    }

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


Model-View

- De lösningar vi visat separerar inte det grafiska användargränssnittet (vyn) från modellen (räknarna). Oftast är önskvärt att separera dessa.
  - Vyn kan man ofta vilja ändra medan modellen ligger fast.
  - Vissa operationer berör bara modellen. Det blir besvärligt att implementera och testa dessa om vy och modell blandas samman.
- Klassen/klasserna som beskriver modellen känner inte till vyn:



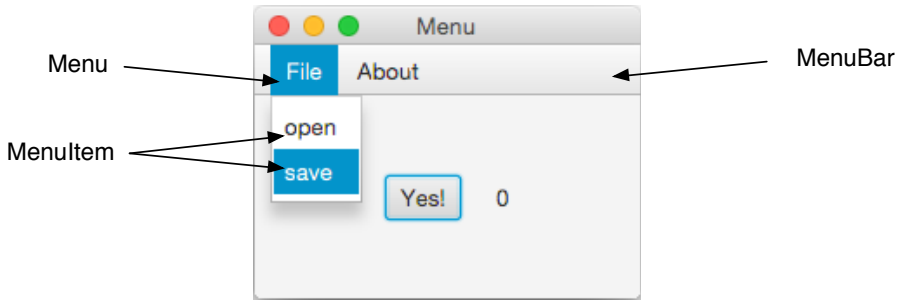
- Ibland talar man även om en kontrolldel (*Model-View-Controller*). Denna reagerar på användarens input, uppdaterar vyn och påverkar modellen. I praktiken är vyn och kontrollen ofta svåra att skilja åt.

Menyer

`MenuBar`, `Menu`, `MenuItem`

Menyer

- För att välja mellan alternativ som kan grupperas i en eller flera menyer.
- Händelsehantering analogt med knappar.



Fler komponenter

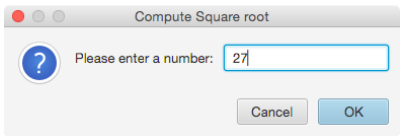
TextField, TextArea

- TextField
  - Ett editerbart textfält – en rad.
  - När användare skriver in radslutstecken genereras ActionEvent.
  - textField.getText() returnerar den sträng som skrivits i fältet.
- TextArea
  - Flera rader.
  - Kan sättas "uneditable" och användas för att visa flera rader text.
  - Kan vara "editable" och användas för att skriva in flera rader text.
  - Radslut genererar ActionEvent.
  - textArea.getText() returnerar det som skrivits (i form av en sträng).

Dialogrutor

Exempel, TextInputDialog

```
public void start(Stage arg0) {
    TextInputDialog dialog = new TextInputDialog(" ");
    dialog.setTitle("Compute Square root");
    dialog.setHeaderText("");
    dialog.setContentText("Please enter a number:");
    Optional<String> result = dialog.showAndWait();
    ...
```



Klassen Optional<T>

Behållare för resultat

- Resultatet man får från dialogrutan är ett Optional<String>-objekt.
- I klassen Optional<T> finns bland annat metoderna:

  boolean isPresent() – returnerar true om det finns ett resultat, annars false (t.ex. om användaren klickat på "Cancel")

  T get() – returnerar resultatet ifall ett sådant finns, genererar i annat fall NoSuchElementException
- Exempel på användning:

```
Optional<String> result = dialog.showAndWait();
    if (result.isPresent()) {
        String s = result.get();
        ...
```


Fler komponenter – dialogrutor

Använda Optional och Alert

```
    ...
    Optional<String> result = dialog.showAndWait();
    if (result.isPresent()) {
        try {
            n  = Double.valueOf(result.get());
            double sqrRoot = Math.sqrt(n);
            Alert alert = new Alert(AlertType.INFORMATION);
            alert.setTitle("Result");
            alert.setHeaderText(null);
            alert.setContentText(Double.toString(sqrRoot));
            alert.showAndWait();
        } catch (NumberFormatException e) {
            Alert alert = new Alert(AlertType.ERROR);
            alert.setTitle("Error in input");
            alert.setHeaderText(null);
            alert.setContentText("Wrong input");
            alert.showAndWait();
        }
    }
}
```

Fler komponenter – Listor

`ListView`

- För att visa en skrollbar lista i en vy finns klassen `ListView`.
- Till en `ListView` kopplas modellen av listan – av typ `ObservableList<T>`.
  - `ObservableList<T>` är ett interface.
  - Det finns färdiga implementeringar av interfacet – t.ex. `ObservableArrayList<T>`.
- Alla modifieringar av listan (modellen) görs via operationer på den modell som kopplas till vyn.
  - Dessa modifieringar visas automatiskt i vyn av listan.

Grafik

- Man kan enkelt rita figurer som linjer, cirklar, rektanglar ...
- De geometriska figurer finns i paketet `javafx.scene.shape`.
- Lägg till dem i scenen på samma sätt som andra GUI-komponenter, `container.getChildren().add(new Circle(x, y, r));`

Exempel på vad du ska kunna

- Implementera enkla användargränsnitt med hjälp av JavaFX.
- Beskriva och kunna tillämpa principerna för händelsehantering i JavaFX.
- Kunna använda lambda-uttryck i Java.
- Kunna separera modell och vy (grafiskt användargränssnitt) vid implementering av program med grafiska användargränssnitt.

Datorlaboration 3

grafiska användargränssnitt – JavaFX

- Skapa en grafisk applikation med JavaFX från grunden, steg för steg
- Bygger på laboration 1: användargränssnitt för textanalysen
- Lambda-uttryck
- Valbara uppgifter: lös problem med hjälp av dokumentationen