Creating GUI with JavaFX

Creating GUI with JavaFX

- JavaFX application structure
- User interface design
- Event-based programming

JavaFX application structure

- A JavaFX application has a tree structure, with the root being the **stage**
- The stage depends on the concrete operating system and device on which the application is executing:
  - On a desktop OS, the stage will be a window
  - Inside a Web page, the stage will be an applet
- Most of the time, the developer does not need to worry where the application is executing, which is an advantage of JavaFX over many other GUI APIs (including Swing)

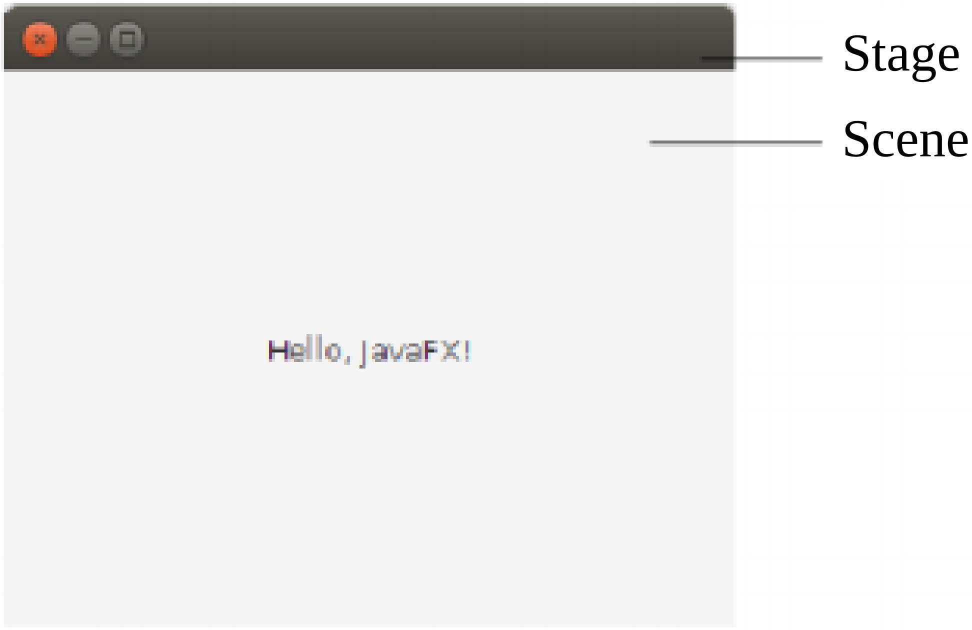
JavaFX application structure

- The stage contains exactly one **scene**, which contains **nodes**

- A node is most often a visual component such as a button or text field, but can also be an image, multimedia player, etc.

- The main class should extend class Application and override its **start** method which accepts one parameter of type Stage – the main stage of the application
  - JavaFX applications are always initialized in this method

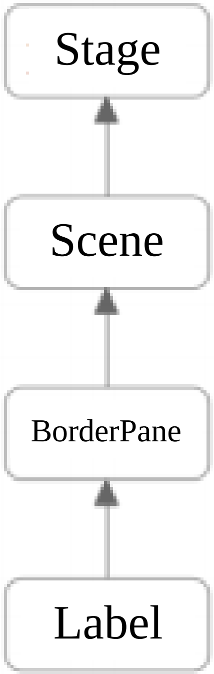
JavaFX application structure



Hello, JavaFX!

Stage

Scene



Stage

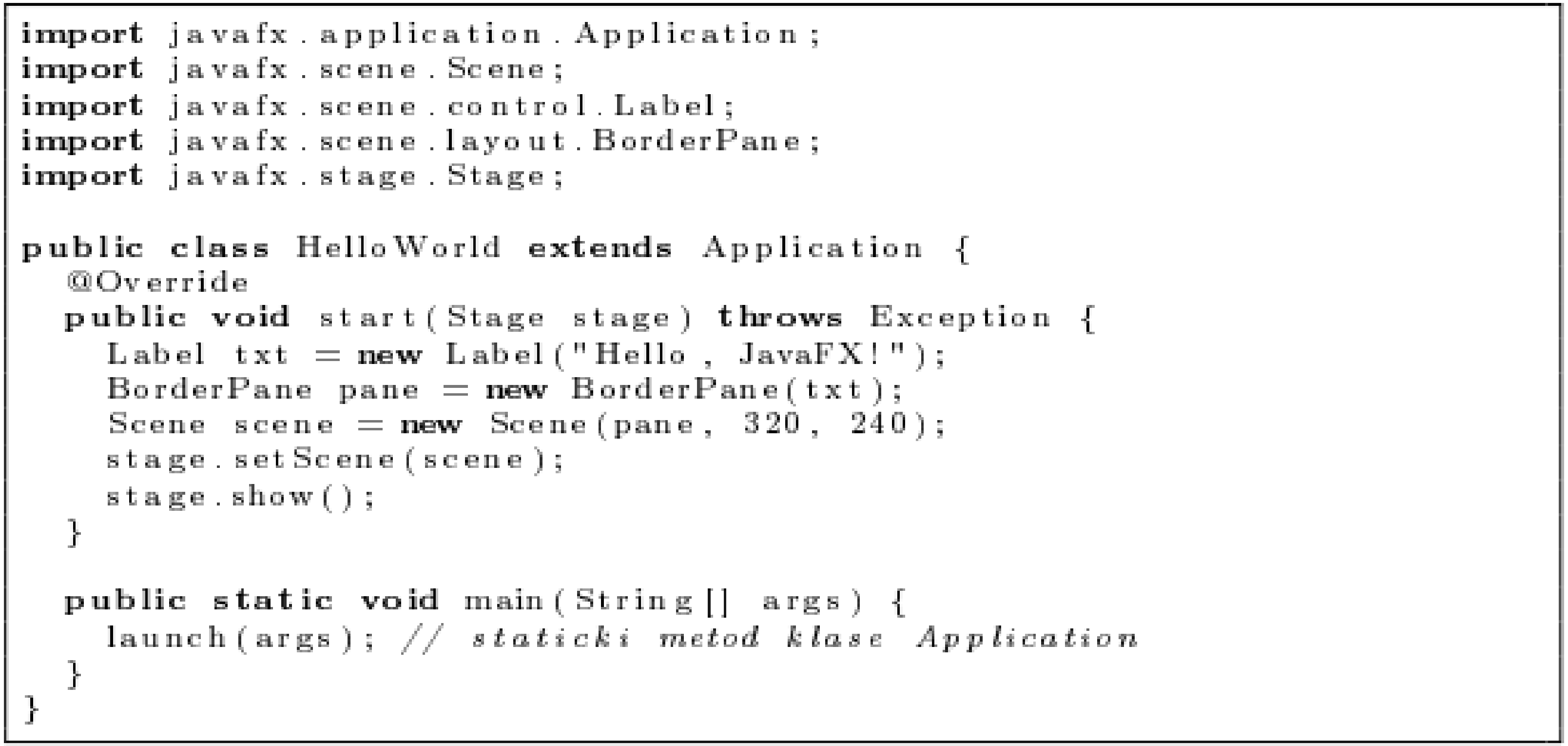
Scene

BorderPane

Label

JavaFX application structure

Primer 12.1: Primer jednostavne JavaFX aplikacije.



```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Label txt = new Label("Hello, JavaFX!");
    BorderPane pane = new BorderPane(txt);
    Scene scene = new Scene(pane, 320, 240);
    stage.setScene(scene);
    stage.show();
  }

  public static void main(String[] args) {
    launch(args); // staticki metod klase Application
  }
}
```

JavaFX application structure

- The scene is set by a call to method **setScene** of a **Stage** object
- In order for the stage to be visible on screen, its method **show** needs to be called
- The scene usually does not contain visual components directly
- Rather, the usual practice is to insert a **pane** into the scene, and put components inside the pane

Creating GUI with JavaFX

- JavaFX application structure
- User interface design
- Event-based programming

User interface design

- **Panes** – objects that contain and arrange nodes on the scene
- Panes adapt to the scene’s dimensions, so that the sizes and positions of the adapt to the changes of the scene’s size
- It is possible to define exact sizes and positions of each node as it is inserted:
  - This approach is not recommended in real-world applications
  - The inserted nodes are static – they will not move nor resize as the user changes the window’s size

User interface design

- JavaFX nodes do not have only one field defining size, but three:

  Minimal, preferred and maximal size

- Preferred size is the size a node will have if there is enough space on the scene (pane)
- When the user shrinks the window, the pane will shrink the nodes, but not below their minimal sizes
  - If minimal sizes do not provide enough space for all nodes, they may overlap
- When the user enlarges the window, some panes will enlarge the nodes, but not more than their maximal sizes

User interface design

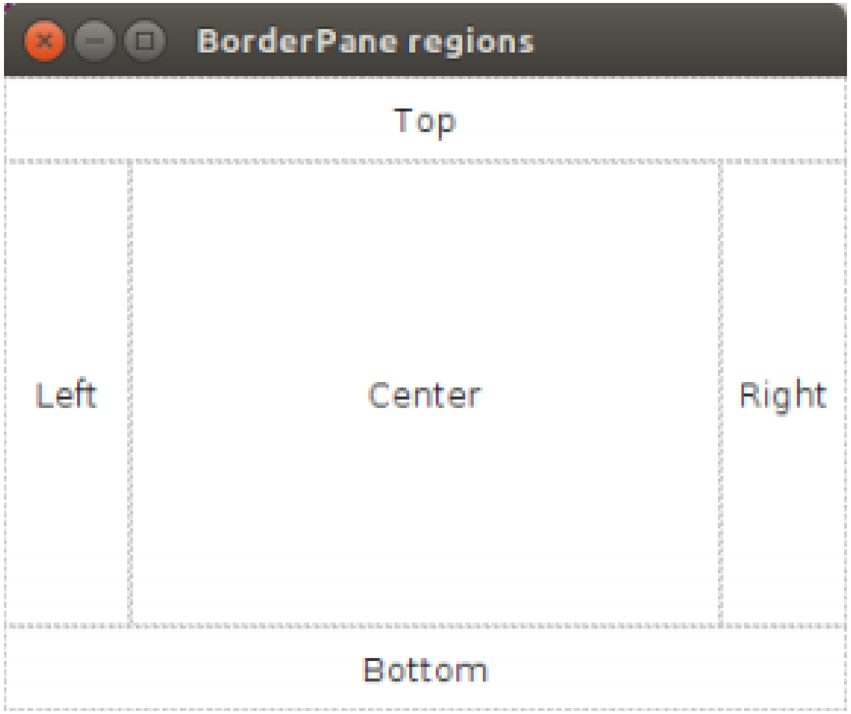
- Node sizes can be set using methods
  - **set*Width(double value)**
  - **set*Height(double value)**
  - **set*Size(double width, double height)**

  where * is replaced by **Min**, **Pref** and **Max**, respectively, for minimal, preferred and maximal size

- Unlimited size is denoted by Double.MAX_VALUE

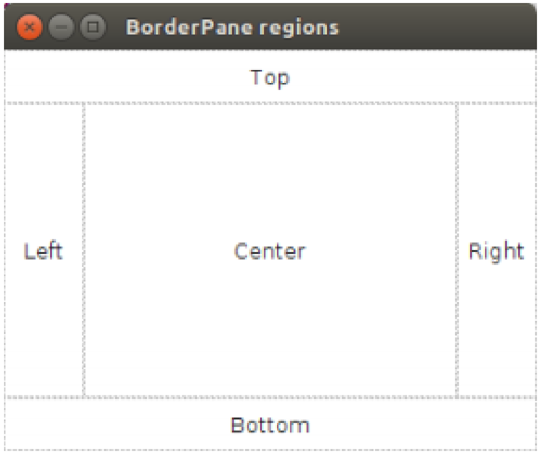
BorderPane

- BorderPane divides the scene into 5 regions, 4 of which are at the edges (denoted Top, Bottom, Left and Right), and one is central (denoted Center).
- Every region can contain at most one node



BorderPane

- Package: javafx.scene.layout
- Constructors:
  - BorderPane()
  - BorderPane(Node center)
  - BorderPane(Node c, Node t, Node r, Node b, Node l)
- Methods:
  - void setCenter(Node n)
  - void setLeft(Node n)
  - void setRight(Node n)
  - void setTop(Node n)
  - void setBottom(Node n)
  - void setPadding(Insets value)
    - Sets the margin around the pane to the Insets value
    - Insets constructors:
      - Insets(double topRightBottomLeft) – all margins identical
      - Insets(double top, double right, double bottom, double left)



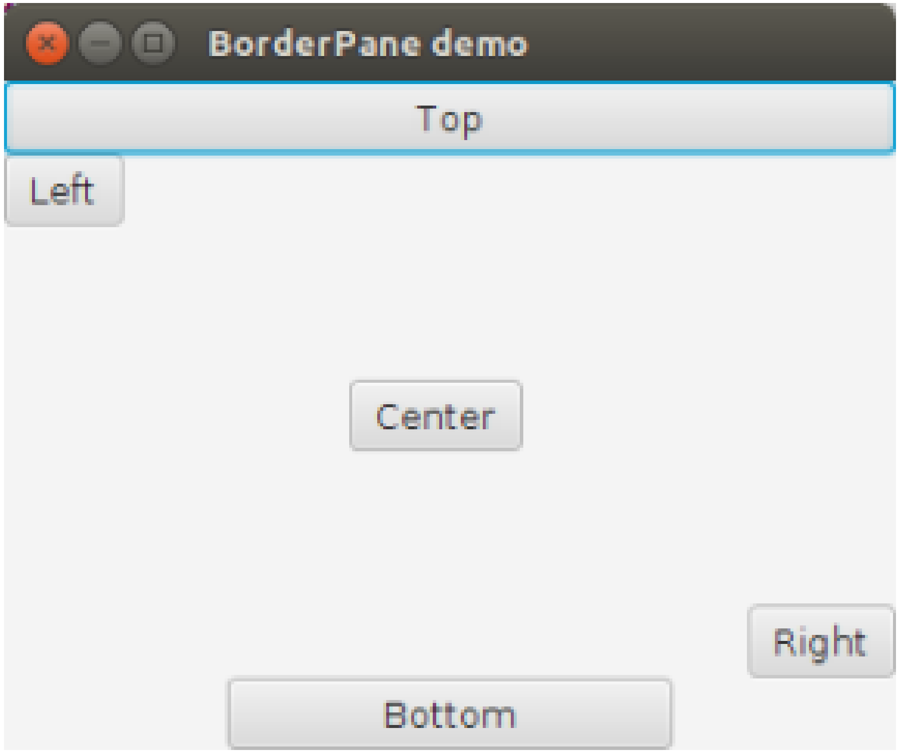
BorderPane

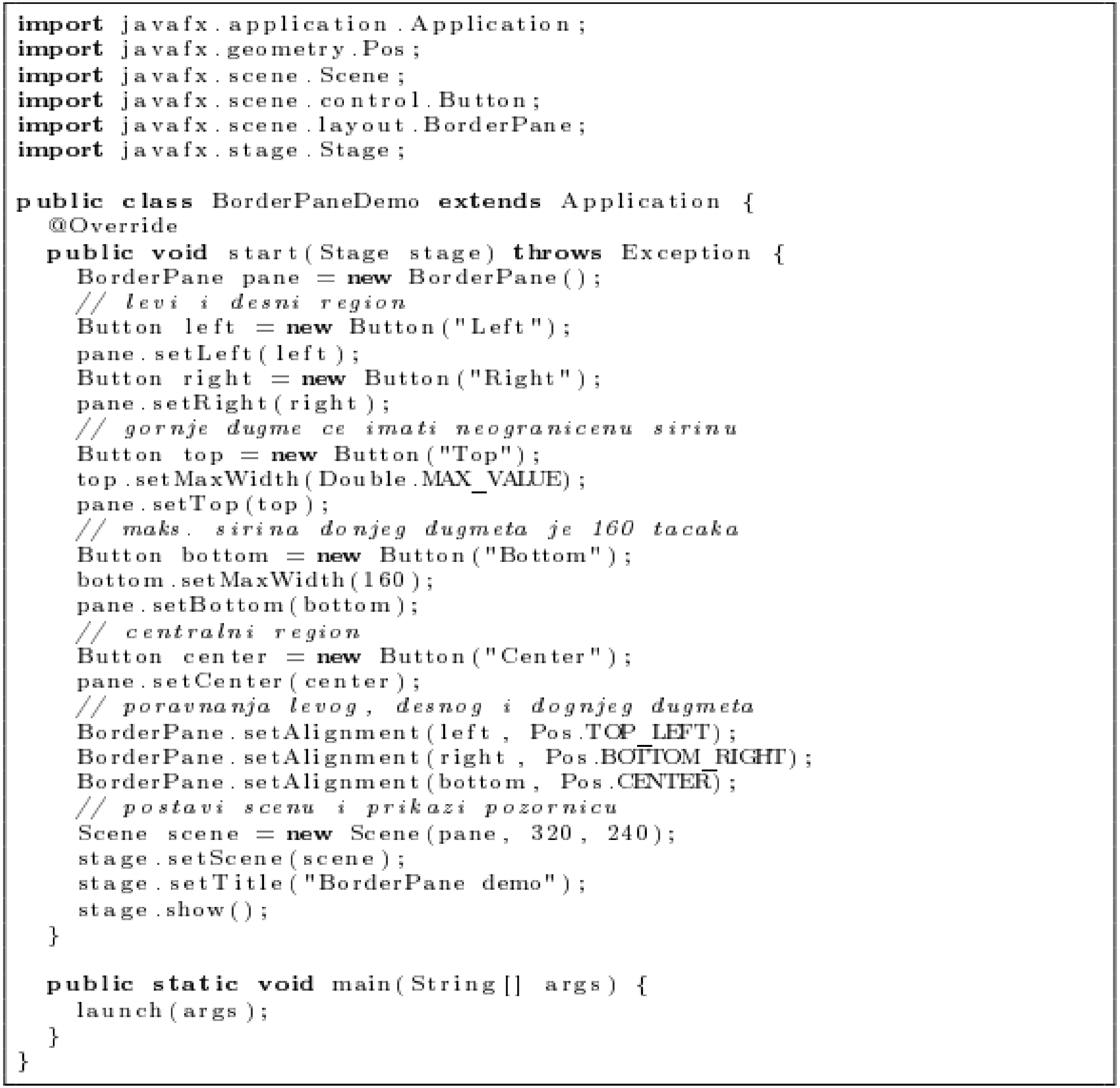
- Sizes of BorderPane’s edge regions depend on node sizes
- The nodes in the top and bottom regions will have their preferred height and maximal width
- Similarly, nodes in the left and right regions will have their preferred width and maximal height
- Edge regions that do not contain nodes are not visible
- The central region occupies all remaining space, its node has maximal size

Static methods of class BorderPane

- **static void setMargin(Node child, Insets value)**
  - Sets the margin around Node child in the part of the pane the child is in to the Insets value
- **static void setAlignment(Node child, Pos value)**
  - Sets the alignment of Node child in the part of the pane it occupies
  - enum Pos constants:
    - Pos.CENTER – centered both vertically and horizontally
    - Other constants are of the form Pos.XXX_YYY
      XXX $\in$ {TOP, BOTTOM, CENTER} – vertical alignment
      YYY $\in$ {LEFT, RIGHT, CENTER} – horizontal alignment
    - For example: Pos.TOP_RIGHT, Pos.BOTTOM_LEFT, etc.

Example – Adding buttons to BorderPane





```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class BorderPaneDemo extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    BorderPane pane = new BorderPane();
    // levi i desni region
    Button left = new Button("Left");
    pane.setLeft(left);
    Button right = new Button("Right");
    pane.setRight(right);
    // gornje dugme ce imati neogranicenu sirinu
    Button top = new Button("Top");
    top.setMaxWidth(Double.MAX_VALUE);
    pane.setTop(top);
    // maks. sirina donjeg dugmeta je 160 tacaka
    Button bottom = new Button("Bottom");
    bottom.setMaxWidth(160);
    pane.setBottom(bottom);
    // centralni region
    Button center = new Button("Center");
    pane.setCenter(center);
    // poravnanja levog, desnog i dognjeg dugmeta
    BorderPane.setAlignment(left, Pos.TOP_LEFT);
    BorderPane.setAlignment(right, Pos.BOTTOM_RIGHT);
    BorderPane.setAlignment(bottom, Pos.CENTER);
    // postavi scenu i prikazi pozornicu
    Scene scene = new Scene(pane, 320, 240);
    stage.setScene(scene);
    stage.setTitle("BorderPane demo");
    stage.show();
  }

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

HBox, VBox and FlowPane

- HBox and VBox arrange nodes on a horizontal and vertical line, respectively
  - Node sizes will initially be set to their preferred sizes, and will be shrunk if needed, respecting their minimal sizes
- FlowPane can arrange nodes both horizontally and vertically, which is determined by a constant of type Orientation
  - Will set the sizes of its nodes to their preferred values
  - If there is not enough space, FlowPane will attempt to display all nodes by breaking rows/columns, but will not change their sizes

FlowPane

- Package: javafx.scene.layout (as well as all Pane classes)
- Constructors:
  - FlowPane()
  - FlowPane(double hgap, double vgap)
  - FlowPane(Orientation orient)
  - FlowPane(Orientation orient, double hgap, double vgap)

  - Enum Orientation:
    - Orientation.HORIZONTAL – arrange nodes horizontally from left to right, if needed break to “next line”
    - Orientation.VERTICAL – arrange nodes vertically from top to bottom, if needed break to “next column”

FlowPane

- **ObservableList<Node> getChildren()** returns the list of nodes contained in the pane

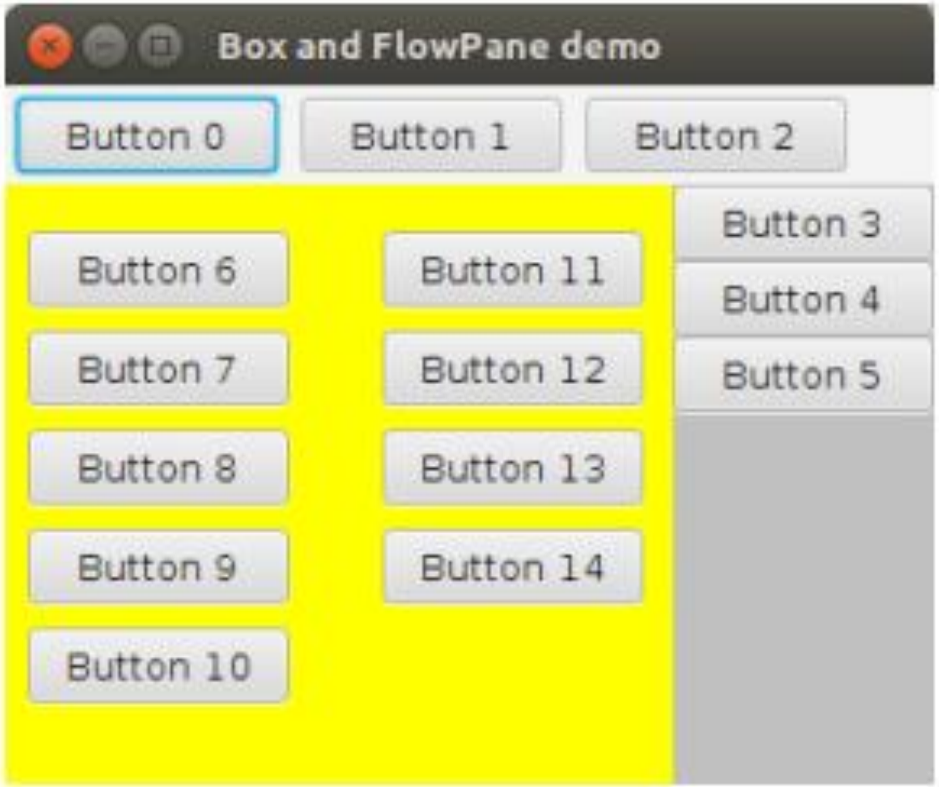
- Calling method **add(Node n)** of that list adds nodes to the pane

- ObservableList allows listeners to be executed whenever the list changes

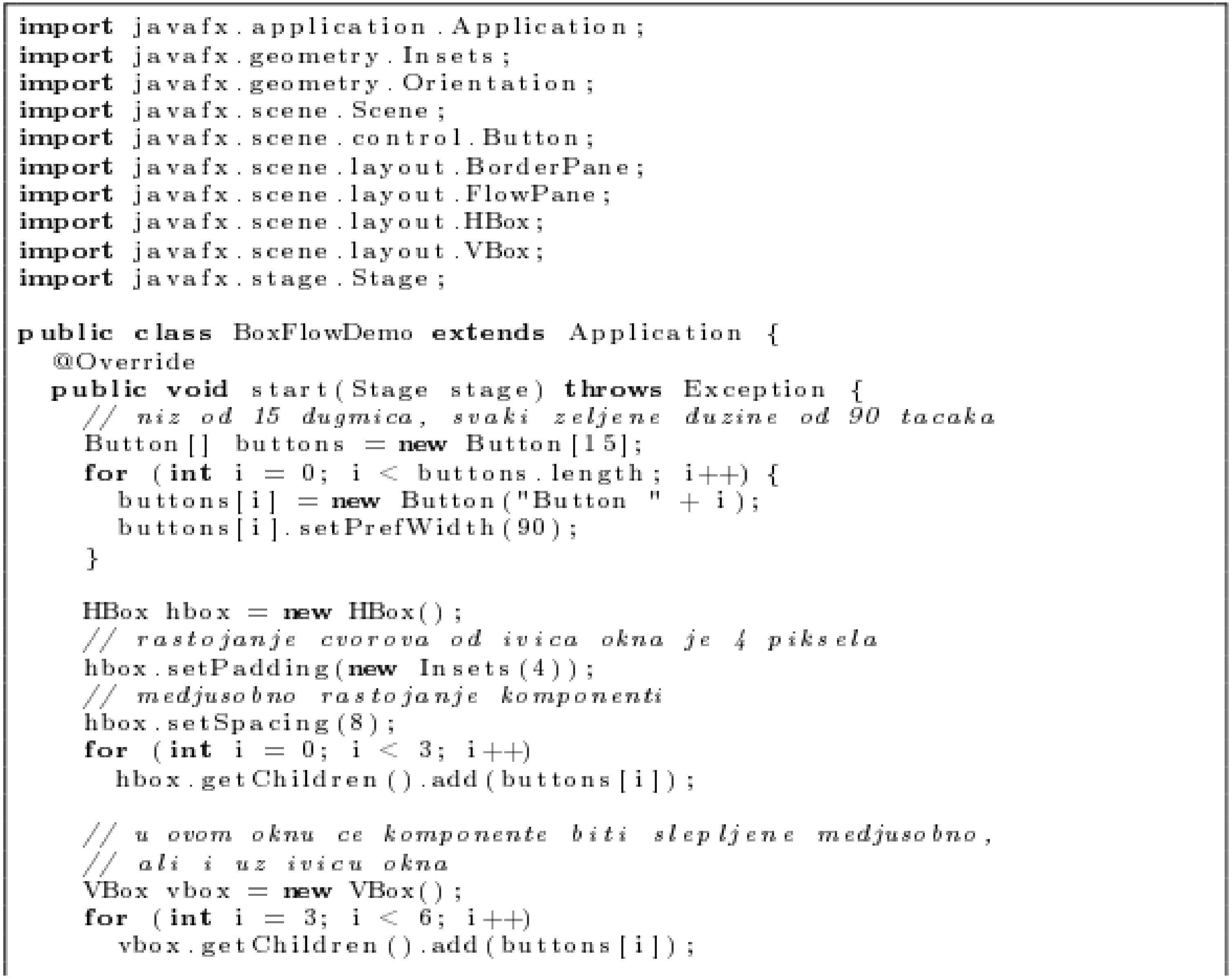
HBox, VBox and FlowPane

- By default, nodes are “stuck together”
- In case of HBox and VBox spacing is set by method **setSpacing**
- In case of FlowPane, it is possible to separately set gorizontal and vertical gaps by calling methods **setHgap** and **setVgap**
- To separate a node from the pane border, there is method **setPadding** taking one parameter of type Insets
- Since panes are nodes themselves, they can be put in other panes

Example – Hbox, VBox and FlowPane

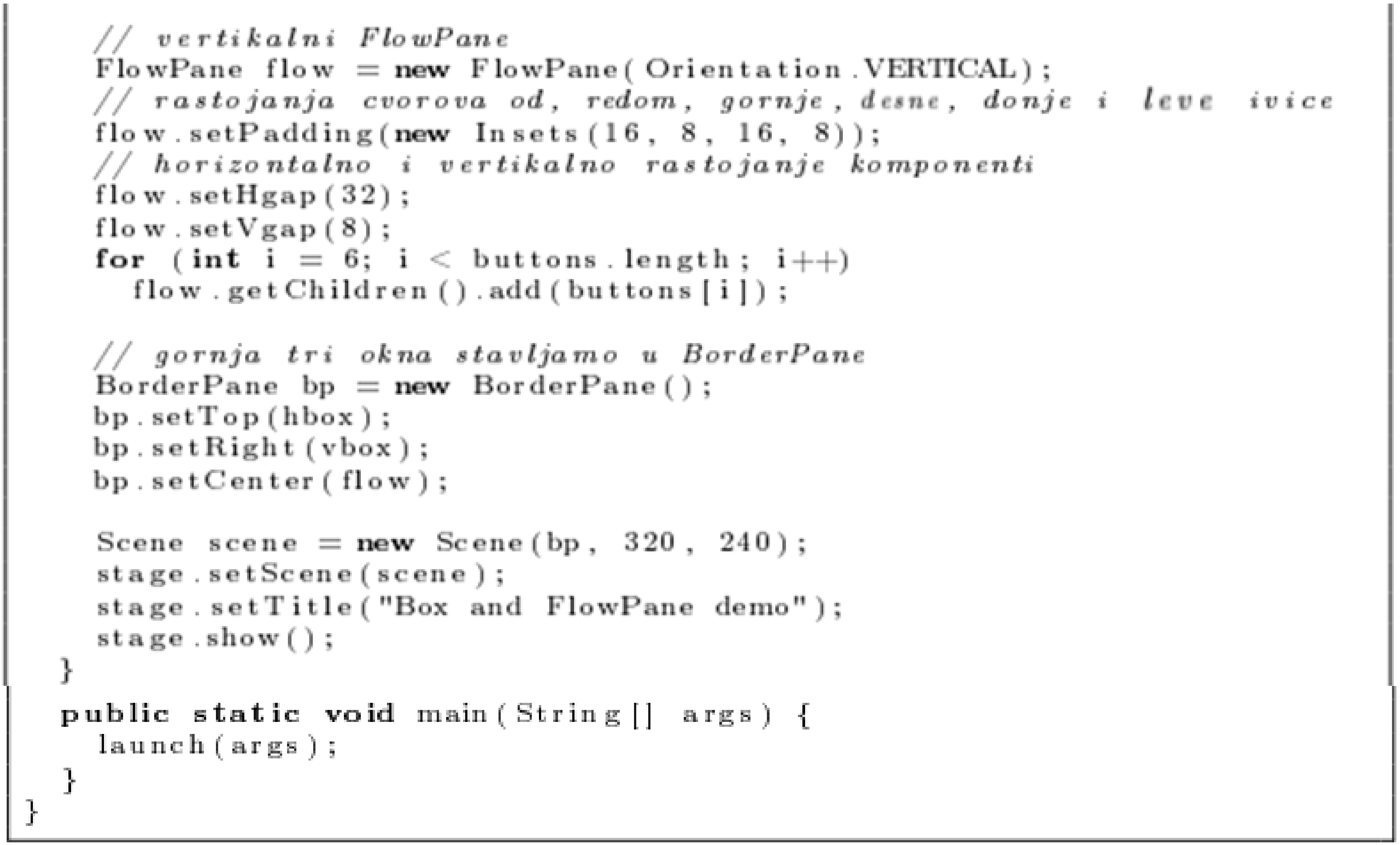


Example – Hbox, VBox and FlowPane



```java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BoxFlowDemo extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    // niz od 15 dugmica, svaki zeljene duzine od 90 tacaka
    Button[] buttons = new Button[15];
    for (int i = 0; i < buttons.length; i++) {
      buttons[i] = new Button("Button " + i);
      buttons[i].setPrefWidth(90);
    }

    HBox hbox = new HBox();
    // rastojanje cvorova od ivica okna je 4 piksela
    hbox.setPadding(new Insets(4));
    // medjusobno rastojanje komponenti
    hbox.setSpacing(8);
    for (int i = 0; i < 3; i++)
      hbox.getChildren().add(buttons[i]);

    // u ovom oknu ce komponente biti slepljene medjusobno,
    // ali i uz ivicu okna
    VBox vbox = new VBox();
    for (int i = 3; i < 6; i++)
      vbox.getChildren().add(buttons[i]);
```

Example – Hbox, VBox and FlowPane



```
    // vertikalni FlowPane
    FlowPane flow = new FlowPane(Orientation.VERTICAL);
    // rastojanja cvorova od, redom, gornje, desne, donje i leve ivice
    flow.setPadding(new Insets(16, 8, 16, 8));
    // horizontalno i vertikalno rastojanje komponenti
    flow.setHgap(32);
    flow.setVgap(8);
    for (int i = 6; i < buttons.length; i++)
      flow.getChildren().add(buttons[i]);

    // gornja tri okna stavljamo u BorderPane
    BorderPane bp = new BorderPane();
    bp.setTop(hbox);
    bp.setRight(vbox);
    bp.setCenter(flow);

    Scene scene = new Scene(bp, 320, 240);
    stage.setScene(scene);
    stage.setTitle("Box and FlowPane demo");
    stage.show();
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

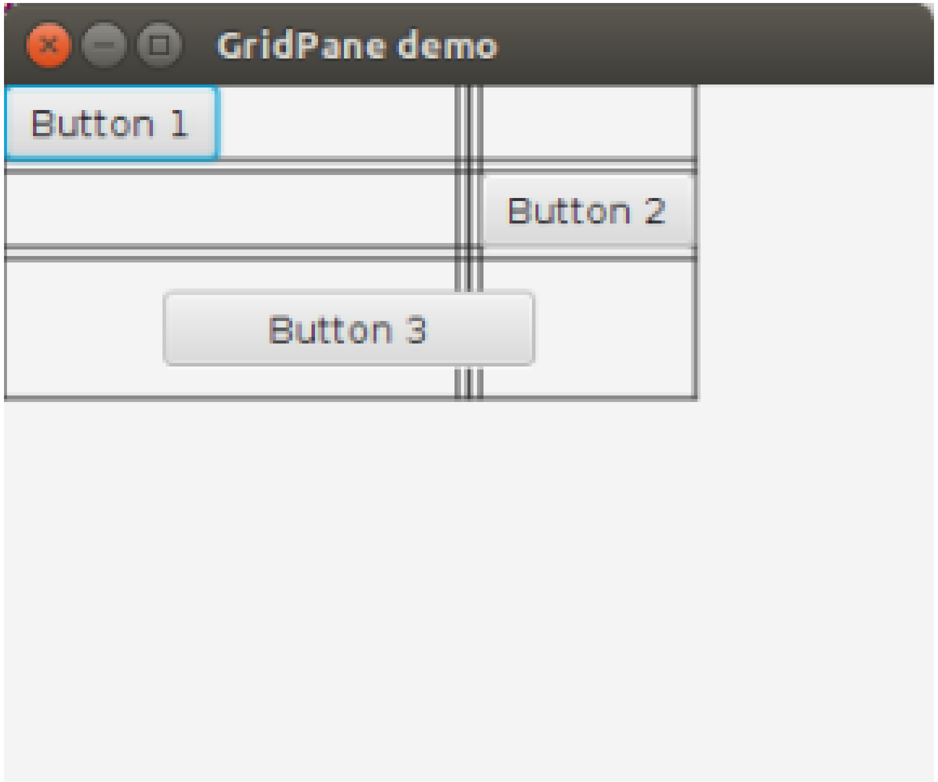
GridPane

- GridPane represents a table
- A node can be put into an arbitrary table cell, and can span multiple rows and columns
- The table grows automatically as nodes are added
- One cell can contain multiple nodes, but they can overlap
- Row and column indices start at 0

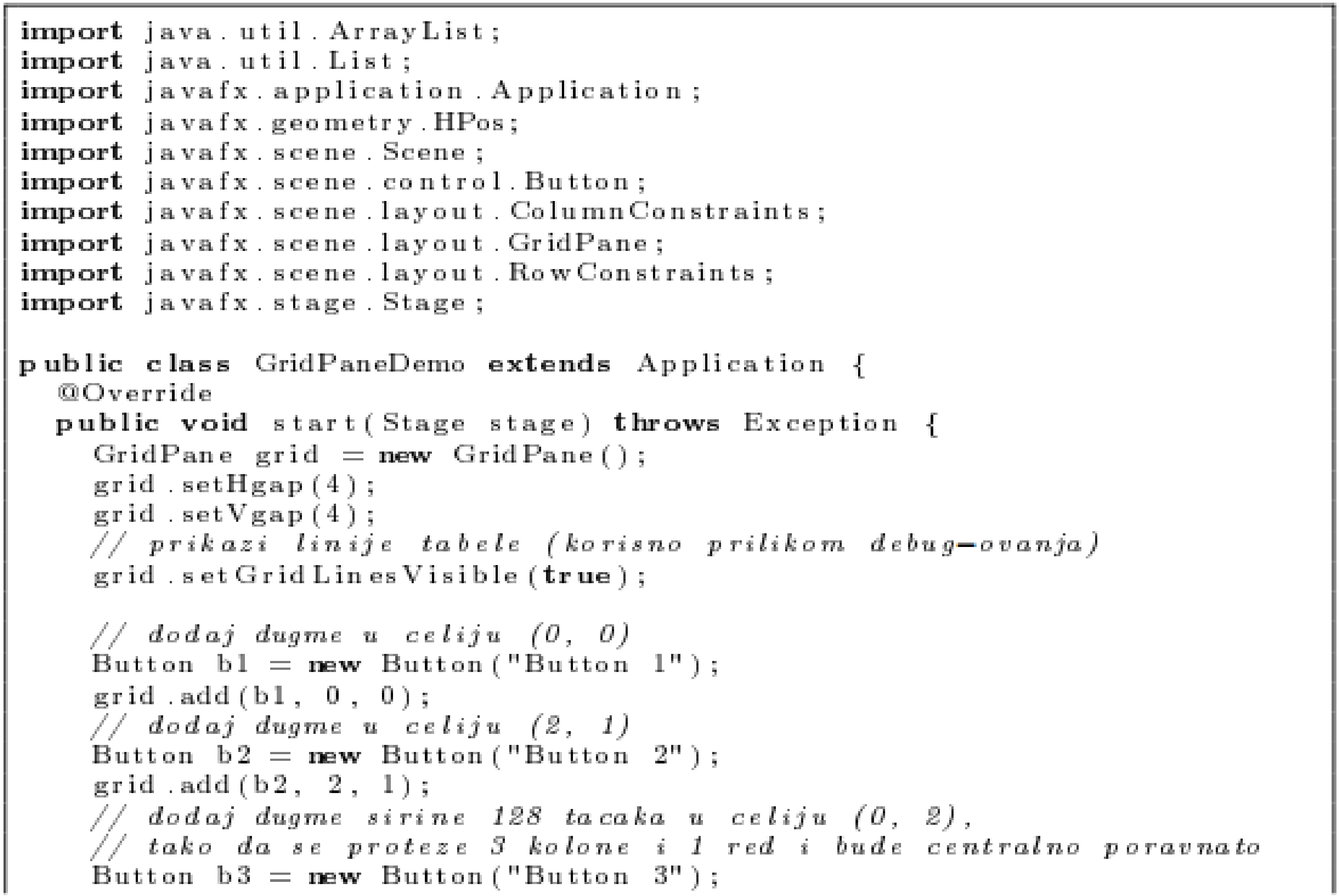
GridPane

- GridPane tries to set node sizes to their preferred values, and shrinks them if needed
- If a node’s minimal size is too large, it can “spill” out of the cell
- By default, a cell has the height of the highest node in the same row, and width of the widest node in the same column
- It is possible to manually set column widths and row heights by manipulating ObservableList objects returned by methods **getColumnConstraints** and **getRowConstraints** of GridPane

Example – using GridPane

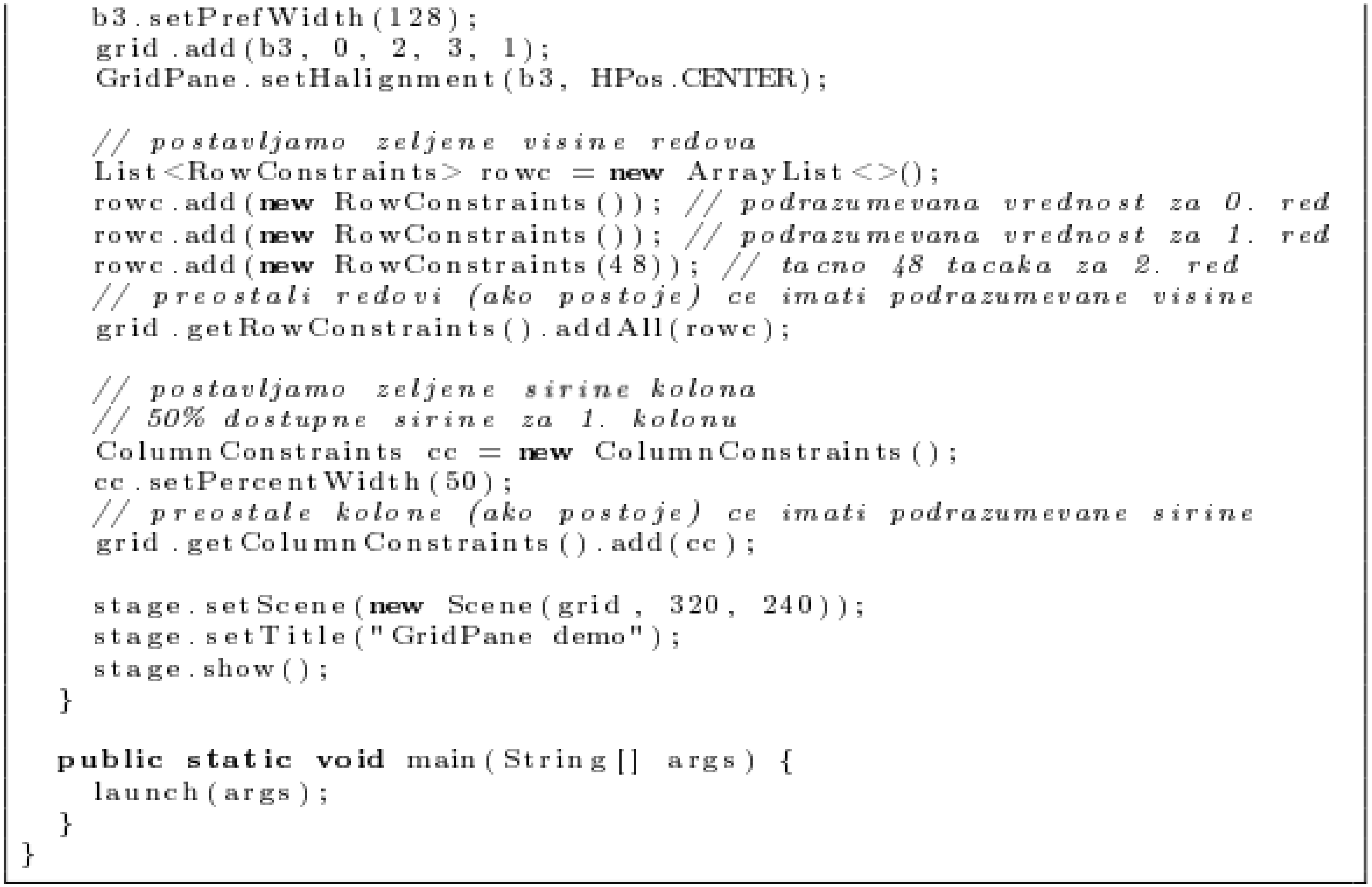


Example – using GridPane



```
import java.util.ArrayList;
import java.util.List;
import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.RowConstraints;
import javafx.stage.Stage;

public class GridPaneDemo extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    GridPane grid = new GridPane();
    grid.setHgap(4);
    grid.setVgap(4);
    // prikazi linije tabele (korisno prilikom debug-ovanja)
    grid.setGridLinesVisible(true);

    // dodaj dugme u celiju (0, 0)
    Button b1 = new Button("Button 1");
    grid.add(b1, 0, 0);
    // dodaj dugme u celiju (2, 1)
    Button b2 = new Button("Button 2");
    grid.add(b2, 2, 1);
    // dodaj dugme sirine 128 tacaka u celiju (0, 2),
    // tako da se proteze 3 kolone i 1 red i bude centralno poravnato
    Button b3 = new Button("Button 3");
```

Example – using GridPane

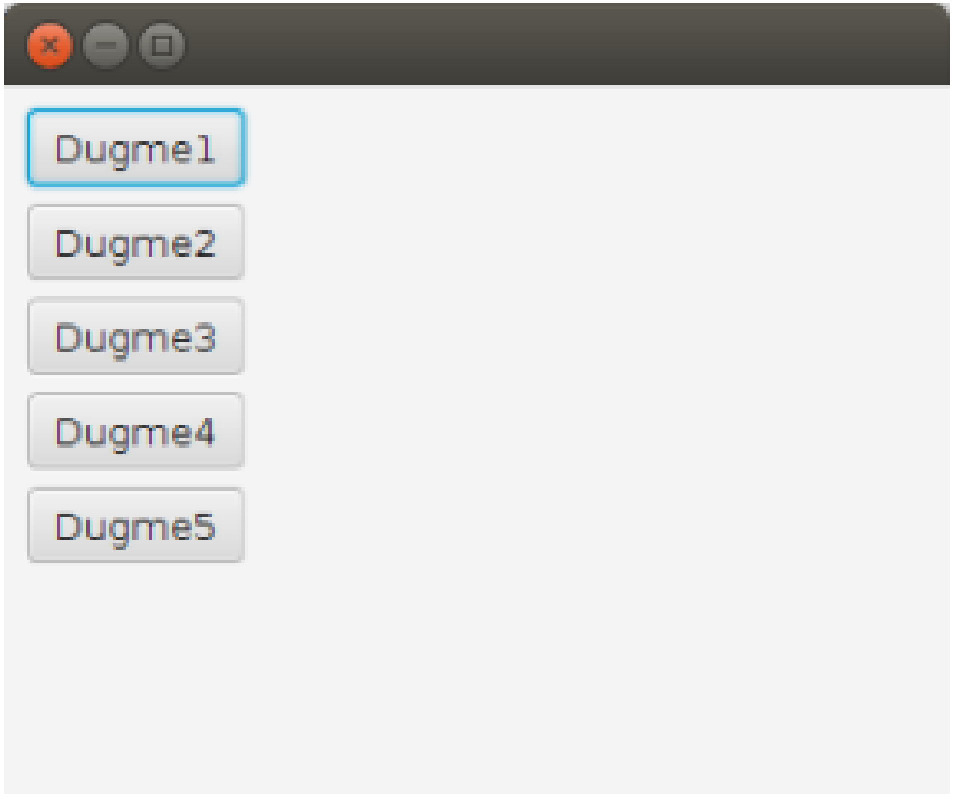


```
    b3.setPrefWidth(128);
    grid.add(b3, 0, 2, 3, 1);
    GridPane.setHalignment(b3, HPos.CENTER);

    // postavljamo zeljene visine redova
    List<RowConstraints> rowc = new ArrayList<>();
    rowc.add(new RowConstraints()); // podrazumevana vrednost za 0. red
    rowc.add(new RowConstraints()); // podrazumevana vrednost za 1. red
    rowc.add(new RowConstraints(48)); // tacno 48 tacaka za 2. red
    // preostali redovi (ako postoje) ce imati podrazumevane visine
    grid.getRowConstraints().addAll(rowc);

    // postavljamo zeljene sirine kolona
    // 50% dostupne sirine za 1. kolonu
    ColumnConstraints cc = new ColumnConstraints();
    cc.setPercentWidth(50);
    // preostale kolone (ako postoje) ce imati podrazumevane sirine
    grid.getColumnConstraints().add(cc);

    stage.setScene(new Scene(grid, 320, 240));
    stage.setTitle("GridPane demo");
    stage.show();
  }

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

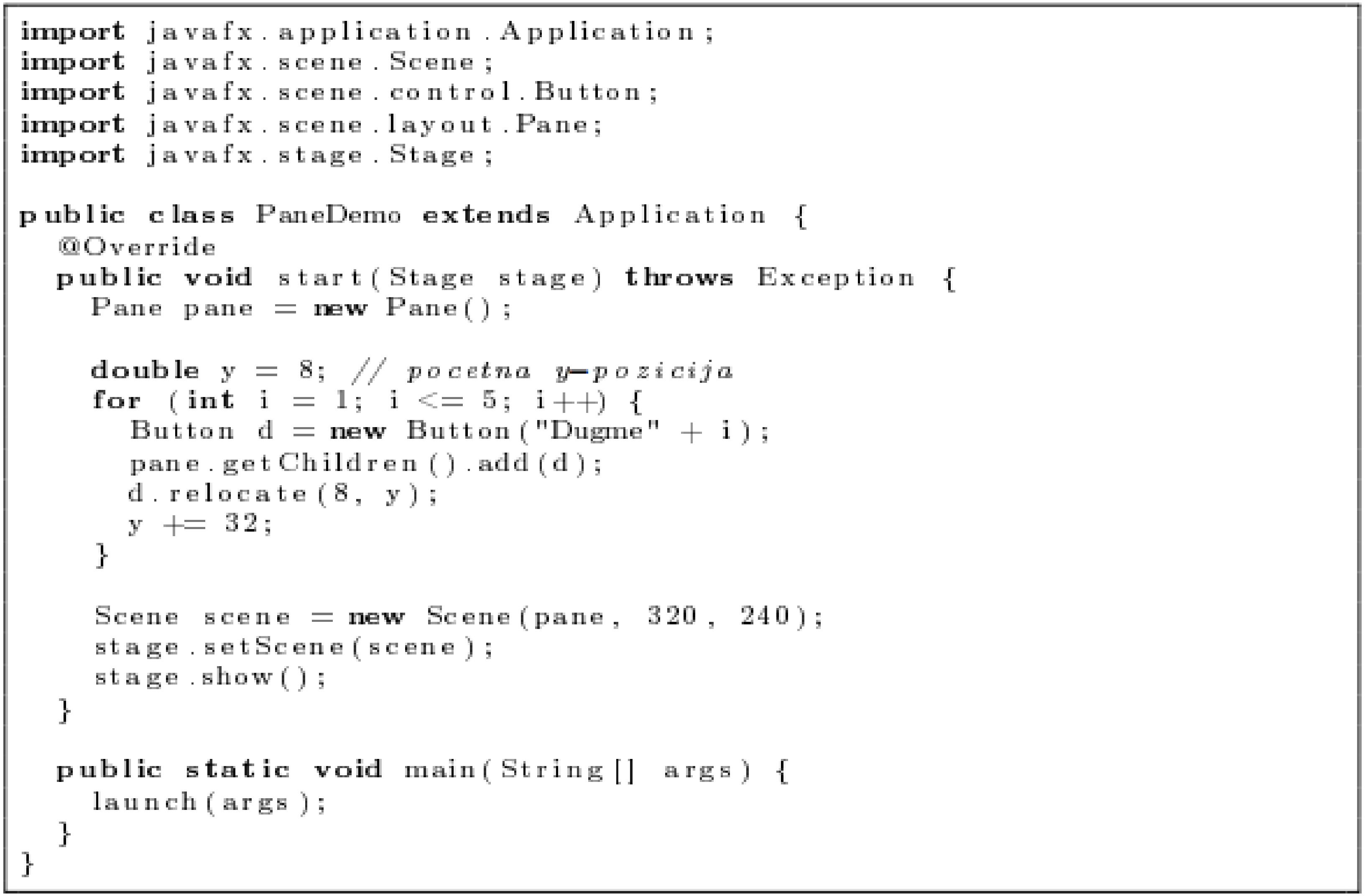
Pane

- Pane – we manually set the (fixed) position and size of each node
- Parent class of all other panes
- Node sizes will be set to preferred values and will not be changed
- The position of each node will be set to (0, 0), the upper-left corner
- New position – calling node method **relocate** taking x and y coordinates, or methods **setLayoutX** i **setLayoutY** for individual setting of x and y

Example – using Pane



Example – using Pane



```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class PaneDemo extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Pane pane = new Pane();

    double y = 8; // pocetna y-pozicija
    for (int i = 1; i <= 5; i++) {
      Button d = new Button("Dugme" + i);
      pane.getChildren().add(d);
      d.relocate(8, y);
      y += 32;
    }

    Scene scene = new Scene(pane, 320, 240);
    stage.setScene(scene);
    stage.show();
  }

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

Creating GUI with JavaFX

- JavaFX application structure
- User interface design
- Event-based programming

Event-based programming

- Execution flow of GUI applications is organized using events
- After startup, the application waits for an event such as mouse click or key stroke, and executes some statements as a reaction to that event
- This model of developing GUI applications is widely known as **event-based programming** (or event-**driven** programming)
- In Java, three important components of an event can be identified:
  - *Event source*: component that generates the event (e.g. button, mouse, keyboard, etc.)
  - *Event listener (handler)*: an object that receives news of events and processes them
  - *Event object*: contains all necessary information about the event that has occurred (type, source, etc.)

Event types and handling

- A JavaFX application can handle different types of events, represented by appropriate classes
- Classes are organized into a hierarchy with root class **javafx.event.Event**
- Some of the most important subclasses:
    - **KeyEvent**: used for key (de)presses
    - **MouseEvent**: when the mouse is moved or mouse button clicked
    - **MouseDragEvent**: when the mouse is dragged
    - **ActionEvent**: specialized event, denotes that the used pushed a button, opened/closed a combo box, selected a menu entry, etc.
    - **WindowEvent**: application window is shown, hidden or closed

Event types and handling

- In addition, JavaFX has built-in support for modern devices with touch screens, through classes:
  - **TouchEvent**, **SwipeEvent** and **ZoomEvent**
- Within each Event class there can be multiple event types, represented by generic class **EventType<T extends Event>**, that is, a field of that type, and constants (final fields) defining all possible event types
- For example, a user can move the mouse or push one of its buttons – these are different types of MouseEvent

Event types and handling

- In order to be notified of an event, it is necessary to associate an event-handling method to the appropriate node
- For example, when the used clicks a button, JavaFX will collect information about the event and forward it to our method
- The collected information, in case of a click, will include the mouse cursor position, as well as which button was clicked, and how many times
- This general approach is used for all types of events

Keyboard events

- JavaFX offers three event types for keyboard handling:
  - KeyEvent.KEY_PRESSED: the user pushed (and holds) a key
  - KeyEvent.KEY_RELEASED: the user released a key
  - KeyEvent.KEY_TYPED: the user typed a key
  - All above constants are of type EventType<KeyEvent>
- The last event type is generated in case the used entered a character either by pressing an alphanumeric key or holding the Alt key and entering the character code on the numeric keypad

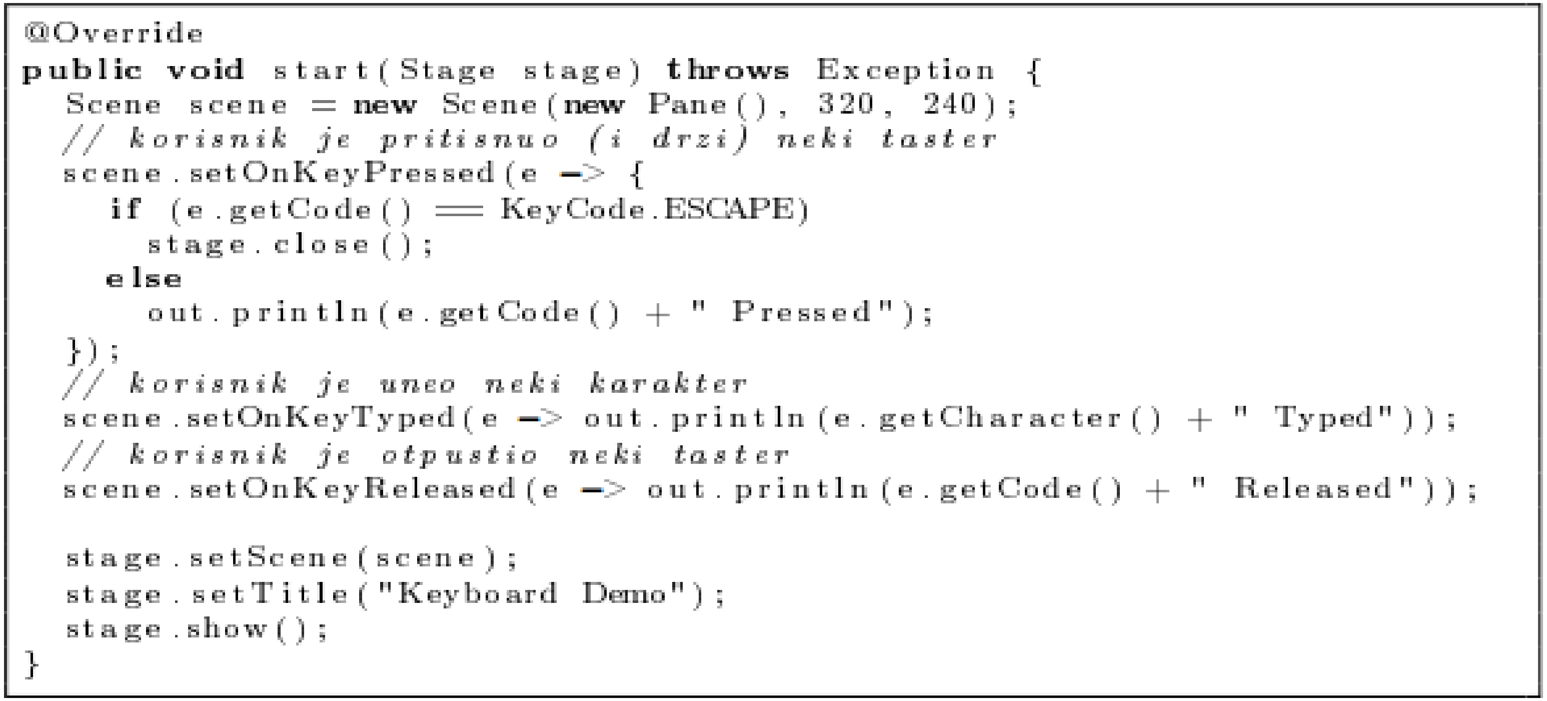
Keyboard events

- Information about the event is collected into a KeyEvent object

- The most important method is **getCode()** which returns the code of the (de)pressed key, represented by enum **KeyCode** whose constants denote keys

- For the **KEY_TYPED** event the **getCharacter()** should be used, returning the typed character as a string

Keyboard events



```java
@Override
public void start(Stage stage) throws Exception {
  Scene scene = new Scene(new Pane(), 320, 240);
  // korisnik je pritisnuo (i drzi) neki taster
  scene.setOnKeyPressed(e -> {
    if (e.getCode() == KeyCode.ESCAPE)
      stage.close();
    else
      out.println(e.getCode() + " Pressed");
  });
  // korisnik je uneo neki karakter
  scene.setOnKeyTyped(e -> out.println(e.getCharacter() + " Typed"));
  // korisnik je otpustio neki taster
  scene.setOnKeyReleased(e -> out.println(e.getCode() + " Released"));

  stage.setScene(scene);
  stage.setTitle("Keyboard Demo");
  stage.show();
}
```

If we press ‘Shift+a’, the following will be printed:

```
SHIFT Pressed
A Pressed
A Typed
A Released
SHIFT Released
```

If we press only ‘a’, the following will be printed:

```
A Pressed
a Typed
A Released
```

Events KEY_PRESSED and KEY_RELEASED contain the unique key (not character) code (in this example, code of the A key), and thus cannot be directly used to distinguish lower and upper case letters

Event handler

- In order to define event handling methods, functional interface **EventHandler<T extends Event>** is used
  - Method: **void handle(T event)**
- In the preceding example, **T** is **KeyEvent**
- Therefore:
  - The lambda expression passed to **scene.setOnKeyPressed()** is of type **EventHandler<KeyEvent>**
  - Argument **e** is of type **KeyEvent**

Mouse events

JavaFX offers multiple classes for handling mouse events:

- MouseEvent covers the basic mouse operations: clicking, moving the cursor and dragging the mouse inside a single node
- MouseDragEvent is used when dragging includes multiple nodes
- DragEvent is generated when the user drags content from another (not necessarily Java) application into the current JavaFX application window
- ScrollEvent is generated when the user moves the mouse wheel

Mouse events

- Class MouseEvent contains multiple types of mouse events, including:
  - MOUSE_PRESSED: the user pressed (and holds) a mouse button
  - MOUSE_RELEASED: the user released the mouse button
  - MOUSE_CLICKED: the user clicked a mouse button
  - MOUSE_ENTERED: the mouse cursor entered the node
  - MOUSE_EXITED: the mouse cursor exited the node
  - MOUSE_DRAGGED: the user is dragging the mouse
- The MouseEvent object collects various information about the event, including:
  - Cursor position, which button was clicked and how many times, did the user also press a control key (Shift, Alt, Ctrl), etc.

Mouse events

- Mouse buttons are denoted by enum MouseButton constants:
  - PRIMARY
  - SECONDARY
  - MIDDLE
  - NONE
- The primary and secondary buttons depend on user settings in the operating system
- The middle button is usually the wheel
- NONE is used when no button is pressed

Mouse events

- Three different pairs of coordinates can be used to determine mouse cursor position
  - X and Y represent the position within the node w.r.t. its upper-left corner
  - SceneX and SceneY are cursor coordinates relative to the application scene (i.e. its upper-left corner),
  - ScreenX and ScreenY are defined w.r.t. the screen in which the application window is displayed

Mouse events

```
public void start(Stage stage) throws Exception {
  Canvas canvas = new Canvas(320, 240);
  final GraphicsContext gc = canvas.getGraphicsContext2D();
  gc.setStroke(Color.BLACK);
  // postavljamo inicijalnu poziciju olovke kada korisnik
  // pritisne levi taster
  canvas.setOnMousePressed(e -> {
    if (e.getButton() == MouseButton.PRIMARY) {
      gc.beginPath();
      gc.moveTo(e.getX(), e.getY());
    }
  });
  // u zavisnosti od toga koji taster je pritisnut,
  // prevlacenje moze crtati liniju ili simulirati gumicu
  canvas.setOnMouseDragged(e -> {
    if (e.getButton() == MouseButton.PRIMARY) {
      gc.lineTo(e.getX(), e.getY());
      gc.stroke();
    }
    else if (e.getButton() == MouseButton.SECONDARY)
      gc.clearRect(e.getX() - 1, e.getY() - 1, 3, 3);
  });
  // dupli klik levim tasterom brise ceo Canvas
  canvas.setOnMouseClicked(e -> {
    if (e.getButton() == MouseButton.PRIMARY && e.getClickCount() == 2)
      gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
  });
  BorderPane pane = new BorderPane();
  pane.setCenter(canvas);
  Scene scene = new Scene(pane);
  stage.setScene(scene);
  stage.setTitle("Paint Demo");
  stage.show();
}
```

Window and action events

- A JavaFX application can also react to events generated by the application window itself

- The WindowEvent class defines several event types:
  - WINDOW_CLOSE_REQUEST: the user wants to close the window
  - WINDOW_HIDING: generated before window minimization starts
  - WINDOW_HIDDEN: the window is minimized
  - WINDOW_SHOWING: generated before the window is displayed on the screen
  - WINDOW_SHOWN: the window is displayed on the screen

Example – annoying windows

```
public void start(Stage stage) throws Exception {
  Button btnClose = new Button("Click me to Close");
  final double Y1 = 40, Y2 = 160;
  btnClose.setPrefSize(260, 40);
  btnClose.relocate(30, Y1);
  // ako korisnik ipak nekako klikne na dugme, zatvoricemo prozor
  btnClose.setOnMouseClicked(e -> {
    if (e.getButton() == MouseButton.PRIMARY && e.getClickCount() == 1)
      stage.close();
  });
  // sakrivamo dekoracije prozora, ukljucujuci i dugmice minimize,
  // maksimize i close
  stage.initStyle(StageStyle.UNDECORATED);
  // ako korisnik pokusa da zatvori prozor (Alt+F4), ignorisacemo ga
  stage.setOnCloseRequest(e -> e.consume());
  // ako kursor misa udje u dugme sa gornje strane, dugme pomeramo
  // gore, i obrnuto
  btnClose.setOnMouseMoved(e -> {
    boolean gore = Math.abs(btnClose.getLayoutY() - Y1) < 0.001;
    double newY = gore ? Y2 : Y1;
    btnClose.setLayoutY(newY);
  });
  Pane root = new Pane(btnClose);
  stage.setScene(new Scene(root, 320, 240));
  stage.setTitle("Annoying Window");
  stage.setResizable(false);
  stage.show();
}
```

Phases in event handling

- JavaFX event handling is performed in several phases
- First, when an event happens, all relevant information is collected and put into the appropriate event object
- Then, the target object is selected
- The target object can be: the stage, the scene or a node
- For example:
  - Keyboard event – the target object is the node currently in focus
  - Mouse event – the target object is the node currently under the mouse cursor
- When the target object is selected, an event dispatch chain is constructed starting from the stage, all the way to the target object

Phases in event handling

- The event is then sent through the chain, this is the event capturing phase
- During this phase, any node in the chain, including the target object, can filter the event
- A filter is a processing function, which can also consume the event, preventing it from being passed on
- The event capturing phase ends when:
  - a filter in the chain consumes the event, or
  - the event passes through the whole chain

Phases in event handling

- Finally, the event bubbling phase is executed
- In this phase, the event is passed to event handling methods
  - First to the target object method, then parent node method, and so on, back up the hierarchy to the stage
- Any event handling method can stop further advance of the event up the hierarchy by calling method consume()