Unit-06

Exception Handling

Outline

- ✔ Exception
- ✔ Using try and catch
- ✔ Nested try statements
- ✔ The Exception Class Hierarchy
- ✔ throw statement
- ✔ throws statement

Exceptions

- An **exception** is an object that describes an **unusual** or **erroneous situation**.
- Exceptions are thrown by a program, and may be caught and handled by another part of the program.
- A program can be separated into a **normal execution flow** and an **exception execution flow**.
- An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught.
- Java has a predefined set of exceptions and errors that can occur during execution.
- A program can deal with an exception in one of three ways:
  - **ignore** it
  - **handle** it where it occurs
  - handle it at **another place** in the program
- The manner in which an exception is processed is an important design consideration.

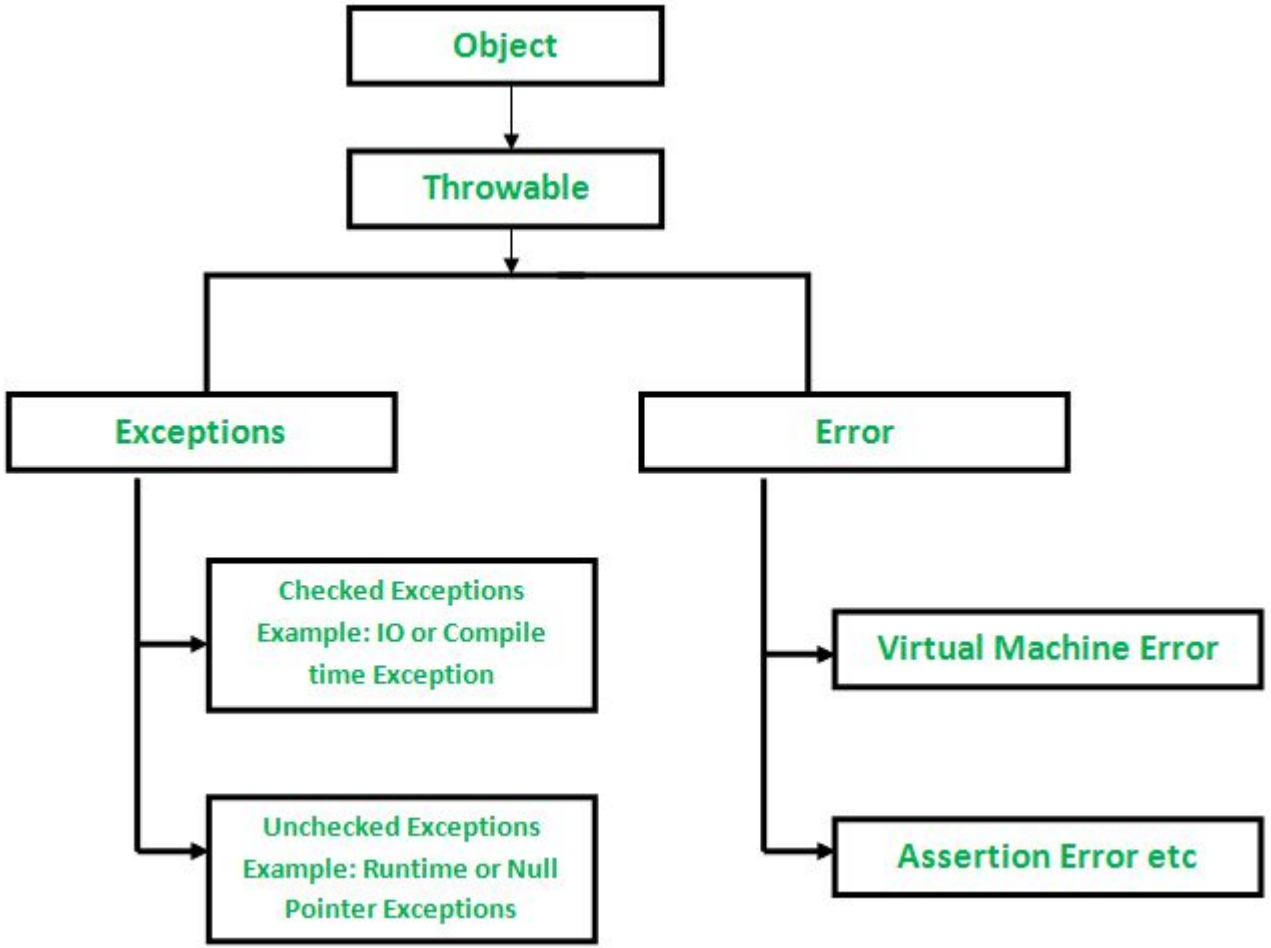
Exceptions in Java

- Exception Handling in Java – An effective means **to handle the runtime errors**
- By this - Regular flow of the application can be preserved
- An exception is **an unwanted or unexpected event**, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object - called the **exception object**.
- contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.

Exceptions vs Error

- Exceptions : Indicates condition that a reasonable application might try to catch
- Error : Indicates a serious problem that a reasonable application should not try to catch
  - Irrecoverable conditions like JVM running out of memory, stack overflow, library incompatibility etc.
  - Beyond control of programmer and we should not try to handle it

Exceptions vs Error

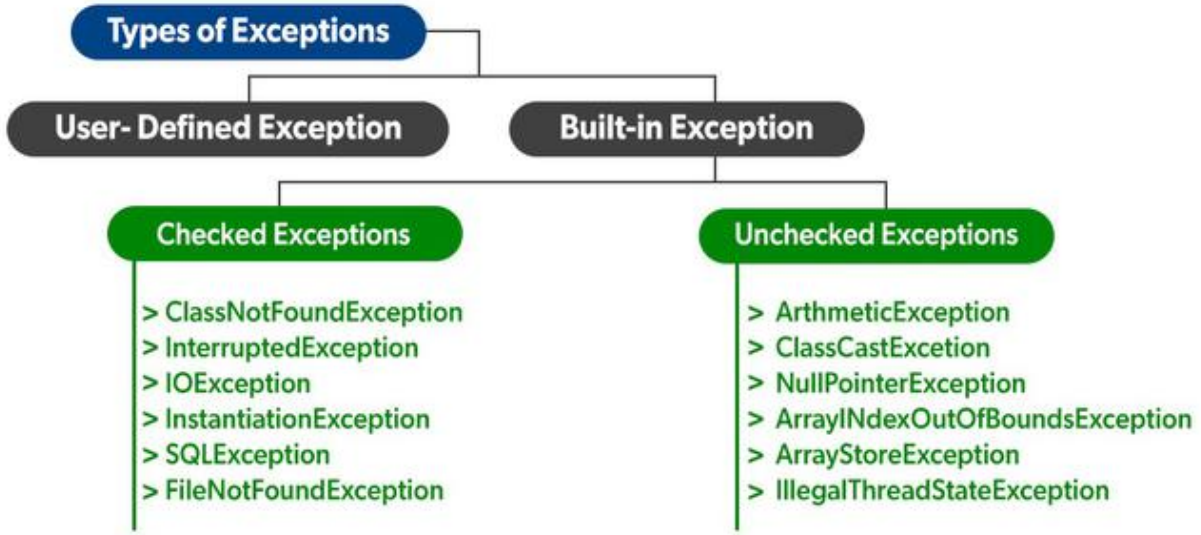


Exceptions in Java

- Exception can occur for many reasons
  - Invalid User Input
  - Device Failure
  - Network Connection Failure
  - Code Error
  - Opening an unavailable file
  - Physical Limitation (out of disk space)

Type of Exceptions

- Built-In Exceptions – available in Java libraries
  - **Checked Exceptions** – compile-time exceptions
  - **Un-checked Exceptions** – not checked @ compile time : if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error
- User-Defined Exceptions - Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions which are called '**user-defined Exceptions**'.



Type of Exceptions

Checked Exception

- The classes that directly **inherit the Throwable class except RuntimeException and Error are known as checked exceptions**. For example, **IOException, SQLException**, etc.
- Checked exceptions are checked at compile-time.

Unchecked Exception

- The classes that **inherit the RuntimeException are known as unchecked exceptions**. For example, **ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException**, etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error

- Error is **irrecoverable**. Some example of errors are **OutOfMemoryError, VirtualMachineError, AssertionError** etc

How does JVM handle an Exceptions ?

- **Default Exception Handling:** Whenever **inside a method**, if an **exception has occurred**, the **method creates an Object** known as **Exception Object** and **hands it off to the run-time system(JVM)**.

- The exception object contains the name and description of the exception and the current state of the program where the exception has occurred.
- Creating the Exception Object and handling it to the run-time system is called **throwing** an Exception.
- There might be a list of the methods that had been called to get to the method where an exception occurred.
- This ordered list of the methods is called **Call Stack**

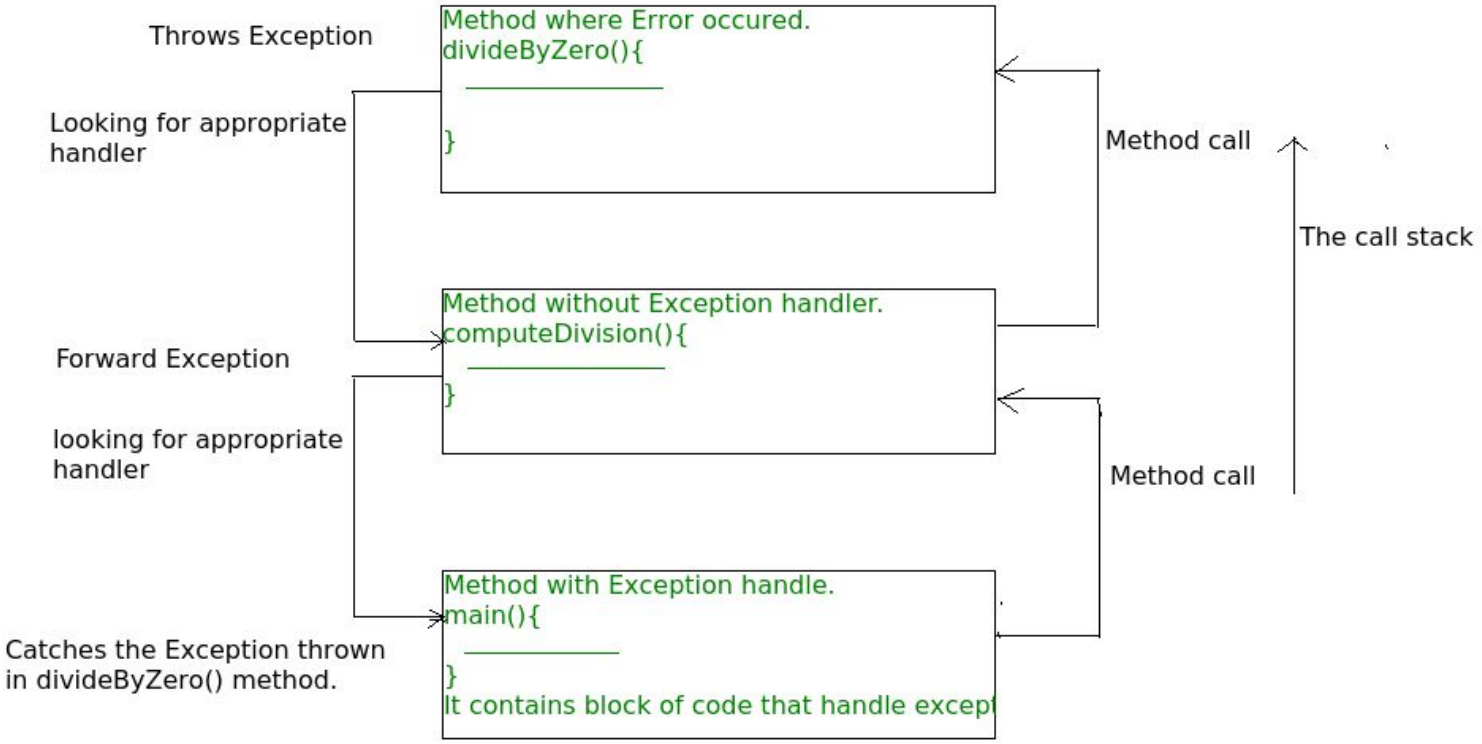
How does JVM handle an Exceptions ?

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred, proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

How does JVM handle an Exceptions ?

```
class ExceptionThrown {
    static int divideByZero(int a, int b){
        // this statement will cause ArithmeticException(/ by zero)
        int i = a/b;
        return i;
    }
    static int computeDivision(int a, int b) {
        int res =0;
        try {
            res = divideByZero(a,b);
        }
        // doesn't matches with ArithmeticException
        catch(NumberFormatException ex) {
            System.out.println("NumberFormatException is occurred");
        }
        return res;
    }
    // In this method found appropriate Exception handler.
    // i.e. matching catch block.
    public static void main(String args[]){
        int a = 1;
        int b = 0;
        try {
            int i = computeDivision(a,b);
        }
        // matching ArithmeticException
        catch(ArithmeticException ex) {
            // getMessage will print description
            // of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
}
```

How does JVM handle an Exceptions ?



The call stack and searching the call stack for exception handler.

How programmer handles an Exception ?

- **Customized Exception Handling:** Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Briefly, here is how they work.
- Program statements that you think can raise exceptions are contained within a **try** block.
- If an exception occurs within the try block, it is **thrown**.
- Your code can **catch** this exception (using catch block) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- **To manually throw an exception, use the keyword throw.**
- Any exception that is thrown out of a method must be specified as such by a throws clause.
- Any code that absolutely must be executed after a try block completes is put in a finally block.

How does JVM handle an Exceptions ?

```
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// optional
finally {
    // block of code to be executed after try block ends
}
```

Points to remember :

- In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it.
- To associate exception handler, we must put a **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument.
- The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.

Points to remember :

- For each try block there can be zero or more catch blocks, but **only one** final block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not.
- If an exception occurs, then it will be executed after **try and catch blocks.**
- And if an exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Using try and catch

- Example:

```
try{
    // code that may cause exception
}
catch(Exception e){
    // code when exception occurred
}
```

Exception is the superclass of all the exception that may occur in Java

- Multiple catch:

```
try{
    // code that may cause exception
}
catch(ArithmeticException ae){
    // code when arithmetic exception occurred
}
catch(ArrayIndexOutOfBoundsException aiobe){
    // when array index out of bound exception occurred
}
```

Nested try statements

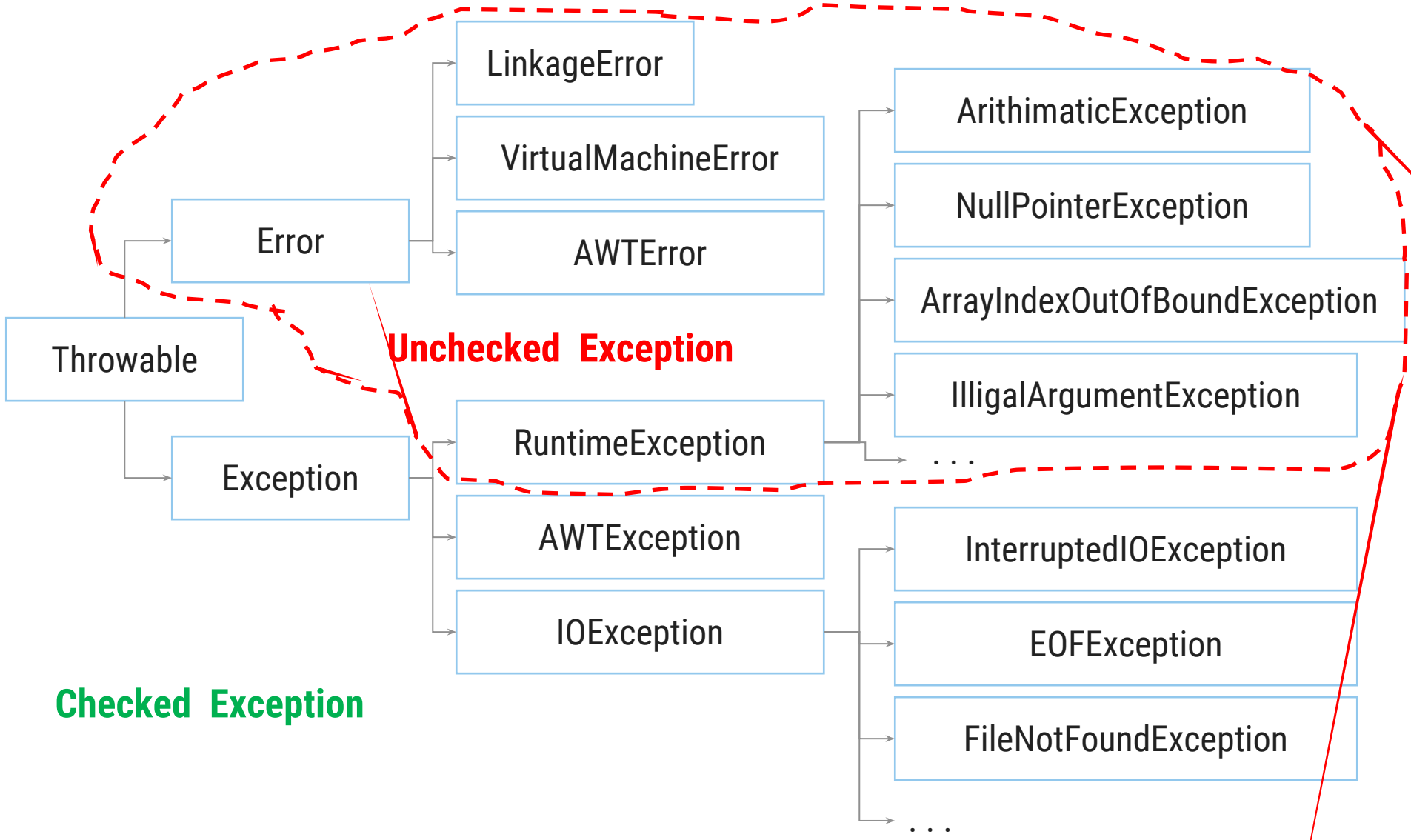
```
try
{
    try
    {
        // code that may cause array index out of bound exception
    }
    catch(ArrayIndexOutOfBoundsException aiobe)
    {
        // code when array index out of bound exception occured
    }
    // other code that may cause arithmetic exception
}
catch(ArithmeticException ae)
{
    // code when arithmetic exception occurred
}
```

Types of Exceptions

- An exception is either checked or unchecked.
- **Checked** Exceptions
  - A checked exception either must be caught by a method, or must be listed in the throws clause of any method that may throw or propagate it.
  - The compiler will issue an error if a checked exception is not caught or asserted in a throws clause
  - Example: IOException, SQLException etc…
- **Unchecked** Exceptions
  - An unchecked exception does not require explicit handling, though it could be processed using try catch.
  - The only unchecked exceptions in Java are objects of type RuntimeException or any of its descendants.
  - Example: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException etc..

The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy.
- All error and exception classes are descendents of the Throwable class
- The custom exception can be created by extending the Exception class or one of its descendants.



Java’s Inbuilt Unchecked Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Java’s Inbuilt Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
IOException	Input Output Exceptions
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

throw statement

- it is possible for your program to throw an exception **explicitly**, using the **throw** statement.
- The general form of throw is shown here:

  throw *ThrowableInstance;*

- Here, ***ThrowableInstance*** *must be an object of type* ***Throwable*** *or a* ***subclass*** *of Throwable.*
- Primitive types, such as int or char, as well as non-throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
  - using a parameter in a catch clause,
  - or creating one with the new operator.

Throw (Example)

```
public class DemoException {
    public static void main(String[] args) {
        int balance = 5000;

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
                throw new Exception("Balance must be grater than 1000");
            }
            else {
                balance = balance - withdraw;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

The finally statement

- The purpose of the **finally** statement will allow the execution of a segment of code regardless if the try statement throws an exception or executes successfully
- The advantage of the **finally** statement is the ability to clean up and release resources that are utilized in the **try** segment of code that might not be released in cases where an exception has occurred.

```
public class MainCall {
    public static void main(String args[]) {
        int balance = 5000;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
               throw new Exception("Balance < 1000 error");
            }
            else {
               balance = balance - withdraw;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            sc.close();
        }
    }
}
```

throws statement

- A throws statement lists the types of exceptions that a **method** might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a **throws clause:**

  type method-name(parameter-list) ***throws*** *exception-list* {

  // body of method

  }
- Here, *exception-list is a comma-separated list of the exceptions that a method can throw.*
- Example :

```
void myMethod() throws ArithmeticException, NullPointerException
{
    // code that may cause exception
}
```

Create Your Own Exception

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).
- The Exception class does not define any methods of its own. It does inherit those methods provided by Throwable.
- Thus, all exceptions have methods that you create and defined by Throwable.

Methods of Exception class

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements.
String toString( )	Returns a String object containing a description of the exception.

Custom Exception (Example)

```
// A Class that represents
user-defined exception
class MyException
    extends Exception
{
  public MyException(String s)
  {
     // Call constructor of
    parent (Exception)
        super(s);
  }
}
```

```
class MainCall {
    static int currentBal = 5000;
    public static void main(String args[]) {
        try {
        int amount = Integer.parseInt(args[0]);
            withdraw(amount);
        } catch (Exception ex) {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
    public static void withdraw(int amount) throws Exception
    {
        int newBalance = currentBal - amount;
        if(newBalance<1000) {
          throw new MyException("My Custom Exception … !!");
        }
    }
}
```

Unit-07

JavaFX and Event-driven programming and animations

Outline

- ✔ What is JavaFX?
- ✔ Architecture of JavaFX API
- ✔ JavaFX Application Structure
- ✔ Lifecycle of JavaFX Application
- ✔ 2D Shape
- ✔ JavaFX - Colors
- ✔ JavaFX – Image
- ✔ Layout Panes
- ✔ JavaFX – Events
- ✔ Property Binding
- ✔ Animation

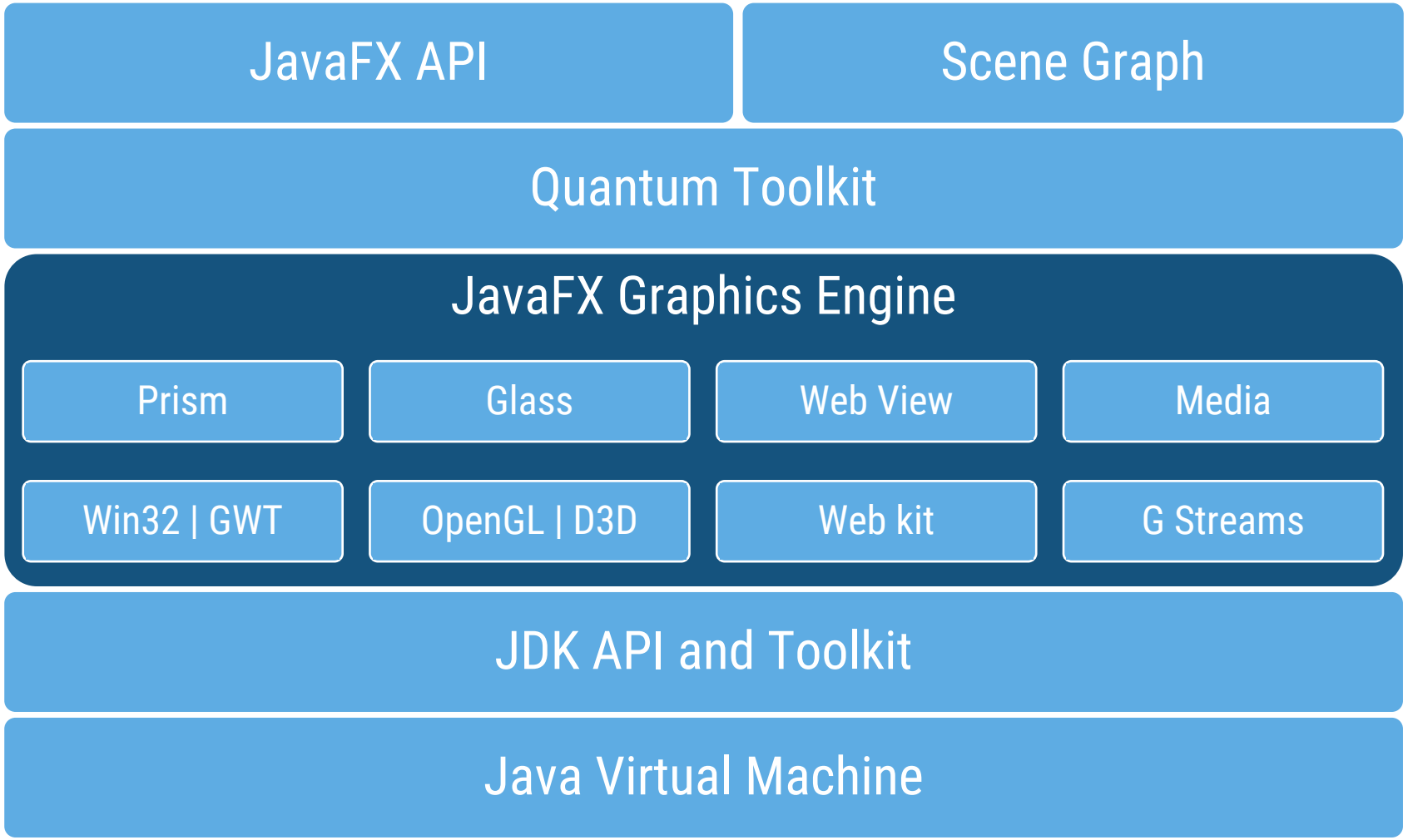
What is JavaFX?

- *JavaFX* is a Java library used to build Rich Internet Applications (RIA) and Desktop Applications.
- The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc.
- To develop GUI Applications using Java programming language, the programmers rely on libraries such as Advanced Windowing Toolkit (AWT) and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.
- Why we need JavaFX
  - To develop Client Side Applications with rich features, the programmers used to depend on various libraries to add features such as Media, UI controls, Web, 2D and 3D, etc.
  - JavaFX provides a rich set of graphics and media API's and it leverages the modern Graphical Processing Unit through hardware accelerated graphics.
  - One can use JavaFX with JVM based technologies such as Java, Groovy and JRuby. If developers opt for JavaFX, there is no need to learn additional technologies.

Features of JavaFX

- Written in Java
- FXML
- Scene Builder
- Swing Interoperability
- Built-in UI controls
- CSS like Styling
- Canvas and Printing API
- Rich set of API's
- Integrated Graphics library
- Graphics pipeline

Architecture of JavaFX API



Architecture of JavaFX API (Cont.)

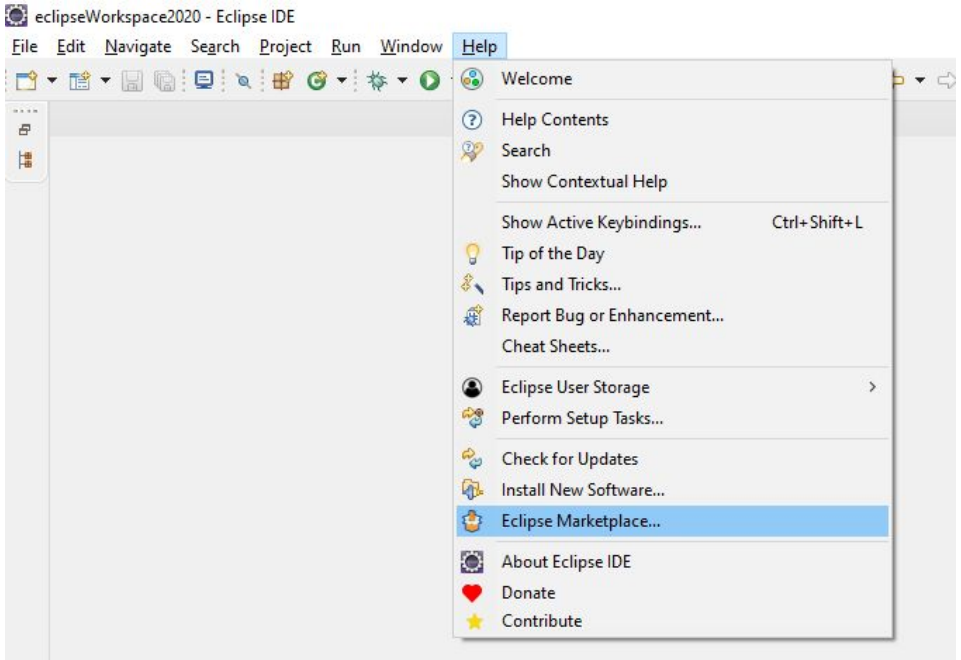
- Scene Graph
  - A Scene Graph is the starting point of the construction of the GUI Application. It holds the (GUI) application primitives that are termed as nodes.
  - A node is a visual/graphical object and it may include
    - Geometrical (Graphical) objects
    - UI controls
    - Containers
    - Media elements
- Prism
  - Prism is a high performance hardware–accelerated graphical pipeline that is used to render the graphics in JavaFX. It can render both 2-D and 3-D graphics.
- GWT (Glass Windowing Toolkit)
  - GWT provides services to manage Windows, Timers, Surfaces and Event Queues.
  - GWT connects the JavaFX Platform to the Native Operating System.

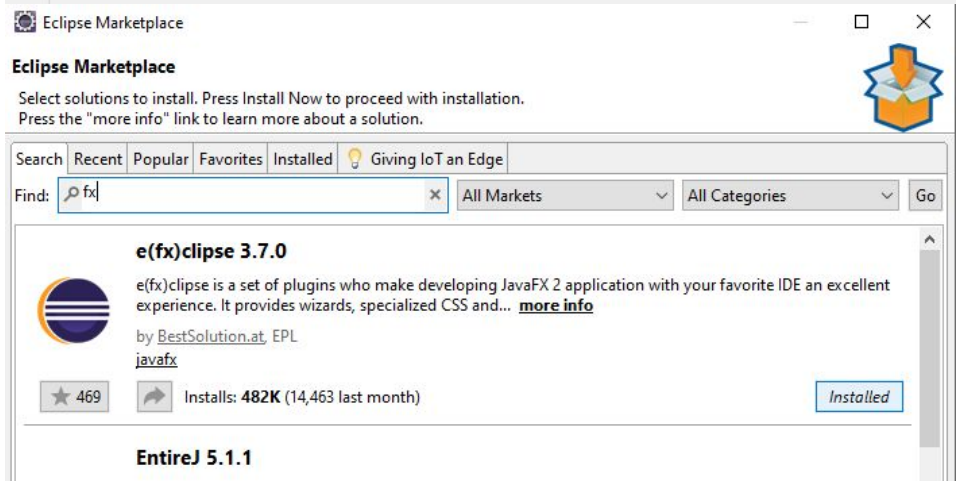
Architecture of JavaFX API (Cont.)

- Quantum Toolkit
  - It is an abstraction over the low-level components of Prism, Glass, Media Engine, and Web Engine. It ties Prism and GWT together and makes them available to JavaFX.
- WebView
  - WebView is the component of JavaFX which is used to process HTML content. It uses a technology called Web Kit, which is an internal open-source web browser engine. This component supports different web technologies like HTML5, CSS, JavaScript, DOM and SVG.
- Media Engine
  - The JavaFX media engine is based on an open-source engine known as a Streamer. This media engine supports the playback of video and audio content.

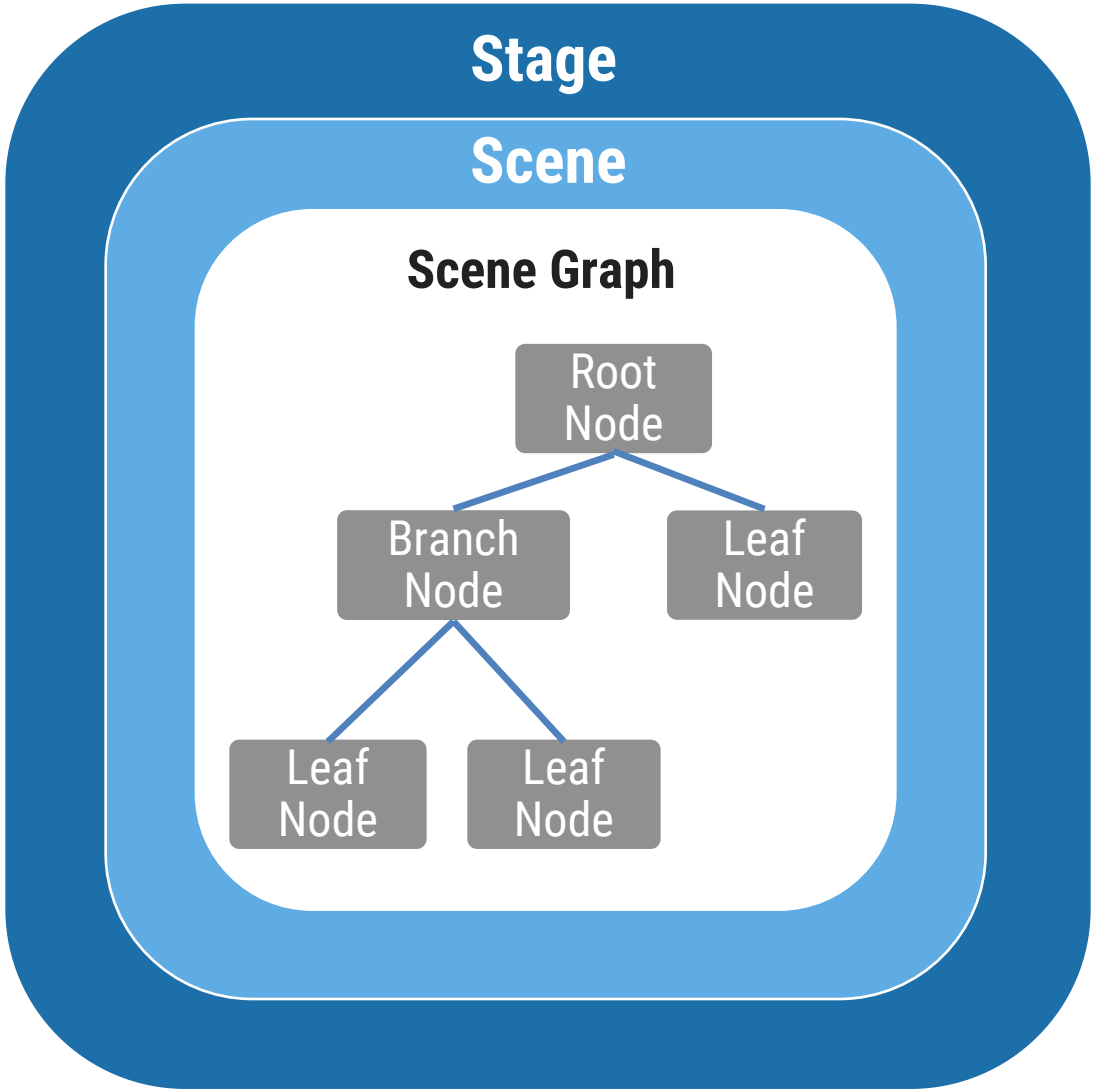
Installing JavaFX

- You can install JavaFX directly on eclipse using the Market Place.
- To download JavaFX from Market Place, open **`Help -> Eclipse Market Place`**
- Then simply search "**fx**" and install the e(fx)clipse from the Market Place.
- It will take time to install and after installation you need to restart the eclipse.
- Alternatively you can also download JavaFX SDK manually from https://gluonhq.com/products/javafx/
- Then you need to create a custom library in eclipse to load JavaFX to your project.





JavaFX Application Structure



Stage

- A stage (a window) contains all the objects of a JavaFX application.
- It is represented by `Stage` class of the package `javafx.stage`.
- The primary stage is created by the platform itself. The created stage object is passed as an argument to the `start()` method of the `Application` class.
- A stage has two parameters determining its position namely Width and Height.
- There are five types of stages available
  - Decorated `// stageObj.initStyle(StageStyle.DECORATED);`
  - Undecorated `// stageObj.initStyle(StageStyle.UNDECORATED);`
  - Transparent `// stageObj.initStyle(StageStyle.TRANSPARENT);`
  - Unified `// stageObj.initStyle(StageStyle.UNIFIED);`
  - Utility `// stageObj.initStyle(StageStyle.UTILITY);`
- You have to call the `show()` method to display the contents of a stage.

Scene

- A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph.
- The class `Scene` of the package `javafx.scene` represents the scene object. At an instance, the scene object is added to only one stage.
- You can create a scene by instantiating the `Scene` Class.
- You can opt for the size of the scene by passing its dimensions (height and width) along with the root node to its constructor.

Scene Graph and Nodes

- A scene graph is a tree-like data structure (hierarchical) representing the contents of a scene. In contrast, a node is a visual/graphical object of a scene graph. A node may include,
  - Geometrical (Graphical) objects (2D and 3D) such as – Circle, Rectangle, Polygon, etc.
  - UI Controls – Button, Checkbox, Choice Box, Text Area, etc.
  - Containers (Layout Panes) – Border Pane, Grid Pane, Flow Pane, etc.
  - Media elements – Audio, Video and Image Objects.
- The `Node` class of the package `javafx.scene` represents a node in JavaFX, this class is the super class of all the nodes.
  - Root Node – First Scene Graph is known as Root node. It's mandatory to pass root node to the scene graph.
  - Branch Node/Parent Node – The node with child nodes are known as branch/parent nodes. The abstract class named `Parent` of the package `javafx.scene` is the base class of all the parent nodes, and those parent nodes will be of the following types
    - Group – A group node is a collective node that contains a list of children nodes.
    - Region – It is the base class of all the JavaFX Node based UI Controls, such as Chart, Pane and Control.
    - WebView – This node manages the web engine and displays its contents.
  - Leaf Node – The node without child nodes is known as the leaf node. For example, Rectangle, Ellipse, Box, ImageView, MediaView are examples of leaf nodes.

Steps to create JavaFX application

1. Extend the Application class of `javafx.application` package in your class.
2. Override `start` method of Application class.
3. Prepare a scene graph with the required nodes.
4. Prepare a Scene with the required dimensions and add the scene graph (root node of the scene graph) to it.
5. Prepare a stage and add the scene to the stage and display the contents of the stage.

3. Prepare a Scene graph

- Since the root node is the first node, you need to create a root node and it can be chosen from the *Group, Region or WebView.*
  - Group

    A Group node is represented by the class named `Group` which belongs to the package `javafx.scene`, you can create a Group node by instantiating this class as shown below.

    ```
    Group root = new Group();
    Group root = new Group(NodeObject);
    ```

  - Region

    It is the Base class of all the JavaFX Node-based UI Controls, such as –

    - `Chart` – This class is the base class of all the charts and it belongs to the package `javafx.scene.chart` which embeds charts in application.
    - `Pane` – A Pane is the base class of all the layout panes such as AnchorPane, BorderPane, DialogPane, etc. This class belong to a package that is called as – `javafx.scene.layout` which inserts predefined layouts in your application.
    - `Control` – It is the base class of the User Interface controls such as Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTMLEditor, etc. This class belongs to the package javafx.scene.control.
  - WebView

    This node manages the web engine and displays its contents.

4. Preparing the Scene

- A JavaFX scene is represented by the `Scene` class of the package `javafx.scene`.

```
Scene scene = new Scene(root,width,heigth);
```

- While instantiating, it is mandatory to pass the root object to the constructor of the `Scene` class whereas width and height of the scene are optional parameters to the constructor.

5. Preparing the Stage

- Stage is the container of any JavaFX application and it provides a window for the application. It is represented by the `Stage` class of the package `javafx.stage`.
- An object of this class is passed as a parameter of the `start()` method of the Application class.
- Using this object, various operations on the stage can be performed like
  - *Set the title* for the stage using the method `setTitle()`.

    ```
    primaryStage.setTitle("Sample application");
    ```
  - *Attach the scene* object to the stage using the `setScene()` method.

    ```
    primaryStage.setScene(scene);
    ```
  - *Display the contents* of the scene using the `show()` method as shown below.

    ```
    primaryStage.show();
    ```

Basic Example of JavaFX

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MyFirstGUI extends Application{
    public void start(Stage primaryStage) throws Exception{
        Group root = new Group();
        Scene s = new Scene(root,600,400);
        primaryStage.setTitle("My First User Interface");
        s.setFill(Color.ORANGE);
        primaryStage.setScene(s);
        primaryStage.show();
    }

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

Create Scene Graph using Group, Region or WebView

Create Scene by adding Group (root) to it along with its width and height

Set the scene to the stage object (primaryStage) which is passed as an argument to start() using setScene() method

Lifecycle of JavaFX Application

- The JavaFX `Application` class has three life cycle methods.

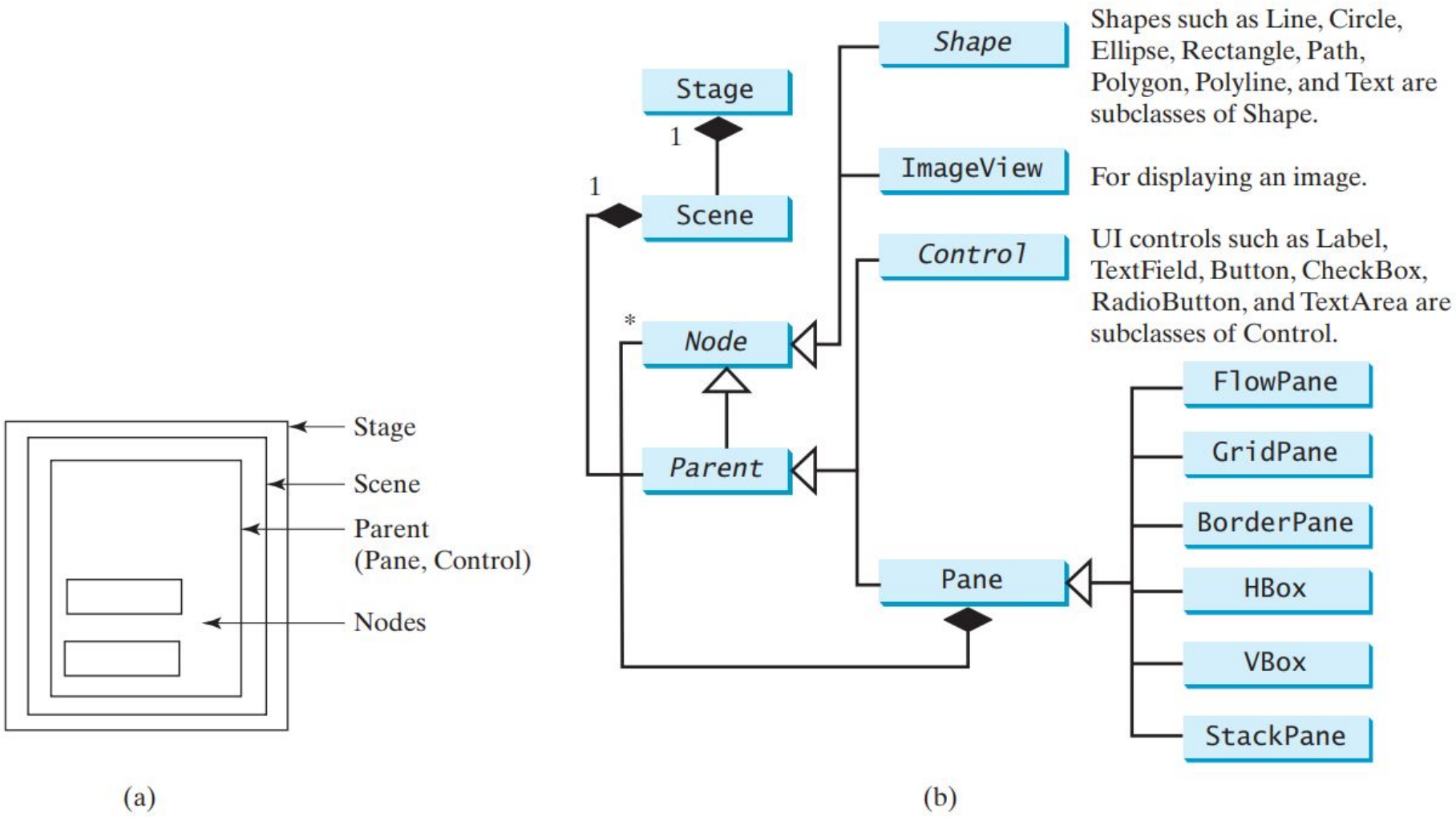
  `init()` - An empty method which can be overridden, stage or scene cannot be created in this method.

  `start()` - The entry point method where the JavaFX graphics code is to be written.

  `stop()` - An empty method which can be overridden, here the logic to stop the application is written.

- It also provides a static method named `launch()` to launch JavaFX application. This method is called from static content only mainly in main method.
- Whenever a JavaFX application is launched, the following actions will be carried out (in the same order).
  - An instance of the application class is created.
  - `init()` method is called.
  - `start()` method is called.
  - The launcher waits for the application to finish and calls the `stop()` method.

Panes, UI Control & Shapes

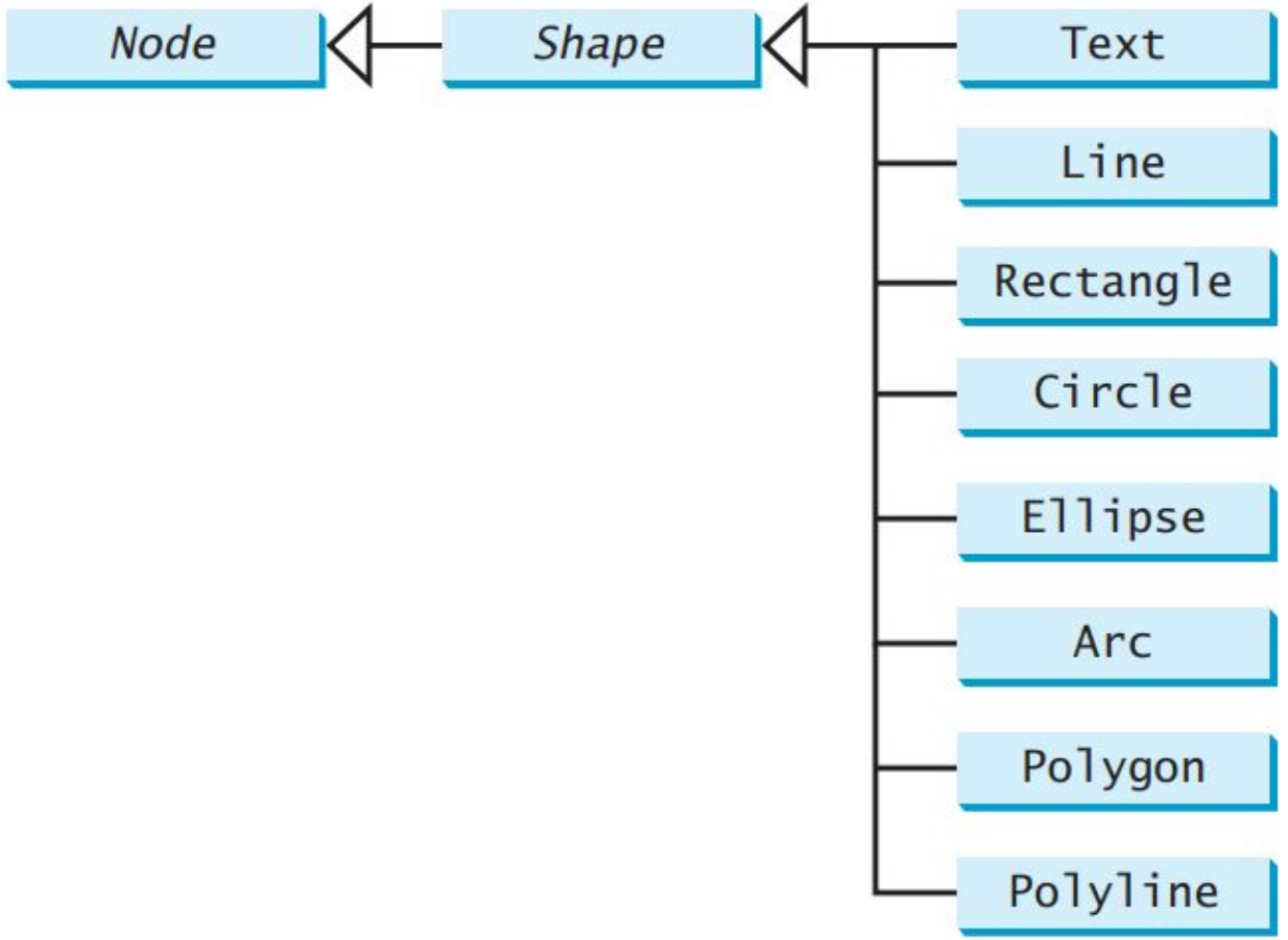


(a) Panes are used to hold nodes. (b) Nodes can be shapes, image views, UI controls, and panes.

2D Shape

- 2D shape is a geometrical figure that can be drawn on the XY plane like Line, Rectangle, Circle, etc.
- Using the JavaFX library, you can draw –
  - Predefined shapes – Line, Rectangle, Circle, Ellipse, Polygon, Polyline, Cubic Curve, Quad Curve, Arc.
  - 2D shape by parsing SVG path.
- Each of the above mentioned 2D shape is represented by a class which belongs to the package `javafx.scene.shape`. The class named Shape is the base class of all the 2-Dimensional shapes in JavaFX.
- The Shape class is the abstract base class that defines the common properties for all shapes.
- Among them are the fill, stroke, and strokeWidth properties.
- The fill property specifies a color that fills the interior of a shape. The stroke property specifies a color that is used to draw the outline of a shape. The strokeWidth property specifies the width of the outline of a shape.

Shapes Class



A shape is a node. The **Shape** class is the root of all shape classes.

Classes for Java FX Shape (javafx.scene.shape)

Shape	Description
Line	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, **javafx.scene.shape.Line** class needs to be instantiated in order to create lines.
Rectangle	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, **javafx.scene.shape.Rectangle** class needs to be instantiated in order to create Rectangles.
Ellipse	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX. **javafx.scene.shape.Ellipse** class needs to be instantiated in order to create Ellipse.
Arc	Arc can be defined as the part of the circumference of the circle of ellipse. In JavaFX, **javafx.scene.shape.Arc** class needs to be instantiated in order to create Arcs.
Circle	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating **javafx.scene.shape.Circle** class.
Polygon	Polygon is a geometrical figure that can be created by joining the multiple Co-planner line segments. In JavaFX, **javafx.scene.shape**. Pollygon class needs to be instantiated in order to create polygon.
Cubic Curve	A Cubic curve is a curve of degree 3 in the XY plane. In Javafx, **javafx.scene.shape.CubicCurve** class needs to be instantiated in order to create Cubic Curves.
Quad Curve	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, **javafx.scene.shape.QuadCurve** class needs to be instantiated in order to create QuadCurve.

Properties for Java FX Shape (`javafx.scene.shape`)

Property	Description	Setter Methods
fill	Used to fill the shape with a defined paint. This is a object <paint> type property.	setFill(Paint)
smooth	This is a boolean type property. If true is passes then the edges of the shape will become smooth.	setSmooth(boolean)
strokeDashOffset	It defines the distances in the coordinate system which shows the shapes in the dashing patterns. This is a double type property.	setStrokeDashOffset(Double)
strokeLineCap	It represents the style of the line end cap. It is a strokeLineCap type property.	setStrokeLineCap(StrokeLineCap)
strokeLineJoin	It represents the style of the joint of the two paths.	setStrokeLineJoin(StrokeLineJoin)
strokeMiterLimit	It applies the limitation on the distance between the inside and outside points of a joint. It is a double type property.	setStrokeMiterLimit(Double)
stroke	It is a colour type property which represents the colour of the boundary line of the shape.	setStroke(paint)
strokeType	It represents the type of the stroke (where the boundary line will be imposed to the shape) whether inside, outside or centred.	setStrokeType(StrokeType)
strokeWidth	It represents the width of the stroke.	setStrokeWidth(Double)

Shape - Text

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.text.Text

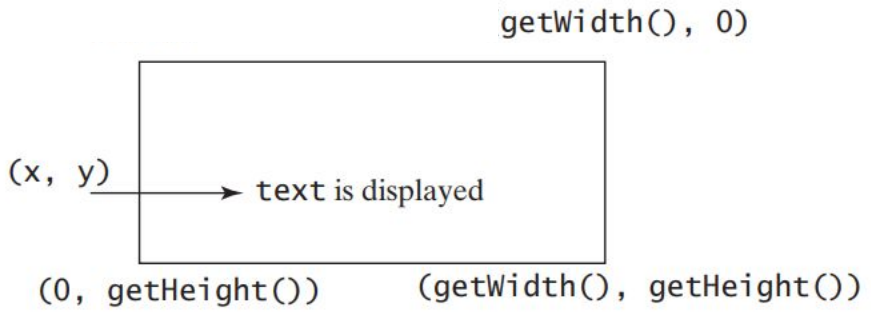
```
-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty<Font>
```

```
+Text()
+Text(text: String)
+Text(x: double, y: double,
   text: String)
```

Defines the text to be displayed.
Defines the x-coordinate of text (default 0).
Defines the y-coordinate of text (default 0).
Defines if each line has an underline below it (default `false`).
Defines if each line has a line through it (default `false`).
Defines the font for the text.

Creates an empty Text.
Creates a Text with the specified text.
Creates a Text with the specified x-, y-coordinates and text.

`Text` defines a node for displaying a text.



(a) Text(x, y, text)

(b) *Three* Text *objects are displayed*

Example : Text

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.geometry.Insets;
6  import javafx.stage.Stage;
7  import javafx.scene.text.Text;
8  import javafx.scene.text.Font;
9  import javafx.scene.text.FontWeight;
10 import javafx.scene.text.FontPosture;
11
12 public class ShowText extends Application {
13   @Override // Override the start method in the Application class
14   public void start(Stage primaryStage) {
15     // Create a pane to hold the texts
16     Pane pane = new Pane();
17     pane.setPadding(new Insets(5, 5, 5, 5));
18     Text text1 = new Text(20, 20, "Programming is fun");
19     text1.setFont(Font.font("Courier", FontWeight.BOLD,
20       FontPosture.ITALIC, 15));
21     pane.getChildren().add(text1);
22
23     Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
24     pane.getChildren().add(text2);
25
26     Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
27     text3.setFill(Color.RED);
28     text3.setUnderline(true);
29     text3.setStrikethrough(true);
30     pane.getChildren().add(text3);
31
32     // Create a scene and place it in the stage
33     Scene scene = new Scene(pane);
34     primaryStage.setTitle("ShowText"); // Set the stage title
35     primaryStage.setScene(scene); // Place the scene in the stage
36     primaryStage.show(); // Display the stage
37   }
38 }
```

create a pane

create a text
set text font

add text to pane

create a two-line text
add text to pane

create a text
set text color
set underline
set strike line
add text to pane

- The **Text** class defines a node that **displays a string at a starting point (x, y)**.
- A Text object is **usually placed in a pane**. The pane's upper-left corner point is **(0, 0)** and the bottom-right point is **(pane.getWidth(), pane.getHeight())**.
- A string may be displayed in multiple lines separated by **\n**.
- The program creates a **Text** (line 18), sets its **font (line 19)**, and places it to the pane **(line 21)**.
- The program creates another Text with multiple lines **(line 23)** and places it to the pane **(line 24)**.
- The program creates the third Text **(line 26)**, sets its color **(line 27)**,
- sets an underline and a strike through line **(lines 28–29)**, and places it to the pane **(line 30)**.

Shape - Line

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Line

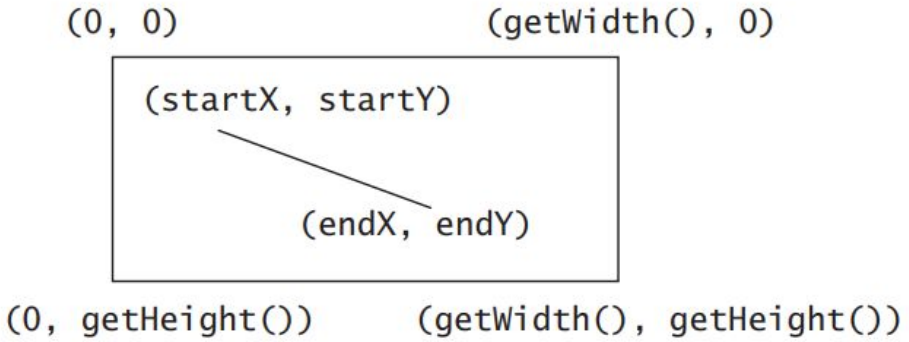
```
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
```

```
+Line()
+Line(startX: double, startY:
   double, endX: double, endY:
   double)
```

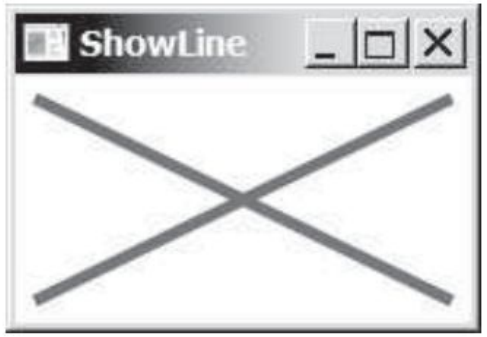
The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.

Creates an empty `Line`.
Creates a `Line` with the specified starting and ending points.

The `Line` class defines a line.



(a) `Line(startX, startY, endX, endY)`



(b) Two lines are displayed across the pane.

Example : Line

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.stage.Stage;
6  import javafx.scene.shape.Line;
7
8  public class ShowLine extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a scene and place it in the stage
12     Scene scene = new Scene(new LinePane(), 200, 200);          create a pane in scene
13     primaryStage.setTitle("ShowLine"); // Set the stage title
14     primaryStage.setScene(scene); // Place the scene in the stage
15     primaryStage.show(); // Display the stage
16   }
17 }
18
19 class LinePane extends Pane {                                  define a custom pane
20   public LinePane() {
21     Line line1 = new Line(10, 10, 10, 10);                      create a line
22     line1.endXProperty().bind(widthProperty().subtract(10));
23     line1.endYProperty().bind(heightProperty().subtract(10));
24     line1.setStrokeWidth(5);                                    set stroke width
25     line1.setStroke(Color.GREEN);                               set stroke
26     getChildren().add(line1);                                   add line to pane
27
28     Line line2 = new Line(10, 10, 10, 10);                      create a line
29     line2.startXProperty().bind(widthProperty().subtract(10));
30     line2.endYProperty().bind(heightProperty().subtract(10));
31     line2.setStrokeWidth(5);
32     line2.setStroke(Color.GREEN);
33     getChildren().add(line2);                                   add line to pane
34   }
35 }
```

- A line connects two points with four parameters **startX**, **startY**, **endX**, and **endY**.
- The **Line** class defines a line.
- The program defines a **custom pane class** named **LinePane (line 19)**.
- The custom pane class creates two lines and binds the starting and ending points of the line with the width and height of the pane (lines 22–23, 29–30) so that the two points of the lines are changed as the pane is resized.

Shape - Rectangle

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Rectangle

```
-x: DoubleProperty
-y:DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty

-arcHeight: DoubleProperty
```

```
+Rectangle()
+Rectanlge(x: double, y:
   double, width: double,
   height: double)
```

The x-coordinate of the upper-left corner of the rectangle (default 0).

The y-coordinate of the upper-left corner of the rectangle (default 0).

The width of the rectangle (default: 0).

The height of the rectangle (default: 0).

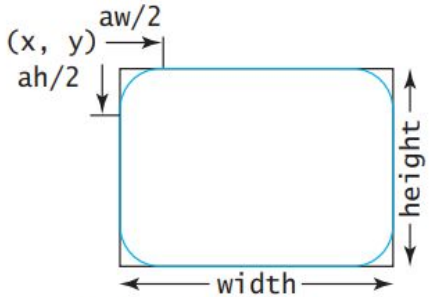
The `arcWidth` of the rectangle (default: 0). `arcWidth` is the horizontal diameter of the arcs at the corner (see Figure 14.31a).

The `arcHeight` of the rectangle (default: 0). `arcHeight` is the vertical diameter of the arcs at the corner (see Figure 14.31a).

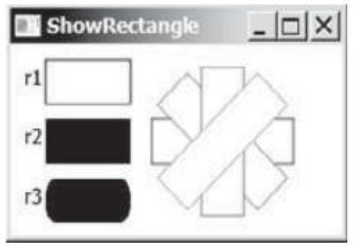
Creates an empty `Rectangle`.

Creates a `Rectangle` with the specified upper-left corner point, width, and height.

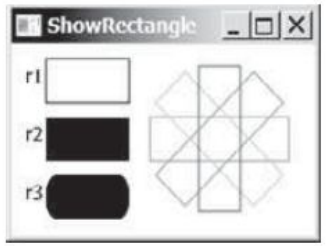
The `Rectangle` class defines a rectangle.



(a) `Rectangle(x, y, w, h)`



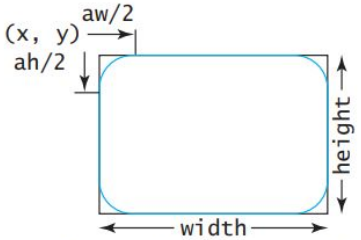
(b) Multiple rectangles are displayed



(c) Transparent rectangles are displayed

Example : Rectangle

```
9  public class ShowRectangle extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     // Create a pane
13     Pane pane = new Pane();                                  create a pane
14
15     // Create rectangles and add to pane
16     Rectangle r1 = new Rectangle(25, 10, 60, 30);            create a rectangle r1
17     r1.setStroke(Color.BLACK);                               set r1's properties
18     r1.setFill(Color.WHITE);
19     pane.getChildren().add(new Text(10, 27, "r1"));
20     pane.getChildren().add(r1);                              add r1 to pane
21
22     Rectangle r2 = new Rectangle(25, 50, 60, 30);            create rectangle r2
23     pane.getChildren().add(new Text(10, 67, "r2"));
24     pane.getChildren().add(r2);                              add r2 to pane
25
26     Rectangle r3 = new Rectangle(25, 90, 60, 30);            create rectangle r3
27     r3.setArcWidth(15);                                      set r3's arc width
28     r3.setArcHeight(25);                                     set r3's arc height
29     pane.getChildren().add(new Text(10, 107, "r3"));
30     pane.getChildren().add(r3);
31
32     for (int i = 0; i < 4; i++) {
33       Rectangle r = new Rectangle(100, 50, 100, 30);         create a rectangle
34       r.setRotate(i * 360 / 8);                              rotate a rectangle
35       r.setStroke(Color.color(Math.random(), Math.random(),
36           Math.random()));
37       r.setFill(Color.WHITE);
38       pane.getChildren().add(r);                             add rectangle to pane
39     }
40
41     // Create a scene and place it in the stage
42     Scene scene = new Scene(pane, 250, 150);
43     primaryStage.setTitle("ShowRectangle"); // Set the stage title
44     primaryStage.setScene(scene); // Place the scene in the stage
45     primaryStage.show(); // Display the stage
46   }
47 }
```



- A **rectangle** is defined by the parameters **x**, **y**, **width**, **height**, **arcWidth**, and **arcHeight**.
- The rectangle's **upper-left corner point is at (x, y)** and parameter **aw (arcWidth)** is the **horizontal diameter of the arcs at the corner**, and **ah (arcHeight)** is the **vertical diameter of the arcs at the corner**
- By **default**, the **fill color** is **black**. So a rectangle is filled with black color.
- The **stroke color** is **white by default**. **Line 17** sets stroke color of rectangle **r1** to black. The program creates rectangle **r3 (line 26)** and sets its arc width and arc height (lines 27–28). So r3 is displayed as a rounded rectangle.
- The program **repeatedly creates** a rectangle (line 33), **rotates it (line 34)**, sets a **random stroke color (lines 35–36)**, its fill color to white (line 37), and adds the rectangle to the pane (line 38).

Shape – Circle , Ellipse

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Circle

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radius: DoubleProperty
+Circle()
+Circle(x: double, y: double)
+Circle(x: double, y: double,
   radius: double)
```

The x-coordinate of the center of the circle (default 0).
The y-coordinate of the center of the circle (default 0).
The radius of the circle (default: 0).

Creates an empty `Circle`.
Creates a `Circle` with the specified center.
Creates a `Circle` with the specified center and radius.

The `Circle` class defines circles.

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

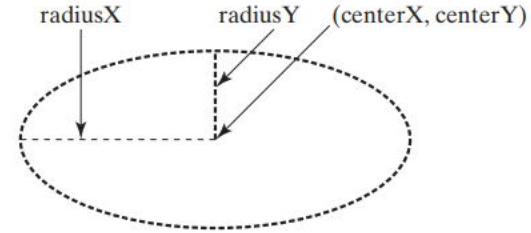
javafx.scene.shape.Ellipse

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
+Ellipse()
+Ellipse(x: double, y: double)
+Ellipse(x: double, y: double,
   radiusX: double, radiusY:
   double)
```

The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).

Creates an empty `Ellipse`.
Creates an `Ellipse` with the specified center.
Creates an `Ellipse` with the specified center and radiuses.

The `Ellipse` class defines ellipses.

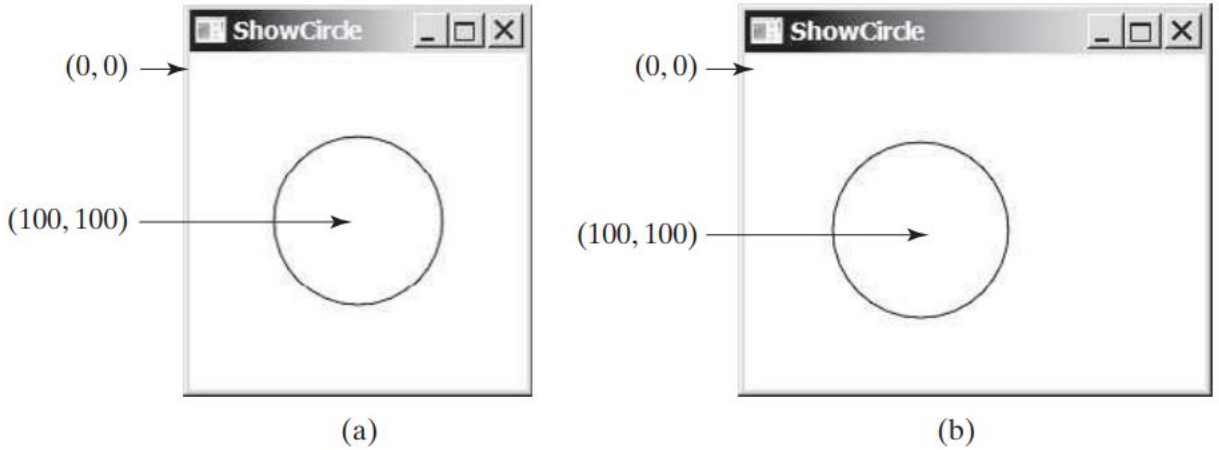


(a) `Ellipse(centerX, centerY, radiusX, radiusY)`

(b) Multiple ellipses are displayed.

Example : Circle

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.stage.Stage;
7
8  public class ShowCircle extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a circle and set its properties
12     Circle circle = new Circle();
13     circle.setCenterX(100);
14     circle.setCenterY(100);
15     circle.setRadius(50);
16     circle.setStroke(Color.BLACK);
17     circle.setFill(Color.WHITE);
18
19     // Create a pane to hold the circle
20     Pane pane = new Pane();
21     pane.getChildren().add(circle);
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 200, 200);
25     primaryStage.setTitle("ShowCircle"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 }
```



Example : Ellipse

```
10    public void start(Stage primaryStage) {
11      // Create a pane
12      Pane pane = new Pane();                                  create a pane
13
14      for (int i = 0; i < 16; i++) {
15        // Create an ellipse and add it to pane
16        Ellipse e1 = new Ellipse(150, 100, 100, 50);           create an ellipse
17        e1.setStroke(Color.color(Math.random(), Math.random(), set random color for stroke
18          Math.random()));
19        e1.setFill(Color.WHITE);                               set fill color
20        e1.setRotate(i * 180 / 16);                            rotate ellipse
21        pane.getChildren().add(e1);                            add ellipse to pane
22      }
23
24      // Create a scene and place it in the stage
25      Scene scene = new Scene(pane, 300, 200);
26      primaryStage.setTitle("ShowEllipse"); // Set the stage title
27      primaryStage.setScene(scene); // Place the scene in the stage
28      primaryStage.show(); // Display the stage
29    }
30  }
```

- The program repeatedly creates an ellipse **(line 16)**
- Sets a random stroke color **(lines 17–18)**
- Sets its fill color to white **(line 19)**
- Rotates it **(line 20)**
- And adds the rectangle to the pane **(line 21)**

Shape - Arc

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Arc

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
-startAngle: DoubleProperty
-length: DoubleProperty
-type: ObjectProperty<ArcType>
```

```
+Arc()
+Arc(x: double, y: double,
   radiusX: double, radiusY:
   double, startAngle: double,
   length: double)
```

The x-coordinate of the center of the ellipse (default 0).

The y-coordinate of the center of the ellipse (default 0).

The horizontal radius of the ellipse (default: 0).

The vertical radius of the ellipse (default: 0).

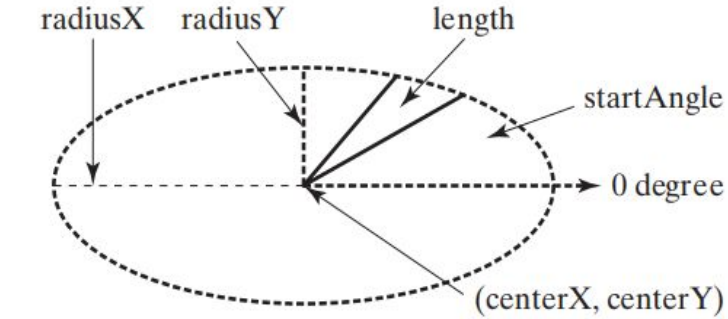
The start angle of the arc in degrees.

The angular extent of the arc in degrees.

The closure type of the arc (`ArcType.OPEN`, `ArcType.CHORD`, `ArcType.ROUND`).

Creates an empty `Arc`.

Creates an `Arc` with the specified arguments.



(a) `Arc(centerX, centerY, radiusX, radiusY, startAngle, length)`

Example : Polygon, Polyline

```
10 public class ShowArc extends Application {
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     // Create a pane
14     Pane pane = new Pane();
15
16     Arc arc1 = new Arc(150, 100, 80, 80, 30, 35); // Create an arc
17     arc1.setFill(Color.RED); // Set fill color
18     arc1.setType(ArcType.ROUND); // Set arc type
19     pane.getChildren().add(new Text(210, 40, "arc1: round"));
20     pane.getChildren().add(arc1); // Add arc to pane
21
22     Arc arc2 = new Arc(150, 100, 80, 80, 30 + 90, 35);
23     arc2.setFill(Color.WHITE);
24     arc2.setType(ArcType.OPEN);
25     arc2.setStroke(Color.BLACK);
26     pane.getChildren().add(new Text(20, 40, "arc2: open"));
27     pane.getChildren().add(arc2);
28
29     Arc arc3 = new Arc(150, 100, 80, 80, 30 + 180, 35);
30     arc3.setFill(Color.WHITE);
31     arc3.setType(ArcType.CHORD);
32     arc3.setStroke(Color.BLACK);
33     pane.getChildren().add(new Text(20, 170, "arc3: chord"));
34     pane.getChildren().add(arc3);
35
36     Arc arc4 = new Arc(150, 100, 80, 80, 30 + 270, 35);
37     arc4.setFill(Color.GREEN);
38     arc4.setType(ArcType.CHORD);
39     arc4.setStroke(Color.BLACK);
40     pane.getChildren().add(new Text(210, 170, "arc4: chord"));
41     pane.getChildren().add(arc4);
42
43     // Create a scene and place it in the stage
44     Scene scene = new Scene(pane, 300, 200);
45     primaryStage.setTitle("ShowArc"); // Set the stage title
46     primaryStage.setScene(scene); // Place the scene in the stage
47     primaryStage.show(); // Display the stage
48   }
49 }
```

create a pane

create arc1
set fill color for arc1
set arc1 as round arc

add arc1 to pane

create arc2
set fill color for arc2
set arc2 as round arc

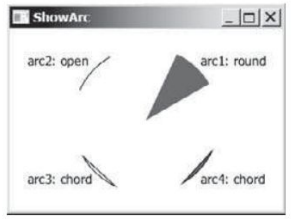
add arc2 to pane

create arc3
set fill color for arc3
set arc3 as chord arc

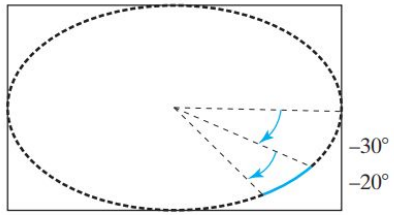
add arc3 to pane

create arc4

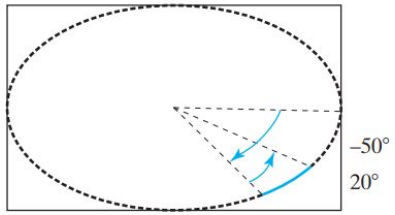
add arc4 to pane



- **Angles may be negative**. A negative starting angle sweeps clockwise from the easterly direction, as shown in Figure.
- A negative spanning angle sweeps clockwise from the starting angle.
- The following two statements define the same arc:
  - new Arc(x, y, radiusX, radiusY, -30, -20);
  - new Arc(x, y, radiusX, radiusY, -50, 20);
- The first statement uses negative starting angle -30 and negative spanning angle -20.
- The second statement uses negative starting angle -50 and positive spanning angle 20.

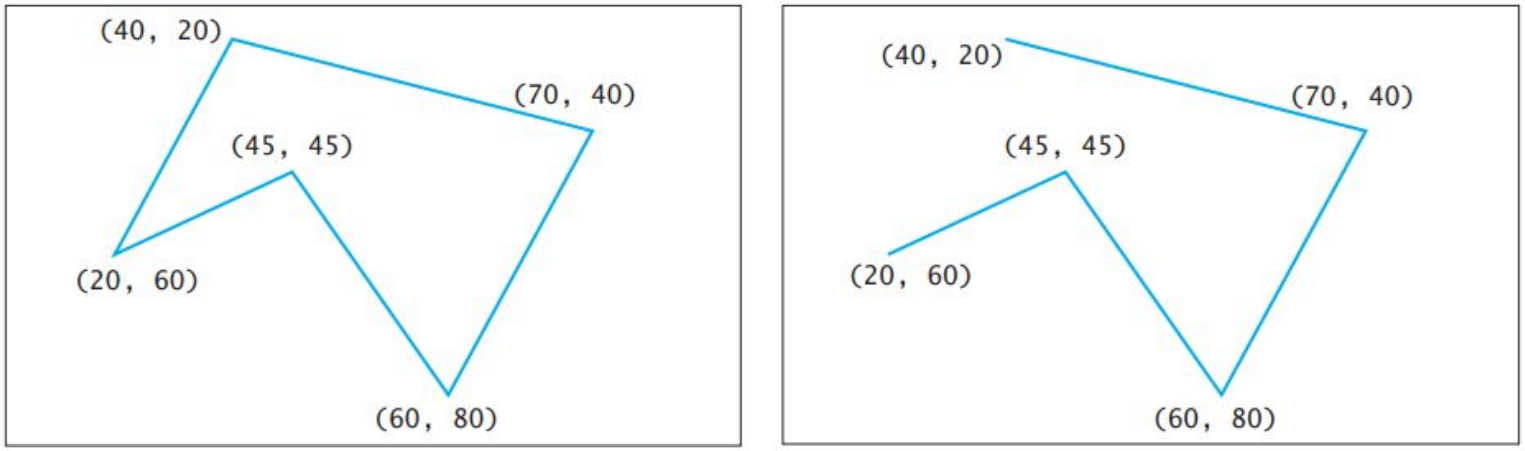


(a) Negative starting angle –30° and negative spanning angle –20°



(b) Negative starting angle –50° and positive spanning angle 20°

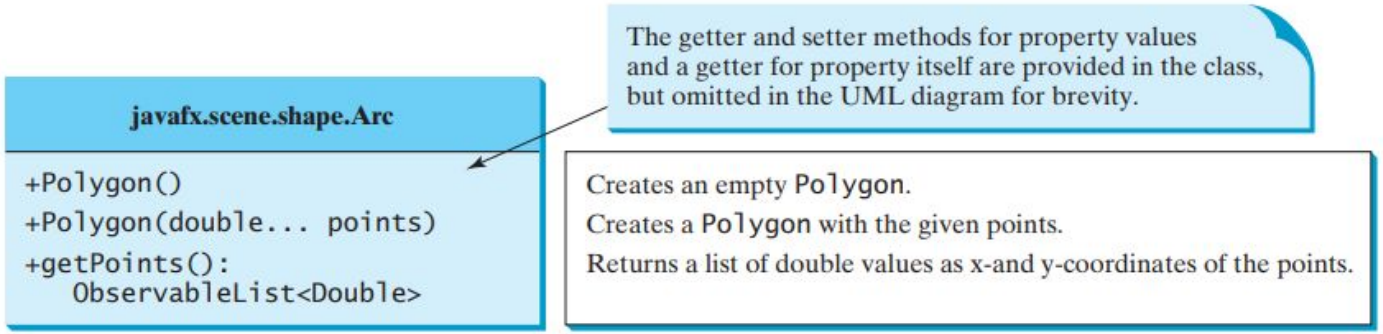
Shape – Polygon , Polyline



(a) Polygon (b) Polyline

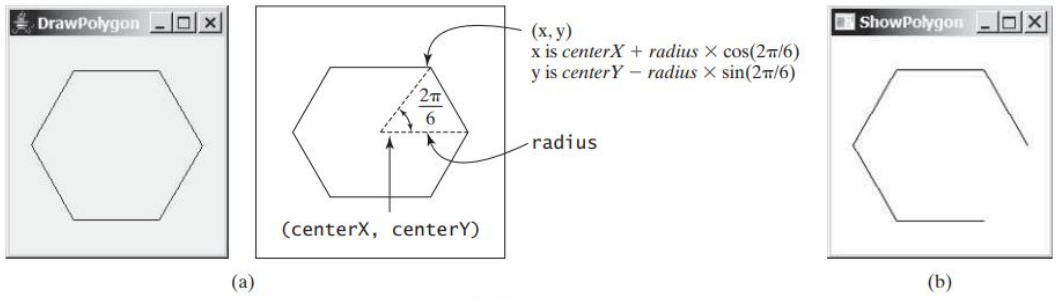
`Polygon` is closed and `Polyline` is not closed.

The UML diagram for the `Polygon` class is shown in Figure . Listing 14.19 gives an example that creates a hexagon, as shown in Figure



Example : Polygon , Polyline

```
 9  public class ShowPolygon extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) {
12      // Create a pane, a polygon, and place polygon to pane
13      Pane pane = new Pane();                                    create a pane
14      Polygon polygon = new Polygon();                           create a polygon
15      pane.getChildren().add(polygon);                           add polygon to pane
16      polygon.setFill(Color.WHITE);
17      polygon.setStroke(Color.BLACK);
18      ObservableList<Double> list = polygon.getPoints();         get a list of points
19
20      final double WIDTH = 200, HEIGHT = 200;
21      double centerX = WIDTH / 2, centerY = HEIGHT / 2;
22      double radius = Math.min(WIDTH, HEIGHT) * 0.4;
23
24      // Add points to the polygon list
25      for (int i = 0; i < 6; i++) {
26        list.add(centerX + radius * Math.cos(2 * i * Math.PI / 6));   add x-coordinate of a point
27        list.add(centerY - radius * Math.sin(2 * i * Math.PI / 6));   add y-coordinate of a point
28      }
29
30      // Create a scene and place it in the stage
31      Scene scene = new Scene(pane, WIDTH, HEIGHT);              add pane to scene
32      primaryStage.setTitle("ShowPolygon"); // Set the stage title
33      primaryStage.setScene(scene); // Place the scene in the stage
34      primaryStage.show(); // Display the stage
35    }
36  }
```



- If you **replace Polygon** by **Polyline**, the program displays a polyline as shown in Figure b.
- The Polyline class is used in the same way as Polygon except that the **starting and ending point are not connected in Polyline**.
- The program **creates a polygon** (line 14) and **adds it to a pane** (line 15).
- The polygon **.getPoints()** method **returns an ObservableList (line 18)**, which **contains** the **add()** method for **adding an element to the list (lines 26–27)**.
- **Note** that the value passed to **add(value) must be a double value**. **If an int value is passed, the int value would be automatically boxed into an Integer**.
- This would cause an error because the ObservableList consists of Double elements.

Classes for Shape (`javafx.scene.shape`)

Shape	Class	Example

Line	`Line`	`Line line = new Line();`
		`line.setStartX(100.0);`
		`line.setStartY(150.0);`
		`line.setEndX(500.0);`
		`line.setEndY(150.0);`

Rectangle & Rounded Rectangle	`Rectangle`	`Rectangle rectangle = new Rectangle();`
		`rectangle.setX(150.0f);`
		`rectangle.setY(75.0f);`
		`rectangle.setWidth(300.0f);`
		`rectangle.setHeight(150.0f);`
		`rectangle.setArcWidth(30.0);`
		`rectangle.setArcHeight(20.0);`

Circle	`Circle`	`Circle circle = new Circle();`
		`circle.setCenterX(300.0f);`
		`circle.setCenterY(135.0f);`
		`circle.setRadius(100.0f);`

Classes for Shape (javafx.scene.shape) (Cont.)

Shape	Class	Example

Ellipse	`Ellipse`	`Ellipse ellipse = new Ellipse();`
		`ellipse.setCenterX(300.0f);`
		`ellipse.setCenterY(150.0f);`
		`ellipse.setRadiusX(150.0f);`
		`ellipse.setRadiusY(75.0f);`

Polygon	`Polygon`	`Polygon polygon = new Polygon();`
		`polygon.getPoints().addAll(new Double[]{`
		`300.0, 50.0,`
		`450.0, 150.0,`
		`300.0, 250.0,`
		`150.0, 150.0,`
		`});`

Classes for Shape (javafx.scene.shape) (Cont.)

Shape	Class	Example

Polyline	Polyline	`Polyline polyline = new Polyline();`
		`polyline.getPoints().addAll(new Double[]{`
		`200.0, 50.0,`
		`400.0, 50.0,`
		`450.0, 150.0,`
		`400.0, 250.0,`
		`200.0, 250.0,`
		`150.0, 150.0,`
		`});`

Cubic Curve	CubicCurve	`CubicCurve cubicCurve = new CubicCurve();`
		`cubicCurve.setStartX(100.0f);`
		`cubicCurve.setStartY(150.0f);`
		`cubicCurve.setControlX1(400.0f);`
		`cubicCurve.setControlY1(40.0f);`
		`cubicCurve.setControlX2(175.0f);`
		`cubicCurve.setControlY2(250.0f);`
		`cubicCurve.setEndX(500.0f);`
		`cubicCurve.setEndY(150.0f);`

Classes for Shape (`javafx.scene.shape`) (Cont.)

Shape	Class	Description

Quad Curve	QuadCurve	`QuadCurve quadCurve = new QuadCurve();`
		`quadCurve.setStartX(100.0);`
		`quadCurve.setStartY(220.0f);`
		`quadCurve.setEndX(500.0f);`
		`quadCurve.setEndY(220.0f);`
		`quadCurve.setControlX(250.0f);`
		`quadCurve.setControlY(0.0f);`

Arc	Arc	`Arc arc = new Arc();`
		`arc.setCenterX(100.0);`
		`arc.setCenterY(100.0);`
		`arc.setRadiusX(100.0);`
		`arc.setRadiusY(100.0);`
		`arc.setStartAngle(0.0);`
		`arc.setLength(100.0);`

Example (Adding 2-D Shapes)

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

public class MyFirstGUI extends Application {
    public void start(Stage primaryState) throws Exception {
        Ellipse ellipse = new Ellipse();
        ellipse.setCenterX(300.0f); ellipse.setCenterY(150.0f);
        ellipse.setRadiusX(150.0f); ellipse.setRadiusY(75.0f);

        Group root = new Group(ellipse);
        Scene s = new Scene(root, 600, 400);
        primaryState.setScene(s);
        primaryState.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Create object of Ellipse class and set its properties.

Pass object of ellipse as argument to Group contructor

JavaFX – Property Binding

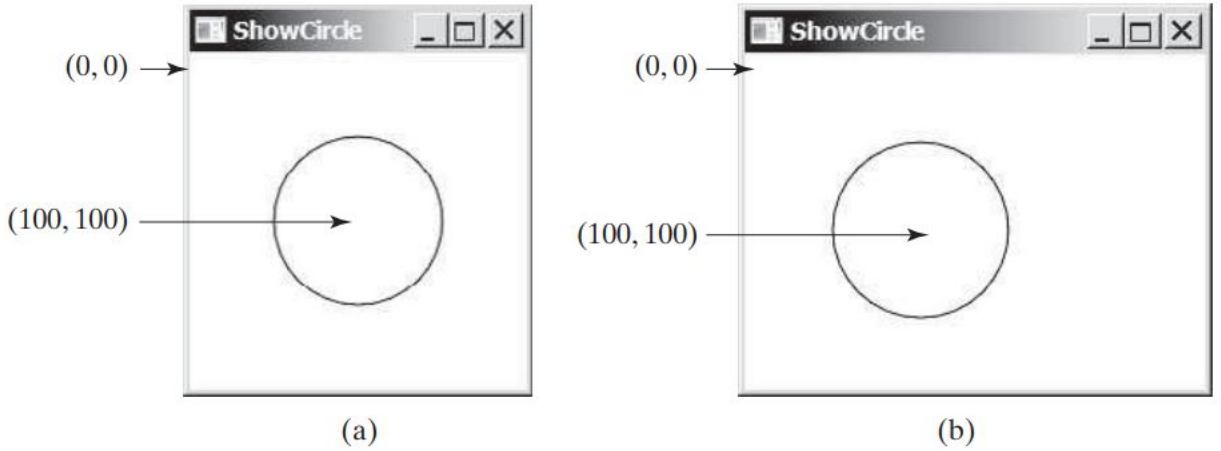
- You can bind a target object to a source object. A **change in the source object will be automatically reflected in the target object**.
- JavaFX introduces a new concept called **property binding** that **enables a target object to be bound to a source object**.
- If the value in the source object changes, the target object is also changed automatically.
- The **target object** is called a **binding object** or a **binding property** and the **source object** is called a **bindable object** or **observable object**
- In Circle Example program the circle is not centered after the window is resized.
- In order to display the circle centered as the window resizes, the x- and y-coordinates of the circle center need to be reset to the center of the pane.

JavaFX – Property Binding

- A target binds with a source using the bind method as follows:
  - **target.bind(source);**
- The bind method is defined in the javafx.beans.property.Property interface.
- A binding property is an instance of javafx.beans.property.Property.
- A source object is an instance of the javafx.beans.value.ObservableValue interface.
- An ObservableValue is an entity that wraps a value and allows to observe the value for changes.
- JavaFX defines binding properties for primitive types and strings. For a double/float/long/int/boolean value, its binding property type is **DoubleProperty/FloatProperty/LongProperty/IntegerProperty/BooleanProperty**.
- For a string, its binding property type is StringProperty. These properties are also subtypes of ObservableValue. So they can also be used as source objects for binding properties.

Example : Circle Class without binding property

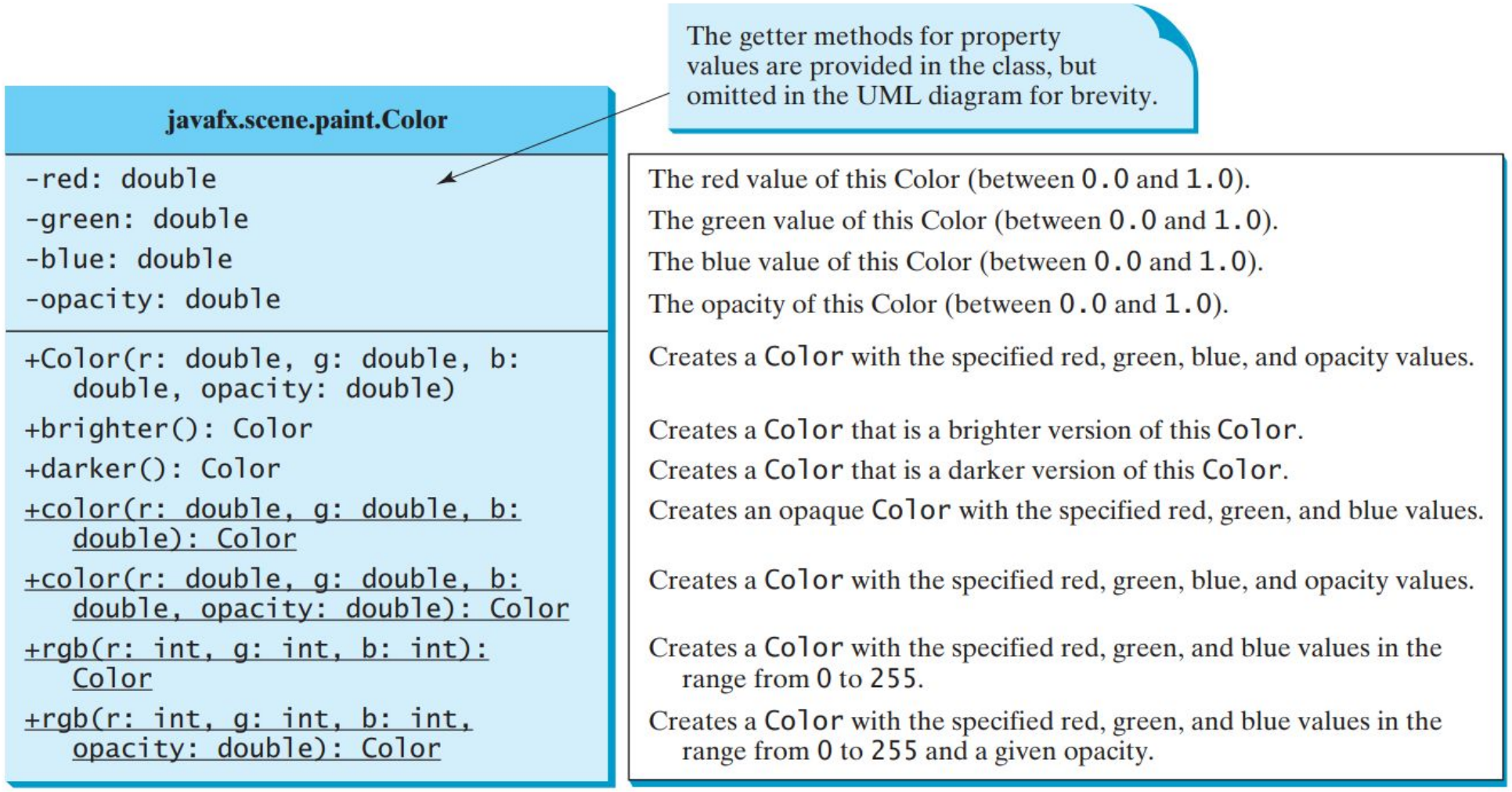
```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.stage.Stage;
7
8  public class ShowCircle extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a circle and set its properties
12     Circle circle = new Circle();
13     circle.setCenterX(100);
14     circle.setCenterY(100);
15     circle.setRadius(50);
16     circle.setStroke(Color.BLACK);
17     circle.setFill(Color.WHITE);
18
19     // Create a pane to hold the circle
20     Pane pane = new Pane();
21     pane.getChildren().add(circle);
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 200, 200);
25     primaryStage.setTitle("ShowCircle"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 }
```



Example : Circle with binding property

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.stage.Stage;
7
8  public class ShowCircleCentered extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a pane to hold the circle
12     Pane pane = new Pane();
13
14     // Create a circle and set its properties
15     Circle circle = new Circle();
16     circle.centerXProperty().bind(pane.widthProperty().divide(2));
17     circle.centerYProperty().bind(pane.heightProperty().divide(2));
18     circle.setRadius(50);
19     circle.setStroke(Color.BLACK);
20     circle.setFill(Color.WHITE);
21     pane.getChildren().add(circle); // Add circle to the pane
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 200, 200);
25     primaryStage.setTitle("ShowCircleCentered"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 }
```

javaFX : Color Class



`Color` encapsulates information about colors.

JavaFX - Colors

- `javafx.scene.paint` package provides various classes to apply colors to an application. This package contains an abstract class named `Paint` and it is the base class of all the classes that are used to apply colors.
- Using these classes, you can apply colors in the following patterns
  - Uniform – color is applied uniformly throughout node.
  - Image Pattern – fills the region of the node with an image pattern.
  - Gradient – the color applied to the node varies from one point to the other. It has two kinds of gradients namely Linear Gradient and Radial Gradient.
- Instance of `Color` class can be created by providing Red, Green, Blue and Opacity value ranging from 0 to 1 in double.

```
Color color = new Color(double red, double green, double blue, double opacity);
Color color = new Color(0.0,0.3,0.2,1.0);
```

- Instance of `Color` class can be created using following methods also

```
Color c = Color.rgb(0,0,255);      //passing RGB values
Color c = Color.hsb(270,1.0,1.0);    //passing HSB values
Color c = Color.web("0x0000FF",1.0); //passing hex code
```

Applying Color to the Nodes

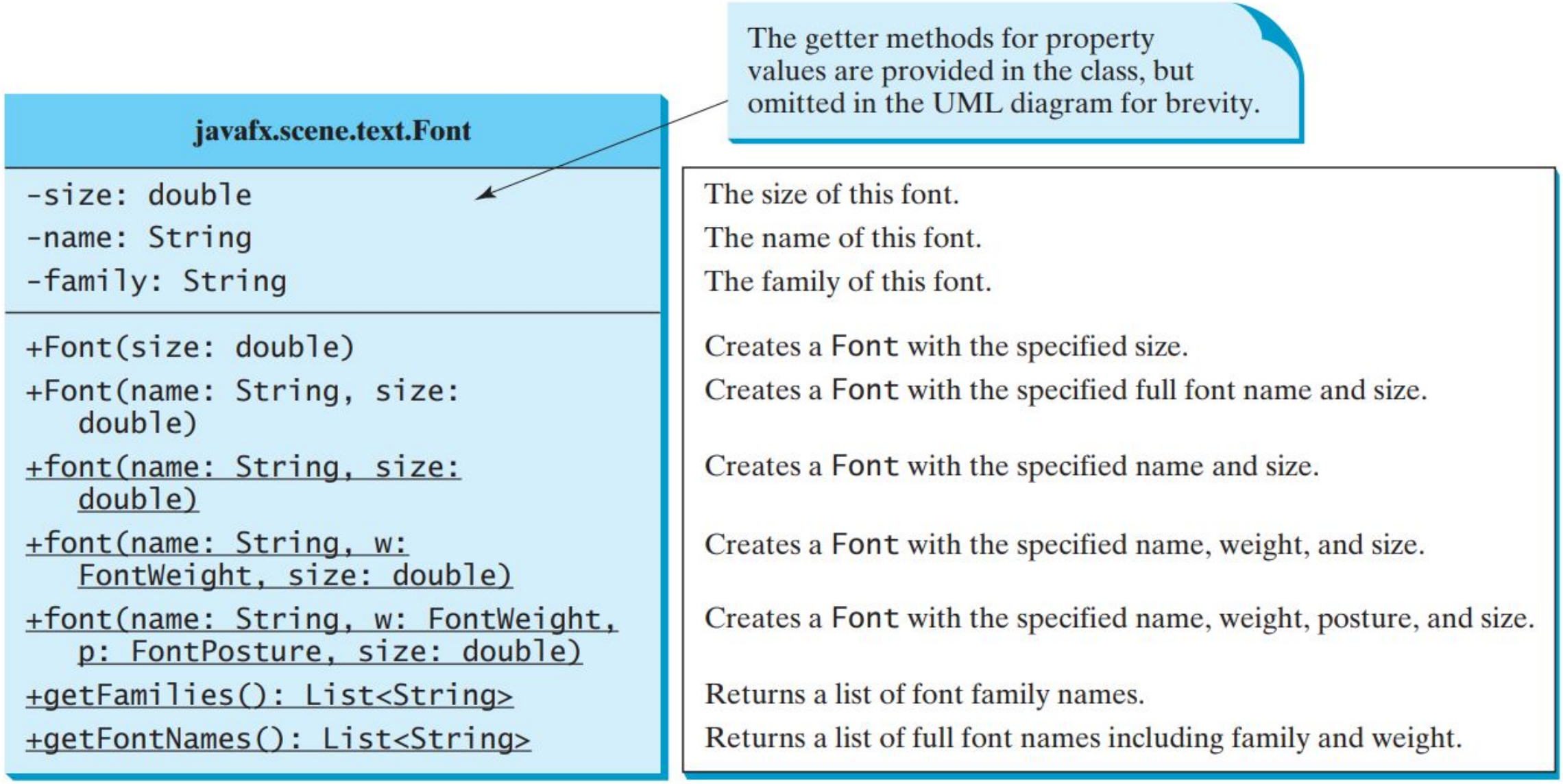
- `setFill(Color)` method is used to apply color to nodes such as Shape, Text, etc.
- `setStroke(Color)` method is used to apply strokes to the nodes.

```
//Setting color to the text
Color color = new Color.BEIGE
text.setFill(color);

//Setting color to the stroke
Color color = new Color.DARKSLATEBLUE
circle.setStroke(color);
```

Color Example

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;

public class MyFirstGUI extends Application {
    public void start(Stage stage) {
        Circle circle = new Circle(); circle.setCenterX(300.0f); circle.setCenterY(180.0f);
        circle.setRadius(90.0f);
        circle.setFill(Color.RED); circle.setStrokeWidth(3); circle.setStroke(Color.GREEN);
        Group root = new Group(circle);
        Scene scene = new Scene(root, 600, 300);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String args[]) {
        launch(args);
    }
}
```

javaFX : Font Class



JavaFX - Font

- `javafx.scene.text` package provides various classes to apply text fonts to an application. This package contains a class named `Font`.
- A Font instance can be constructed using its constructors or using its static methods.
- A Font is defined by its name, weight, posture, and size.
- Times, Courier, and Arial are the examples of the font names.
- You can obtain a list of available font family names by invoking the static getFamilies() method.
- List is an interface that defines common methods for a list. ArrayList is a concrete implmentation of List.
- The font postures are two constants: FontPosture.ITALIC and FontPosture.REGULAR.
- For example, the following statements create two fonts.
  - Font font1 = new Font("SansSerif", 16);
  - Font font2 = Font.font("Times New Roman", FontWeight.BOLD, FontPosture.ITALIC, 12);

Example - Font

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.*;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.scene.text.*;
7  import javafx.scene.control.*;
8  import javafx.stage.Stage;
9
10 public class FontDemo extends Application {
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     // Create a pane to hold the circle
14     Pane pane = new StackPane();
15
16     // Create a circle and set its properties
17     Circle circle = new Circle();
18     circle.setRadius(50);
19     circle.setStroke(Color.BLACK);
20     circle.setFill(new Color(0.5, 0.5, 0.5, 0.1));
21     pane.getChildren().add(circle); // Add circle to the pane
22
23     // Create a label and set its properties
24     Label label = new Label("JavaFX");
25     label.setFont(Font.font("Times New Roman",
26       FontWeight.BOLD, FontPposture.ITALIC, 20));
27     pane.getChildren().add(label);
28
29     // Create a scene and place it in the stage
30     Scene scene = new Scene(pane);
31     primaryStage.setTitle("FontDemo"); // Set the stage title
32     primaryStage.setScene(scene); // Place the scene in the stage
33     primaryStage.show(); // Display the stage
34   }
35 }
```

create a StackPane

create a Circle

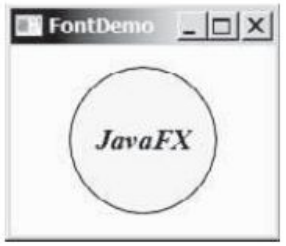
create a Color

add circle to the pane

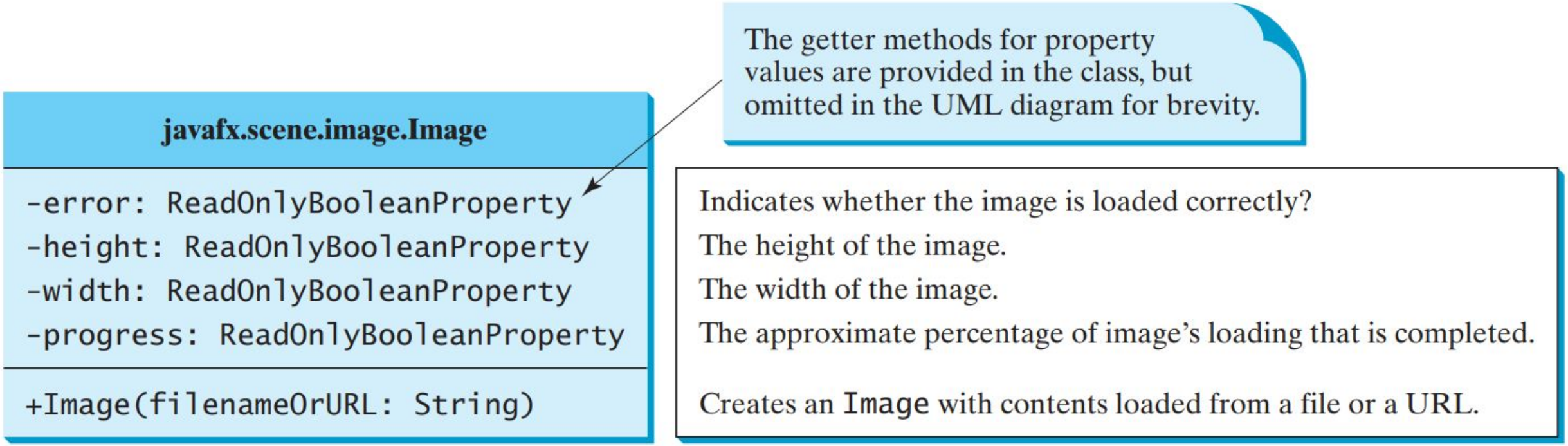
create a label

create a font

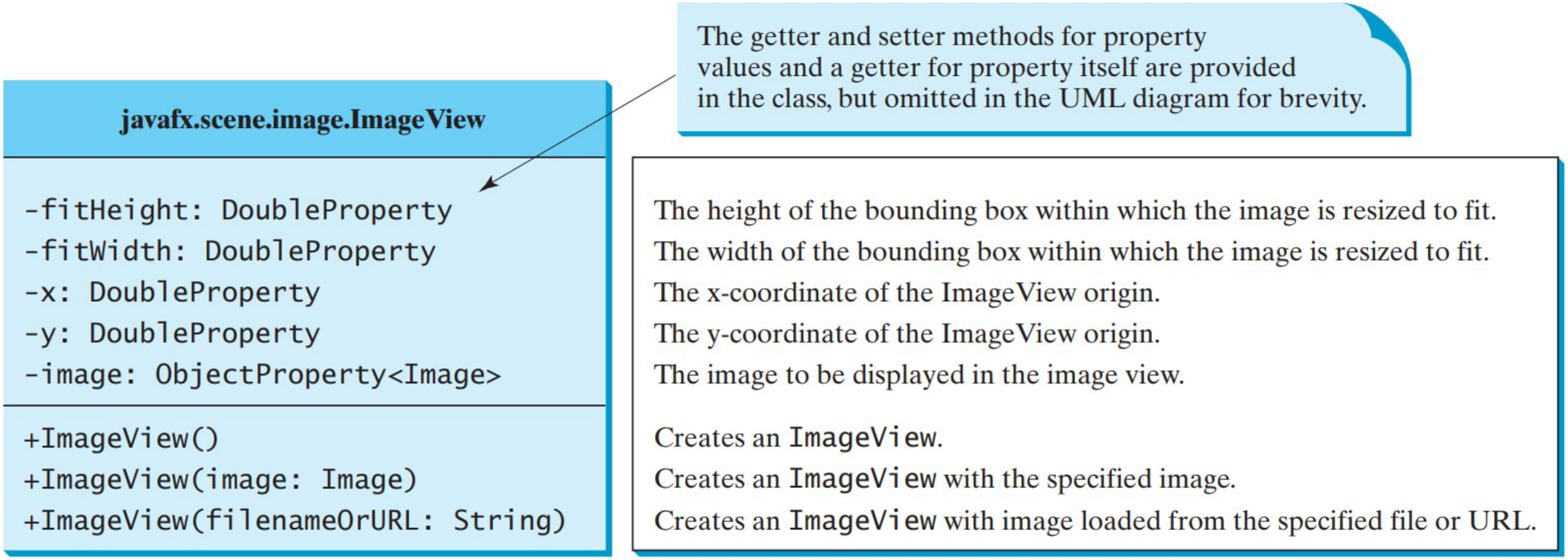
add label to the pane



javaFX : Image Class – encapsulate image information



javaFX : ImageView Class – for displaying an image



JavaFX – Image

- You can load and modify images using the classes provided by JavaFX in the package `javafx.scene.image`.
- JavaFX supports the image formats like Bmp, Gif, Jpeg, Png.
- Class Image of `javafx.scene.image` package is used to load an image
- Any of the following argument is required to the constructor of the class
  - An InputStream object of the image to be loaded or

```
FileInputStream inputstream = new FileInputStream ("C:\\image.jpg");
Image image = new Image(inputstream);
```

  - A string variable holding the URL for the image.

```
Image image = new Image("http://sample.com/res/flower.png");
```

- After loading image in Image object, view is set to load the image using `ImageView` class

```
ImageView imageView = new ImageView(image);
```

Color Example (Image)

```
import java.io.FileInputStream; import javafx.application.Application;
import javafx.scene.Group; import javafx.scene.Scene; import javafx.scene.image.Image;
import javafx.scene.image.ImageView; import javafx.stage.Stage;

public class ImageExample extends Application {
    public void start(Stage stage) throws Exception {
        Image image = new Image(new FileInputStream("F://bhavya.jpg"));
        ImageView imageView = new ImageView(image);
        imageView.setX(50);
        imageView.setY(25);
        imageView.setFitHeight(455);
        imageView.setFitWidth(500);
        imageView.setPreserveRatio(true);
        Group root = new Group(imageView);
        Scene scene = new Scene(root, 600, 500);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String args[]) { launch(args); }
}
```

Layout Panes

- After constructing all the required nodes in a scene, we will generally arrange them in order.
- This arrangement of the components within the container is called the Layout of the container.
- JavaFX provides several predefined layouts such as HBox, VBox, Border Pane, Stack Pane, Text Flow, Anchor Pane, Title Pane, Grid Pane, Flow Panel, etc.
- Each of the above mentioned layout is represented by a class and all these classes belongs to the package `javafx.layout`. The class named `Pane` is the base class of all the layouts in JavaFX.

Example : StackPane - Pane

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5  import javafx.scene.layout.StackPane;
6
7  public class ButtonInPane extends Application {
8    @Override // Override the start method in the Application class
9    public void start(Stage primaryStage) {
10     // Create a scene and place a button in the scene
11     StackPane pane = new StackPane();
12     pane.getChildren().add(new Button("OK"));
13     Scene scene = new Scene(pane, 200, 50);
14     primaryStage.setTitle("Button in a pane"); // Set the stage title
15     primaryStage.setScene(scene); // Place the scene in the stage
16     primaryStage.show(); // Display the stage
17   }
18 }
```

- The program creates a **StackPane** (line 11) and adds a button as a child of the pane (line 12).
- The **getChildren()** method returns an instance of **javafx.collections.ObservableList**.
- **ObservableList** behaves very much **like an ArrayList** for storing a collection of elements.
- Invoking **add(e)** adds an element to the list.
- The StackPane places the nodes in the center of the pane on top of each other

Layout Panes (`javafx.scene.layout`)

Sr.	Shape & Description
1	HBox • The HBox layout arranges all the nodes in our application in a single horizontal row. • The class named HBox of the package javafx.scene.layout represents the text horizontal box layout.
2	VBox • The VBox layout arranges all the nodes in our application in a single vertical column. • The class named VBox of the package javafx.scene.layout represents the text Vertical box layout.
3	BorderPane • The Border Pane layout arranges the nodes in our application in top, left, right, bottom and center positions. • The class named BorderPane of the package javafx.scene.layout represents the border pane layout.

Layout Panes (`javafx.scene.layout`) (Cont.)

Sr.	Shape & Description
4	StackPane • The stack pane layout arranges the nodes in our application on top of another just like in a stack. The node added first is placed at the bottom of the stack and the next node is placed on top of it. • The class named StackPane of the package javafx.scene.layout represents the stack pane layout.
5	TextFlow • The Text Flow layout arranges multiple text nodes in a single flow. • The class named TextFlow of the package javafx.scene.layout represents the text flow layout.
6	AnchorPane • The Anchor pane layout anchors the nodes in our application at a particular distance from the pane. • The class named AnchorPane of the package javafx.scene.layout represents the Anchor Pane layout.

Layout Panes (`javafx.scene.layout`) (Cont.)

Sr.	Shape & Description
7	TilePane • The Tile Pane layout adds all the nodes of application in the form of uniformly sized tiles. • The class named TilePane of the package javafx.scene.layout represents the TilePane layout.
8	GridPane • The Grid Pane layout arranges the nodes in our application as a grid of rows and columns. This layout comes handy while creating forms. • The class named GridPane of the package javafx.scene.layout represents the GridPane layout.
9	FlowPane • The flow pane layout wraps all the nodes in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width. • The class named FlowPane of the package javafx.scene.layout represents the Flow Pane layout.

Creating a Layout

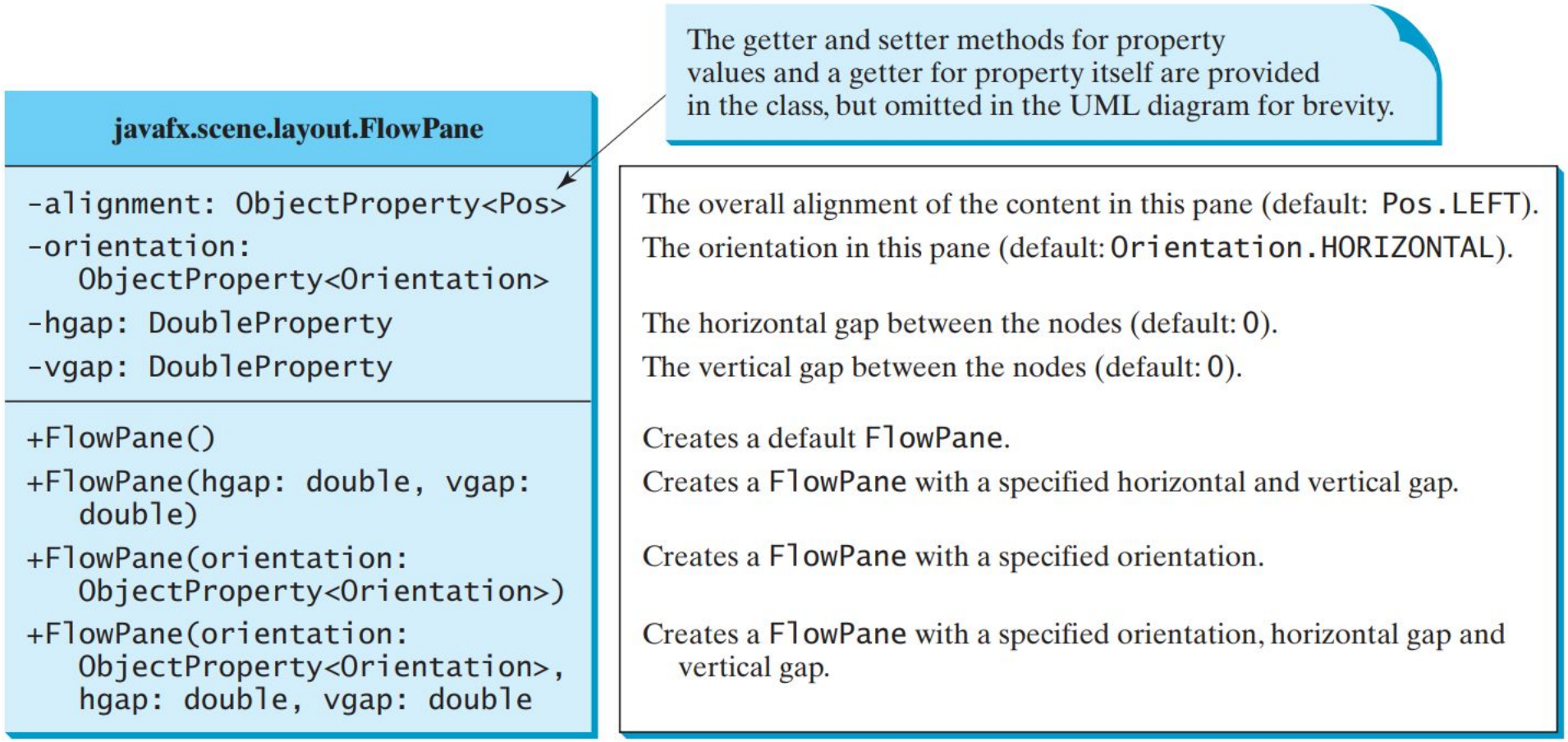
- To create a layout, you need to –
  - Create nodes.
  - Instantiate the respective class of the required layout.
  - Set the properties of the layout.
  - Add all the created nodes to the layout.

```
public void start(Stage stage) {
    TextField textField = new TextField();
    Button playButton = new Button("Play");
    Button stopButton = new Button("stop");
    HBox hbox = new HBox();
    hbox.setSpacing(10);
    hbox.setMargin(textField, new Insets(20, 20, 20, 20));
    hbox.setMargin(playButton, new Insets(20, 20, 20, 20));
    hbox.setMargin(stopButton, new Insets(20, 20, 20, 20));
    ObservableList<Node> list = hbox.getChildren();
    list.addAll(textField, playButton, stopButton);
    Scene scene = new Scene(hbox);
    stage.setScene(scene);
    stage.show();
}
```

Layout – Pane - Container

Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

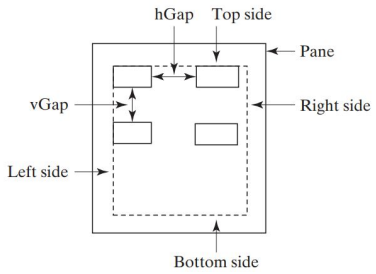
Pane - FlowPane



`FlowPane` lays out nodes row by row horizontally or column by column vertically.

Example : FlowPane

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.TextField;
6  import javafx.scene.layout.FlowPane;
7  import javafx.stage.Stage;
8
9  public class ShowFlowPane extends Application {                    extend Application
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     // Create a pane and set its properties
13     FlowPane pane = new FlowPane();                                 create FlowPane
14     pane.setPadding(new Insets(11, 12, 13, 14));
15     pane.setHgap(5);
16     pane.setVgap(5);
17
18     // Place nodes in the pane
19     pane.getChildren().addAll(new Label("First Name:"),             add UI controls to pane
20       new TextField(), new Label("MI:"));
21     TextField tfMi = new TextField();
22     tfMi.setPrefColumnCount(1);
23     pane.getChildren().addAll(tfMi, new Label("Last Name:"),
24       new TextField());
25
26     // Create a scene and place it in the stage
27     Scene scene = new Scene(pane, 200, 250);                        add pane to scene
28     primaryStage.setTitle("ShowFlowPane"); // Set the stage title
29     primaryStage.setScene(scene); // Place the scene in the stage   place scene to stage
30     primaryStage.show(); // Display the stage                       display stage
31   }
32 }
```



- The program creates a **FlowPane** (line 13) and sets its padding property with an Insets object (line 14).
- An **Insets** object specifies the size of the border of a pane. The constructor **Insets(11, 12, 13, 14)** creates an Insets with the border sizes for **top (11), right (12), bottom (13), and left (14)** in pixels.
- You can also use the constructor **Insets(value)** to create an Insets with the **same value for all four** sides.
- The **hGap** and **vGap** properties are in lines 15–16 to specify the horizontal gap and vertical gap between two nodes in the pane.

Pane - GridPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.GridPane

```
-alignment: ObjectProperty<Pos>
-gridLinesVisible: BooleanProperty
-hgap: DoubleProperty
-vgap: DoubleProperty

+GridPane()
+add(child: Node, columnIndex: int, rowIndex: int): void
+addColumn(columnIndex: int, children: Node...): void
+addRow(rowIndex: int, children: Node...): void
+getColumnIndex(child: Node): int
+setColumnIndex(child: Node, columnIndex: int): void
+getRowIndex(child:Node): int
+setRowIndex(child: Node, rowIndex: int): void
+setHalighnment(child: Node, value: HPos): void
+setValighnment(child: Node, value: VPos): void
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).
Is the grid line visible? (default: `false`)

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a `GridPane`.
Adds a node to the specified column and row.

Adds multiple nodes to the specified column.

Adds multiple nodes to the specified row.

Returns the column index for the specified node.

Sets a node to a new column. This method repositions the node.

Returns the row index for the specified node.
Sets a node to a new row. This method repositions the node.

Sets the horizontal alignment for the child in the cell.

Sets the vertical alignment for the child in the cell.

The `GridPane` places the nodes in a grid with a specified column and row

`GridPane` lays out nodes in the specified cell in a grid.

Example : GridPane

```
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.TextField;
9  import javafx.scene.layout.GridPane;
10 import javafx.stage.Stage;
11
12 public class ShowGridPane extends Application {
13   @Override // Override the start method in the Application class
14   public void start(Stage primaryStage) {
15     // Create a pane and set its properties
16     GridPane pane = new GridPane();                    create a grid pane
17     pane.setAlignment(Pos.CENTER);                     set properties
18     pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
19     pane.setHgap(5.5);
20     pane.setVgap(5.5);
21
22     // Place nodes in the pane
23     pane.add(new Label("First Name:"), 0, 0);          add label
24     pane.add(new TextField(), 1, 0);                   add text field
25     pane.add(new Label("MI:"), 0, 1);
26     pane.add(new TextField(), 1, 1);
27     pane.add(new Label("Last Name:"), 0, 2);
28     pane.add(new TextField(), 1, 2);
29     Button btAdd = new Button("Add Name");
30     pane.add(btAdd, 1, 3);                             add button
31     GridPane.setHalignment(btAdd, HPos.RIGHT);         align button right
32
33     // Create a scene and place it in the stage
34     Scene scene = new Scene(pane);                     create a scene
35     primaryStage.setTitle("ShowGridPane"); // Set the stage title
36     primaryStage.setScene(scene); // Place the scene in the stage
37     primaryStage.show(); // Display the stage          display stage
38   }
39 }
```

- The program creates a **GridPane** (line 16) and sets its properties (line 17–20).
- The **alignment is set to the center position (line 17)**, which causes the nodes to be placed in the center of the grid pane.
- If you resize the window, you will see the nodes remains in the center of the grid pane.
- **The program adds the label in column 0 and row 0 (line 23). The column and row index starts from 0. The add method places a node in the specified column and row. Not every cell in the grid needs to be filled.**
- A button is placed in column 1 and row 3 **(line 30)**, but there are no nodes placed in column 0 and row 3. To remove a node from a GridPane, use pane. **getChildren().remove(node)**. To remove all nodes, use **pane.getChildren(). removeAll()**.
- The program invokes the static **setHalignment** method to align the button right in the cell **(line 31)**.
- Note that the **scene size is not set (line 34)**. In this case, the scene size is automatically computed according to the sizes of the nodes placed inside the scene.

Pane - BorderPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.BorderPane

```
-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>
```

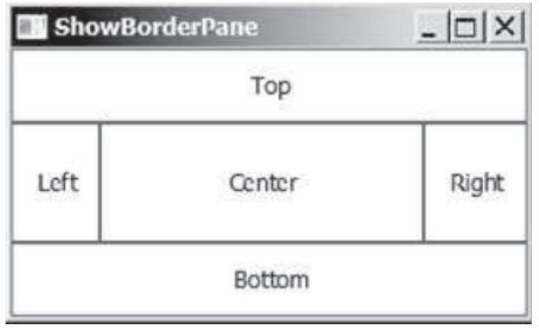
```
+BorderPane()
+setAlignment(child: Node, pos: Pos)
```

The node placed in the top region (default: `null`).
The node placed in the right region (default: `null`).
The node placed in the bottom region (default: `null`).
The node placed in the left region (default: `null`).
The node placed in the center region (default: `null`).

Creates a `BorderPane`.

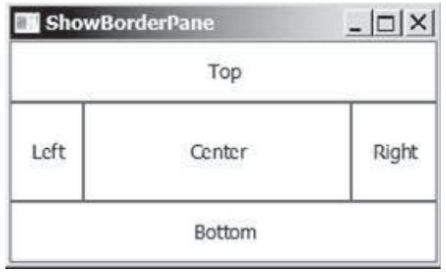
Sets the alignment of the node in the `BorderPane`.

BorderPane places the nodes in top, bottom, left, right, and center regions.



Example : BorderPane

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.BorderPane;
6  import javafx.scene.layout.StackPane;
7  import javafx.stage.Stage;
8
9  public class ShowBorderPane extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     // Create a border pane
13     BorderPane pane = new BorderPane();                    create a border pane
14
15     // Place nodes in the pane
16     pane.setTop(new CustomPane("Top"));                    add to top
17     pane.setRight(new CustomPane("Right"));                add to right
18     pane.setBottom(new CustomPane("Bottom"));              add to bottom
19     pane.setLeft(new CustomPane("Left"));                  add to left
20     pane.setCenter(new CustomPane("Center"));              add to center
21
22     // Create a scene and place it in the stage
23     Scene scene = new Scene(pane);
24     primaryStage.setTitle("ShowBorderPane"); // Set the stage title
25     primaryStage.setScene(scene); // Place the scene in the stage
26     primaryStage.show(); // Display the stage
27   }
28 }
29
30 // Define a custom pane to hold a label in the center of the pane
31 class CustomPane extends StackPane {                      define a custom pane
32   public CustomPane(String title) {
33     getChildren().add(new Label(title));                   add a label to pane
34     setStyle("-fx-border-color: red");                     set style
35     setPadding(new Insets(11.5, 12.5, 13.5, 14.5));        set padding
36   }
37 }
```



- A **BorderPane** can place nodes in five regions: top, bottom, left, right, and center, using the **setTop(node), setBottom(node), setLeft(node), setRight(node),** and **setCenter(node)** methods. The class diagram for GridPane
- The program defines **CustomPane that extends StackPane (line 31)**. The constructor of CustomPane adds a label with the specified title (line 33), sets a style for the border color, and sets a padding using insets (line 35).
- The program creates a **BorderPane (line 13)** and places five instances of CustomPane into five regions of the border pane **(lines 16–20)**.
- Note that a pane is a node. So a pane can be added into another pane.
- To remove a node from the top region, invoke **setTop(null)**.
- If a region is not occupied, no space will be allocated for this region.

Pane – Hbox / Vbox Pane

javafx.scene.layout.HBox

```
-alignment: ObjectProperty<Pos>
-fillHeight: BooleanProperty
-spacing: DoubleProperty
+HBox()
+HBox(spacing: double)
+setMargin(node: Node, value:
   Insets): void
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the children in the box (default: `Pos.TOP_LEFT`).
Is resizable children fill the full height of the box (default: `true`).
The horizontal gap between two nodes (default: 0).

Creates a default `HBox`.
Creates an `HBox` with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

HBox places the nodes in one row.

javafx.scene.layout.VBox

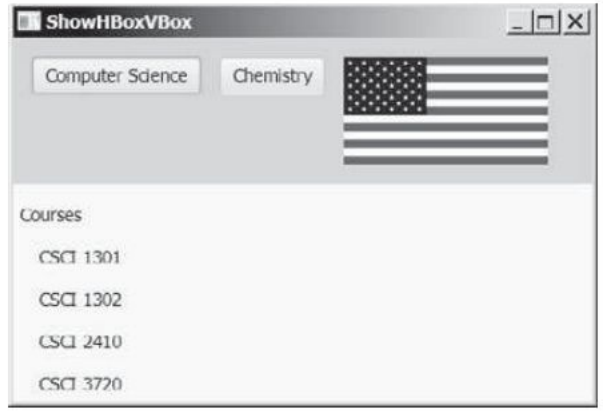
```
-alignment: ObjectProperty<Pos>
-fillWidth: BooleanProperty
-spacing: DoubleProperty
+VBox()
+VBox(spacing: double)
+setMargin(node: Node, value:
   Insets): void
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the children in the box (default: `Pos.TOP_LEFT`).
Is resizable children fill the full width of the box (default: `true`).
The vertical gap between two nodes (default: 0).

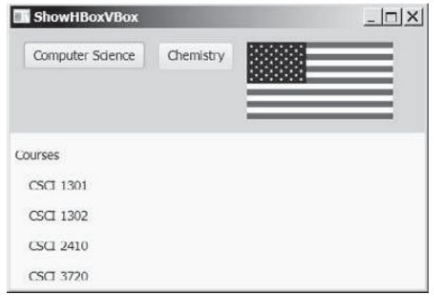
Creates a default `VBox`.
Creates a `VBox` with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

VBox places the nodes in one column.



Example : Hbox / Vbox Pane

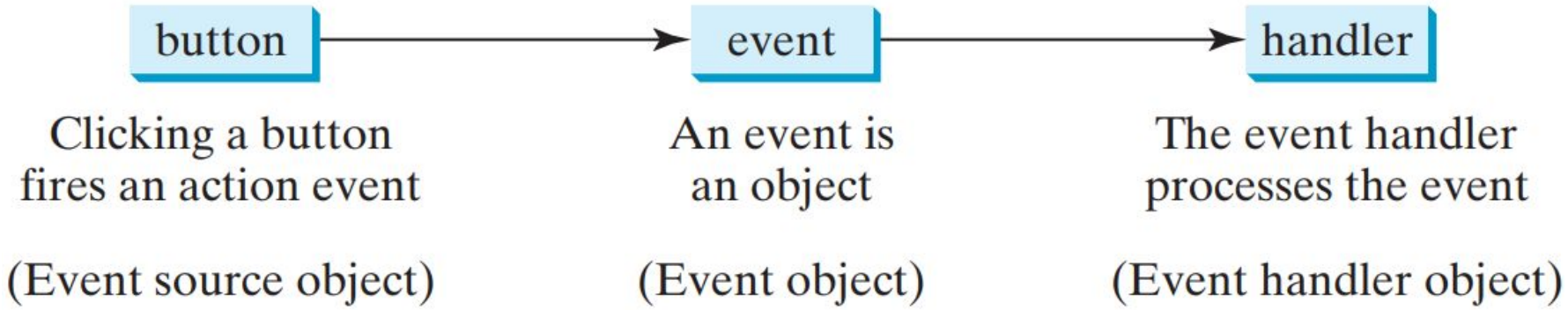
```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.control.Label;
6  import javafx.scene.layout.BorderPane;
7  import javafx.scene.layout.HBox;
8  import javafx.scene.layout.VBox;
9  import javafx.stage.Stage;
10 import javafx.scene.image.Image;
11 import javafx.scene.image.ImageView;
12
13 public class ShowHBoxVBox extends Application {
14   @Override // Override the start method in the Application class
15   public void start(Stage primaryStage) {
16     // Create a border pane
17     BorderPane pane = new BorderPane();                    create a border pane
18
19     // Place nodes in the pane
20     pane.setTop(getHBox());                                add an HBox to top
21     pane.setLeft(getVBox());                               add a VBox to left
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane);                         create a scene
25     primaryStage.setTitle("ShowHBoxVBox"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage              display stage
28   }
29
30   private HBox getHBox() {                                 getHBox
31     HBox hBox = new HBox(15);
32     hBox.setPadding(new Insets(15, 15, 15, 15));
33     hBox.setStyle("-fx-background-color: gold");
34     hBox.getChildren().add(new Button("Computer Science")); add buttons to HBox
35     hBox.getChildren().add(new Button("Chemistry"));
36     ImageView imageView = new ImageView(new Image("image/us.gif"));
37     hBox.getChildren().add(imageView);
38     return hBox;                                           return an HBox
39   }
40
41   private VBox getVBox() {                                 getVBox
42     VBox vBox = new VBox(15);
43     vBox.setPadding(new Insets(15, 5, 5, 5));
44     vBox.getChildren().add(new Label("Courses"));          add a label
45
46     Label[] courses = {new Label("CSCI 1301"), new Label("CSCI 1302"),
47         new Label("CSCI 2410"), new Label("CSCI 3720")};
48
49     for (Label course: courses) {
50       VBox.setMargin(course, new Insets(0, 0, 0, 15));     set margin
51       vBox.getChildren().add(course);                      add a label
52     }
53
54     return vBox;                                           return vBox
55   }
56 }
```



- An **HBox** lays out its children in a single horizontal row.
- A **VBox** lays out its children in a single vertical column.
- Recall that a **FlowPane** can lay out its children in multiple rows or multiple columns, but an **HBox** or a **VBox can lay out children only in one row or one column**.
- The program defines the **getHBox()** method. This method **returns** an **HBox** that contains two buttons and an image view **(lines 30–39)**.
- The **background color of the HBox is set to gold using Java CSS (line 33)**.
- The program defines the **getVBox()** method. This method returns a **VBox** that contains five labels **(lines 41–55)**.
- The first label is added to the VBox in line 44 and the other four are added in line 51.
- The **setMargin()** method is used to set a node's margin when placed inside the VBox (line 50).

Event-driven programming

- To **respond to a button click**, you need to write the code to process the **button-clicking action**.
- The **button** is an **event source object**—where the action originates.
- You need to **create an object** capable of **handling the action event** on a button. This object is called an **event handler**, as shown in Figure



An event handler processes the event fired from the source object.

Event-driven programming

- To be a **handler of an action event**, two requirements must be met:
  - 1. The object must be an instance of the **EventHandler<T extends Event>** interface. This interface defines the common behavior for all handlers denotes that **T** is a **generic type** that is a **subtype of Event**.
  - 2. The **EventHandler** object handler must be registered with the event source object using the method **source.setOnAction(handler)**.
- The **EventHandler<ActionEvent>** interface contains the **handle(ActionEvent)** method for processing the action event. Your **handler class must override this method to respond to the event.**

Example : Event-Driven

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.layout.HBox;
6  import javafx.stage.Stage;
7  import javafx.event.ActionEvent;
8  import javafx.event.EventHandler;
9
10 public class HandleEvent extends Application {
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     // Create a pane and set its properties
14     HBox pane = new HBox(10);
15     pane.setAlignment(Pos.CENTER);
16     Button btOK = new Button("OK");
17     Button btCancel = new Button("Cancel");
18     OKHandlerClass handler1 = new OKHandlerClass();          create handler
19     btOK.setOnAction(handler1);                              register handler
20     CancelHandlerClass handler2 = new CancelHandlerClass();  create handler
21     btCancel.setOnAction(handler2);                          register handler
22     pane.getChildren().addAll(btOK, btCancel);
23
24     // Create a scene and place it in the stage
25     Scene scene = new Scene(pane);
26     primaryStage.setTitle("HandleEvent"); // Set the stage title
27     primaryStage.setScene(scene); // Place the scene in the stage
28     primaryStage.show(); // Display the stage
29   }
30 }
31
32 class OKHandlerClass implements EventHandler<ActionEvent> {   handler class
33   @Override
34   public void handle(ActionEvent e) {                        handle event
35     System.out.println("OK button clicked");
36   }
37 }
38
39 class CancelHandlerClass implements EventHandler<ActionEvent> {   handler class
40   @Override
41   public void handle(ActionEvent e) {                        handle event
42     System.out.println("Cancel button clicked");
43   }
44 }
```

- **Two handler classes** are defined in lines 32–44.
- Each handler class implements **EventHandler<ActionEvent>** to process **ActionEvent**.
- The object **handler1** is an instance of **OKHandlerClass** (line 18), which is registered with the button **btOK** (line 19). When the OK button is clicked, the **handle(ActionEvent) method** (line 34) in **OKHandlerClass** is invoked to process the event.
- The object **handler2** is an instance of **CancelHandlerClass** (line 20), which is registered with the button **btCancel** in line 21. When the Cancel button is clicked, the **handle(ActionEvent)** method (line 41) in **CancelHandlerClass** is invoked to process the event.

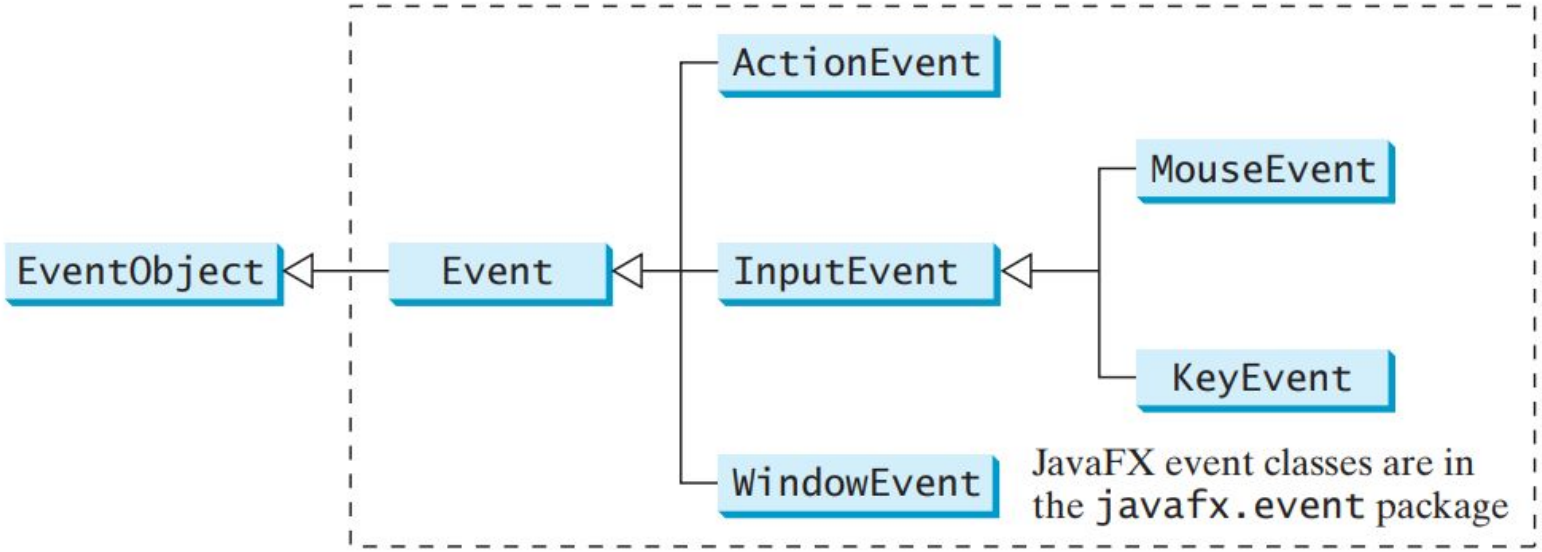
(a)

(b)

JavaFX - Events

- An event is an **object created from an event source**.
- Firing an event means to create an event and delegate the handler to handle the event.
- An event can be defined as a **signal to the program that something has happened**.
- Events are **triggered by external user actions**, such as **mouse movements**, **mouse clicks**, and **keystrokes**.
- The component that creates an event and fires it is called the **event source object**, or simply **source object** or **source component/node**
- The root class of the **Java event** classes is **java.util.EventObject**.
- The root class of the **JavaFX event** classes is **javafx.event.Event**

JavaFX - Event



- An **event object** contains whatever properties are pertinent to the event.
- You **can identify the source object of an event** using the **getSource()** instance method in the **EventObject** class.
- The subclasses of **EventObject** deal with specific types of events, such as action events, window events, mouse events, and key events

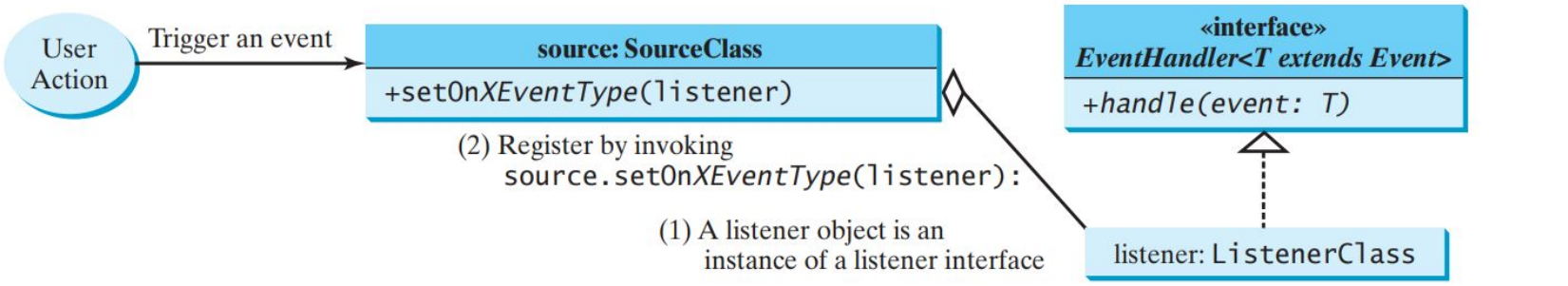
JavaFX - Event

User Action, Source Object, Event Type, Handler Interface, and Handler

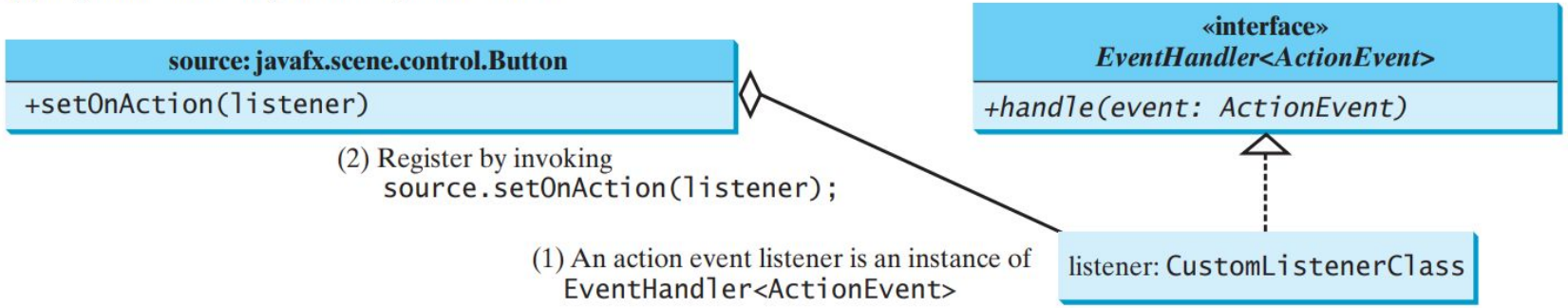
User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

Registering Handlers and Handling Event

- A **handler** is **an object** that **must be registered with an event source object**, and it **must be an instance of an appropriate event-handling interface**
- Java uses a delegation-based model for event handling: a **source object fires an event**, and **an object interested in the event handles it**. The latter object is **called an event handler or an event listener**.



(a) A generic source object with a generic event T



(b) A `Button` source object with an `ActionEvent`

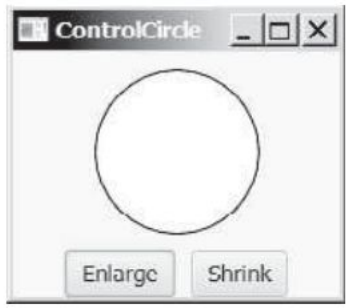
A listener must be an instance of a listener interface and must be registered with a source object.

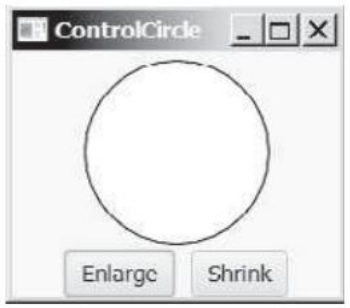
Registering Handlers and Handling Event

- The ***handler object*** **must** be an ***instance of the corresponding event-handler interface*** to ensure that the handler has the correct method for processing the event.
- JavaFX defines a unified handler interface **EventHandler<T extends Event>** for an event **T**.
- The **handler** interface **contains** the **handle(T e)** method for processing the event.
  - For example, the handler interface for **ActionEvent** is **EventHandler<ActionEvent>;** each handler for **ActionEvent** should implement the **handle(ActionEvent e)** method for processing an ActionEvent.
- The ***handler object*** **must** be ***registered by the source object***.
- Registration methods depend on the event type.
  - For **ActionEvent**, the method is **setOnAction()**.
  - For a **mouse pressed event**, the method is **setOnMousePressed()**.
  - For a **key pressed event**, the method is **setOnKeyPressed()**.

Example

```
5  import javafx.scene.layout.StackPane;
6  import javafx.scene.layout.HBox;
7  import javafx.scene.layout.BorderPane;
8  import javafx.scene.paint.Color;
9  import javafx.scene.shape.Circle;
10 import javafx.stage.Stage;
11
12 public class ControlCircleWithoutEventHandling extends Application {
13   @Override // Override the start method in the Application class
14   public void start(Stage primaryStage) {
15     StackPane pane = new StackPane();
16     Circle circle = new Circle(50);                          circle
17     circle.setStroke(Color.BLACK);
18     circle.setFill(Color.WHITE);
19     pane.getChildren().add(circle);
20
21     HBox hBox = new HBox();
22     hBox.setSpacing(10);
23     hBox.setAlignment(Pos.CENTER);
24     Button btEnlarge = new Button("Enlarge");                buttons
25     Button btShrink = new Button("Shrink");
26     hBox.getChildren().add(btEnlarge);
27     hBox.getChildren().add(btShrink);
28
29     BorderPane borderPane = new BorderPane();
30     borderPane.setCenter(pane);
31     borderPane.setBottom(hBox);
32     BorderPane.setAlignment(hBox, Pos.CENTER);
33
34     // Create a scene and place it in the stage
35     Scene scene = new Scene(borderPane, 200, 150);
36     primaryStage.setTitle("ControlCircle"); // Set the stage title
37     primaryStage.setScene(scene); // Place the scene in the stage
38     primaryStage.show(); // Display the stage
39   }
49 }
```





Example

Handler and its registration

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.layout.StackPane;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.layout.BorderPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Circle;
12 import javafx.stage.Stage;
13
14 public class ControlCircle extends Application {
15   private CirclePane circlePane = new CirclePane();
16
17   @Override // Override the start method in the Application class
18   public void start(Stage primaryStage) {
19     // Hold two buttons in an HBox
20     HBox hBox = new HBox();
21     hBox.setSpacing(10);
22     hBox.setAlignment(Pos.CENTER);
23     Button btEnlarge = new Button("Enlarge");
24     Button btShrink = new Button("Shrink");
25     hBox.getChildren().add(btEnlarge);
26     hBox.getChildren().add(btShrink);
27
28     // Create and register the handler
29     btEnlarge.setOnAction(new EnlargeHandler());
30
31     BorderPane borderPane = new BorderPane();
32     borderPane.setCenter(circlePane);
33     borderPane.setBottom(hBox);
34     BorderPane.setAlignment(hBox, Pos.CENTER);
35
36     // Create a scene and place it in the stage
37     Scene scene = new Scene(borderPane, 200, 150);
38     primaryStage.setTitle("ControlCircle"); // Set the stage title
39     primaryStage.setScene(scene); // Place the scene in the stage
40     primaryStage.show(); // Display the stage
41   }
42
43   class EnlargeHandler implements EventHandler<ActionEvent> {
44     @Override // Override the handle method
45     public void handle(ActionEvent e) {
46       circlePane.enlarge();
47     }
48   }
49 }
```

create/register handler

handler class

CirclePane class

enlarge method

```
50
51 class CirclePane extends StackPane {
52   private Circle circle = new Circle(50);
53
54   public CirclePane() {
55     getChildren().add(circle);
56     circle.setStroke(Color.BLACK);
57     circle.setFill(Color.WHITE);
58   }
59
60   public void enlarge() {
61     circle.setRadius(circle.getRadius() + 2);
62   }
63
64   public void shrink() {
65     circle.setRadius(circle.getRadius() > 2 ?
66       circle.getRadius() - 2 : circle.getRadius());
67   }
68 }
```

Inner Class

```java
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```java
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)

```java
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

- An inner class, or nested class, is a class defined within the scope of another class.
- Inner classes are useful for defining handler classes.
- Normally, you define a class as an inner class if it is used only by its outer class

Inner Class - Features

- An inner class is **compiled into** a class named **OuterClassName$InnerClassName.class**. For example, the **inner class A in Test is compiled** into **Test$A.class**
- An inner class **can reference the data and the methods defined in the outer class** in which it nests, so you **need not pass the reference of an object of the outer class to the constructor of the inner class**.
- An inner class can be defined with a **visibility modifier subject to the same visibility rules** applied to a member of the class.
- An inner class can be **defined as static**. A **static inner class can be accessed using the outer class name**. A static inner class **cannot access nonstatic members of the outer class**.
- **Objects** of an **inner class** are often **created in the outer class**. **But** you can **also create an object of an inner class from another class**.
  - If the inner class is **nonstatic**, you **must first create an instance of the outer class**, then use the following syntax to create an object for the inner class:
    - `OuterClass.InnerClass innerObject = outerObject.new InnerClass();`
  - If the inner class is **static**, use the following syntax to create an object for it:
    - `OuterClass.InnerClass innerObject = new OuterClass.InnerClass();`

Inner Class - Features

- A simple use of inner classes is **to combine dependent classes into a primary class**. This reduces the number of source files. It also makes class files easy to organize since they are all named with the primary class as the prefix.
- Another practical use of inner classes is to **avoid class-naming conflicts**
- A **handler class** is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is **appropriate to be defined inside the main class as an inner class**.

Anonymous Inner Class Handler

- An anonymous inner class is an **inner class without a name**. It **combines defining an inner class and creating an instance of the class into one step**.
- The syntax for an anonymous inner class is shown below
  - ```
    new SuperClassName/InterfaceName() {
        // Implement or override methods in superclass or interface
        // Other methods if necessary
       }
    ```

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new class EnlargeHandler
      implements EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

(b) Anonymous inner class
```

Anonymous Inner Class - feature

- An anonymous inner class **must always extend a superclass or implement an interface**, **but it cannot have an explicit extends or implements clause**.
- An anonymous inner class **must implement all the abstract methods** in the superclass or in the interface.
- An anonymous inner class **always uses the no-arg constructor from its superclass to create an instance**. If an anonymous **inner class implements an interface**, the constructor is **Object()**.
- An anonymous inner class **is compiled into** a class named **OuterClassName$n.class**. For example, if the outer class Test has two anonymous inner classes, they are compiled into Test$1.class and Test$2.class

Example

anonymous handler

handle event

```
10  public class AnonymousHandlerDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      // Hold two buttons in an HBox
14      HBox hBox = new HBox();
15      hBox.setSpacing(10);
16      hBox.setAlignment(Pos.CENTER);
17      Button btNew = new Button("New");
18      Button btOpen = new Button("Open");
19      Button btSave = new Button("Save");
20      Button btPrint = new Button("Print");
21      hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);
22
23      // Create and register the handler
24      btNew.setOnAction(new EventHandler<ActionEvent>() {
25        @Override // Override the handle method
26        public void handle(ActionEvent e) {
27          System.out.println("Process New");
28        }
29      });
30
31      btOpen.setOnAction(new EventHandler<ActionEvent>() {
32        @Override // Override the handle method
33        public void handle(ActionEvent e) {
34          System.out.println("Process Open");
35        }
36      });
37
38      btSave.setOnAction(new EventHandler<ActionEvent>() {
39        @Override // Override the handle method
40        public void handle(ActionEvent e) {
41          System.out.println("Process Save");
42        }
43      });
44
45      btPrint.setOnAction(new EventHandler<ActionEvent>() {
46        @Override // Override the handle method
47        public void handle(ActionEvent e) {
48          System.out.println("Process Print");
49        }
50      });
51
52      // Create a scene and place it in the stage
53      Scene scene = new Scene(hBox, 300, 50);
54      primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
55      primaryStage.setScene(scene); // Place the scene in the stage
56      primaryStage.show(); // Display the stage
57    }
58  }
```

Simplifying Event Handling using Lambda Expression

- Lambda expressions can be used to greatly simplify coding for event handling.
- Lambda expression is a new feature in Java 8.
- Lambda expressions can be viewed as an anonymous class with a concise syntax
- The basic syntax for a lambda expression is
  - either

```
(type1 param1, type2 param2, ...) -> expression
```

  - or

```
(type1 param1, type2 param2, ...) -> { statements; }
```

- The data type for a parameter may be explicitly declared or implicitly inferred by the compiler.
- The parentheses can be omitted if there is only one parameter without an explicit data type.
- Example, the lambda expression is as follows
  -

```
e -> {
    // Code for processing event e
}
```

Simplifying Event Handling using Lambda Expression

```
btEnlarge.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      // Code for processing event e
    }
  }
});
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

(b) Lambda expression event handler

Example

lambda handler

lambda handler

lambda handler

lambda handler

```
 9  public class LambdaHandlerDemo extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) {
12      // Hold two buttons in an HBox
13      HBox hBox = new HBox();
14      hBox.setSpacing(10);
15      hBox.setAlignment(Pos.CENTER);
16      Button btNew = new Button("New");
17      Button btOpen = new Button("Open");
18      Button btSave = new Button("Save");
19      Button btPrint = new Button("Print");
20      hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);
21
22      // Create and register the handler
23      btNew.setOnAction((ActionEvent e) -> {
24        System.out.println("Process New");
25      });
26
27      btOpen.setOnAction((e) -> {
28        System.out.println("Process Open");
29      });
30
31      btSave.setOnAction(e -> {
32        System.out.println("Process Save");
33      });
34
35      btPrint.setOnAction(e -> System.out.println("Process Print"));
36
37      // Create a scene and place it in the stage
38      Scene scene = new Scene(hBox, 300, 50);
39      primaryStage.setTitle("LambdaHandlerDemo"); // Set title
40      primaryStage.setScene(scene); // Place the scene in the stage
41      primaryStage.show(); // Display the stage
42    }
43  }
```

JavaFX - Events

- In GUI applications, web applications and graphical applications, whenever a user interacts with the application (nodes), an event is said to have been occurred.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.
- JavaFX provides support to handle a wide varieties of events. The class named `Event` of the package `javafx.event` is the base class for an event.
- JavaFX provides a wide variety of events. Some of them are as follows:
  1. Mouse Event – occurs when a mouse is clicked.
     - Class – `MouseEvent`
     - Actions - mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target, etc.
  2. Key Event – indicates the key stroke occurred on a node.
     - Class – `KeyEvent`
     - Actions - key pressed, key released and key typed.
  3. Drag Event – occurs when the mouse is dragged.
     - Class - `DragEvent`.
     - Actions - drag entered, drag dropped, drag entered target, drag exited target, drag over, etc.
  4. Window Event – occurs when window showing/hiding takes place.
     - Class – `WindowEvent`
     - Actions – window hiding, window shown, window hidden, window showing, etc.

MouseEvent Class

javafx.scene.input.MouseEvent	
+getButton(): MouseButton	Indicates which mouse button has been clicked.
+getClickCount(): int	Returns the number of mouse clicks associated with this event.
+getX(): double	Returns the *x*-coordinate of the mouse point in the event source node.
+getY(): double	Returns the *y*-coordinate of the mouse point in the event source node.
+getSceneX(): double	Returns the *x*-coordinate of the mouse point in the scene.
+getSceneY(): double	Returns the *y*-coordinate of the mouse point in the scene.
+getScreenX(): double	Returns the *x*-coordinate of the mouse point in the screen.
+getScreenY(): double	Returns the *y*-coordinate of the mouse point in the screen.
+isAltDown(): boolean	Returns true if the `Alt` key is pressed on this event.
+isControlDown(): boolean	Returns true if the `Control` key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse `Meta` button is pressed on this event.
+isShiftDown(): boolean	Returns true if the `Shift` key is pressed on this event.

KeyEvent Class

javafx.scene.input.KeyEvent	
+getCharacter(): String	Returns the character associated with the key in this event.
+getCode(): KeyCode	Returns the key code associated with the key in this event.
+getText(): String	Returns a string describing the key code.
+isAltDown(): boolean	Returns true if the `Alt` key is pressed on this event.
+isControlDown(): boolean	Returns true if the `Control` key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse `Meta` button is pressed on this event.
+isShiftDown(): boolean	Returns true if the `Shift` key is pressed on this event.

Event Handling

- Event Handling is the mechanism that controls the event and decides what should happen, if an event occurs. This mechanism has the code which is known as an event handler that is executed when an event occurs.
- JavaFX provides handlers and filters to handle events. In JavaFX every event has
  - *Target* – The node on which an event occurred. A target can be a window, scene, and a node.
  - *Source* – The source from which the event is generated will be the source of the event.
  - *Type* – Type of the occurred event; in case of mouse event – mouse pressed, mouse released are the type of events.
- Phases of Event Handling
  - Target selection
  - Route Construction
  - Event Capturing Phase
  - Event Bubbling Phase

Event Handling Phases

- Target selection
  - When an action occurs, the system determines which node is the target based on internal rules:
  - Key events - the target is the node that has focus.
  - Mouse events - the target is the node at the location of the cursor.
  - Gesture events - the target is the node at the center point of all touches at the beginning of the gesture.
  - Swipe events - the target is the node at the center of the entire path of all of the fingers.
  - Touch events - the target for each touch point is the node at the location first pressed.
- Route Construction
  - Whenever an event is generated, the default/initial route of the event is determined by construction of an *Event Dispatch chain*. It is the path from the stage to the source node.
- Event Capturing Phase
  - After the construction of the event dispatch chain, the root node of the application dispatches the event.
  - This event travels to all nodes in the dispatch chain (from top to bottom).
  - If any of these nodes has a filter registered for the generated event, it will be executed.
  - If none of the nodes in the dispatch chain has a filter for the event generated, then it is passed to the target node and finally the target node processes the event.

Event Handling Phases (Cont.)

- Event Bubbling Phase
  - In the event bubbling phase, the event is travelled from the target node to the stage node (bottom to top).
  - If any of the nodes in the event dispatch chain has a handler registered for the generated event, it will be executed.
  - If none of these nodes have handlers to handle the event, then the event reaches the root node and finally the process will be completed.

Event Handlers and Filters

- Event filters and handlers are those which contains application logic to process an event.
- A node can register to more than one handler/filter. In case of parent–child nodes, you can provide a common filter/handler to the parents, which is processed as default for all the child nodes.
- During the event capturing phase, a filter is executed and during the event bubbling phase, a handler is executed.
- All the handlers and filters implement the interface `EventHandler` of the package `javafx.event`.

Handling Mouse Event

```
public class JavafxSample extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
    Group root = new Group();
    Scene scene =
        new Scene(root, 300, 250);
    scene.setOnMouseClicked(mouseHandler);
        scene.setOnMouseDragged(mouseHandler);
        scene.setOnMouseEntered(mouseHandler);
        scene.setOnMouseExited(mouseHandler);
        scene.setOnMouseMoved(mouseHandler);
        scene.setOnMousePressed(mouseHandler);
        scene.setOnMouseReleased(mouseHandler);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

```
    EventHandler<MouseEvent> mouseHandler =
    new EventHandler<MouseEvent>(){
        @Override
      public void handle(MouseEvent mouseEvent){
        System.out.println(
       mouseEvent.getEventType() + "\n" +
       "X : Y - "
       + mouseEvent.getX() + " : " +
mouseEvent.getY() + "\n" +
       "SceneX : SceneY - "
       + mouseEvent.getSceneX()+" :
"+mouseEvent.getSceneY() +
       "\n" + "ScreenX : ScreenY - "
       + mouseEvent.getScreenX()+" :
"+mouseEvent.getScreenY());
     }
    };
}
```

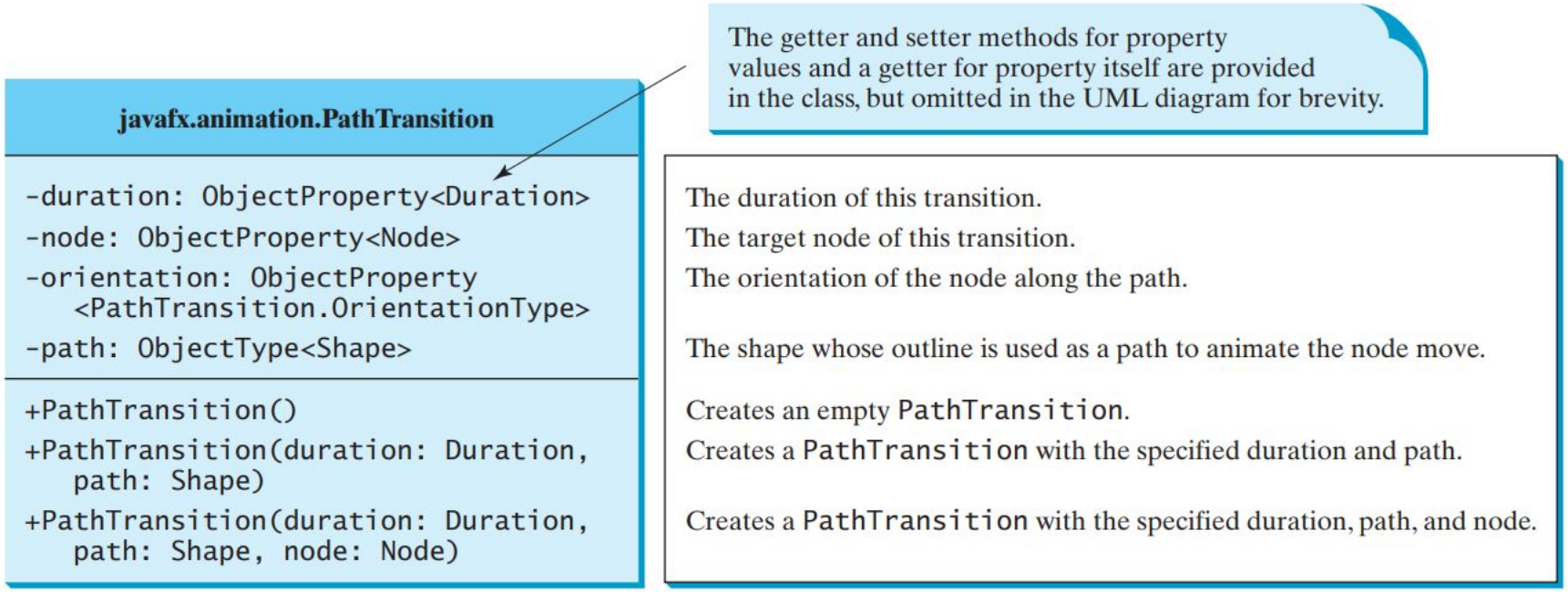
Creating a Calculator using JavaFX

Animation

- The **abstract Animation** class provides the core functionalities for animations in JavaFX
- Many **concrete subclasses of Animation** are provided in JavaFX.
- Here look up
  - **PathTransition**,
  - **FadeTransition** and
  - **Timeline**.

Path Transition

- The **PathTransition** class animates the **moves** of a **node along a path** from one end to the other **over a given time**.
- PathTransition is a subtype of **Animation**



Path Transition

- The **Duration** class defines a duration of time. It is an **immutable** class.
- The class defines constants **INDEFINTE**, **ONE**, **UNKNOWN**, and **ZERO** to represent an indefinte duration, 1 milliseconds, unknow, and 0 duration.
- You can use `new Duration(double millis)` to create an instance of Duration, the **add, subtract, multiply,** and **divide** **methods** to perform arithmetic operations, and the **toHours(), toMinutes(), toSeconds(),** and **toMillis()** to return the number of hours, minutes, seconds, and milliseconds in this duration. You can also use **compareTo** to compare two durations.
- The constants **NONE** and **ORTHOGONAL_TO_TANGENT** are defined in **PathTransition .OrientationType**. The latter specifies that the node is kept perpendicular to the path's tangent along the geometric path.

Example : Path Transition

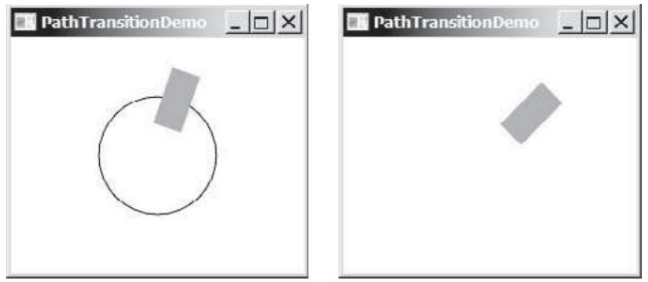
```
26
27      // Add circle and rectangle to the pane
28      pane.getChildren().add(circle);
29      pane.getChildren().add(rectangle);
30
31      // Create a path transition
32      PathTransition pt = new PathTransition();
33      pt.setDuration(Duration.millis(4000));
34      pt.setPath(circle);
35      pt.setNode(rectangle);
36      pt.setOrientation(
37        PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
38      pt.setCycleCount(Timeline.INDEFINITE);
39      pt.setAutoReverse(true);
40      pt.play(); // Start animation
41
42      circle.setOnMousePressed(e -> pt.pause());
43      circle.setOnMouseReleased(e -> pt.play());
44
45      // Create a scene and place it in the stage
46      Scene scene = new Scene(pane, 250, 200);
47      primaryStage.setTitle("PathTransitionDemo"); // Set the stage title
```

add circle to pane
add rectangle to pane

create a PathTransition
set transition duration
set path in transition
set node in transition
set orientation

set cycle count indefinite
set auto reverse true
play animation

pause animation
resume animation

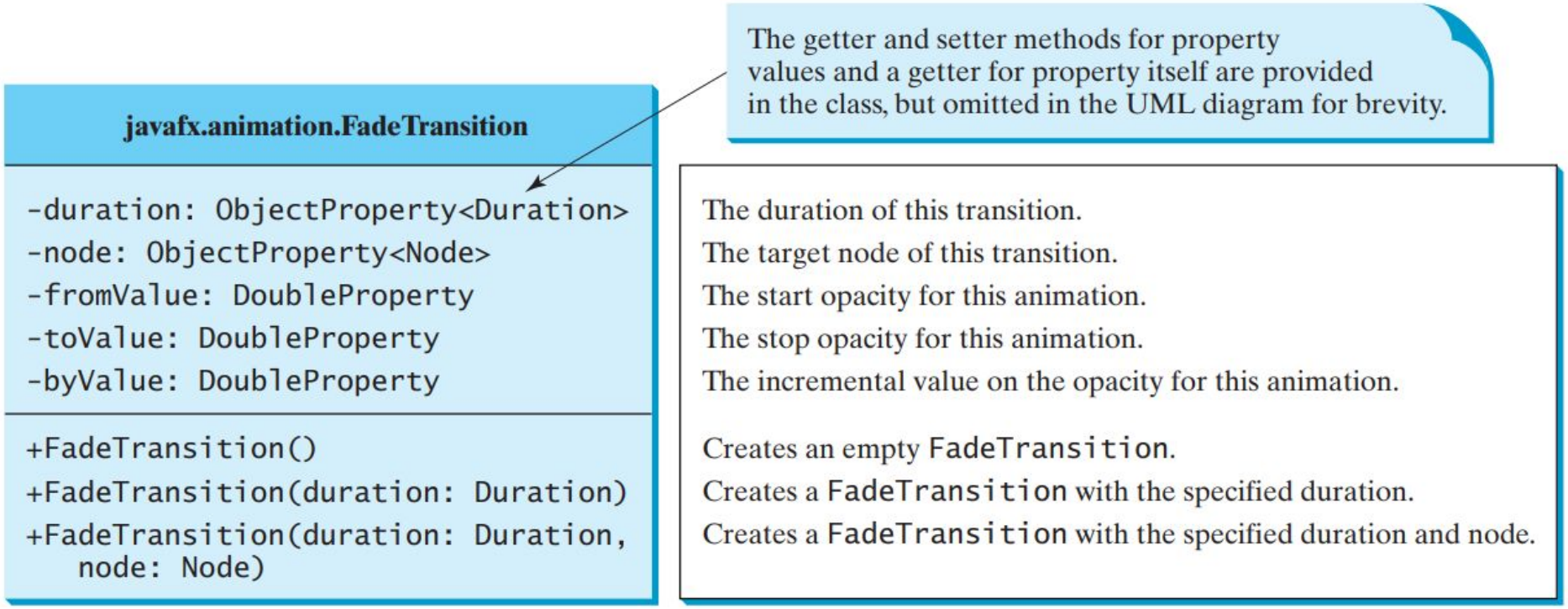


(a) (b)

- The program **creates** a **path transition (line 32)**, sets its duration to 4 seconds for one cycle of animation (line 33),
- **sets circle as the path (line 34)**, **sets rectangle as the node (line 35)**, and
- sets the orientation to orthogonal to tangent **(line 36)**.
- The cycle count is set to indefinite (line 38) so the animation continues forever.
- The auto reverse is set to true (line 39) so that the direction of the move is reversed in the alternating cycle.
- The program starts animation by invoking the **play()** method **(line 40)**. If the **pause()** method is replaced by the **stop()** method in line 42, the animation will start over from the beginning when it restarts.

Fade Transition

- The **FadeTransition** class animates the **change** of a **opacity in a node over a given time**.
- FadeTransition is a subtype of **Animation**



Example : Fade Transition

```
15      Pane pane = new Pane();                                          create a pane
16      Ellipse ellipse = new Ellipse(10, 10, 100, 50);                  create an ellipse
17      ellipse.setFill(Color.RED);                                      set ellipse fill color
18      ellipse.setStroke(Color.BLACK);                                  set ellipse stroke color
19      ellipse.centerXProperty().bind(pane.widthProperty().divide(2));  bind ellipse properties
20      ellipse.centerYProperty().bind(pane.heightProperty().divide(2));
21      ellipse.radiusXProperty().bind(
22        pane.widthProperty().multiply(0.4));
23      ellipse.radiusYProperty().bind(
24        pane.heightProperty().multiply(0.4));
25      pane.getChildren().add(ellipse);                                 add ellipse to pane
26
27      // Apply a fade transition to ellipse
28      FadeTransition ft =                                              create a FadeTransition
29        new FadeTransition(Duration.millis(3000), ellipse);
30      ft.setFromValue(1.0);                                            set start opaque value
31      ft.setToValue(0.1);                                              set end opaque value
32      ft.setCycleCount(Timeline.INDEFINITE);                           set cycle count
33      ft.setAutoReverse(true);                                         set auto reverse true
34      ft.play(); // Start animation                                    play animation
35
36      // Control animation
37      ellipse.setOnMousePressed(e -> ft.pause());                      pause animation
38      ellipse.setOnMouseReleased(e -> ft.play());                      resume animation
```

- A **fade transition** is created with a duration of 3 seconds for the ellipse **(line 29)**.
- It sets the **start opaque to 1.0 (line 30)** and the **stop opaque 0.1 (line 31)**.
- The cycle count is set to infinite so the animation is repeated indefinitely **(line 32)**.
- When the mouse is pressed, the animation is paused **(line 37)**.
- When the mouse is released, the animation resumes from where it was paused **(line 38)**.

TimeLine

- **PathTransition** and **FadeTransition** define specialized animations.
- The **Timeline** class can be used to program any **animation using one or more KeyFrames**.
- Each KeyFrame is **executed sequentially at a specified time interval**.
- Timeline **inherits from Animation**.
- You can construct a **Timeline** using the constructor
  - `new Timeline(KeyFrame... keyframes)`
- A **KeyFrame** can be constructed using
  - `new KeyFrame(Duration duration, EventHandler onFinished)`
- The handler **onFinished** is called when the duration for the key frame is elapsed

Example : Fade Transition

```
17  StackPane pane = new StackPane();
18  Text text = new Text(20, 50, "Programming is fun");
19  text.setFill(Color.RED);
20  pane.getChildren().add(text); // Place text into the stack pane
21
22  // Create a handler for changing text
23  EventHandler<ActionEvent> eventHandler = e -> {
24    if (text.getText().length() != 0) {
25      text.setText("");
26    }
27    else {
28      text.setText("Programming is fun");
29    }
30  };
31
32  // Create an animation for alternating text
33  Timeline animation = new Timeline(
34    new KeyFrame(Duration.millis(500), eventHandler));
35  animation.setCycleCount(Timeline.INDEFINITE);
36  animation.play(); // Start animation
37
38  // Pause and resume animation
39  text.setOnMouseClicked(e -> {
40    if (animation.getStatus() == Animation.Status.PAUSED) {
41      animation.play();
42    }
43    else {
44      animation.pause();
45    }
46  });
```

create a stack pane
create a text
add text to pane
handler for changing text
set text empty
set text
create a Timeline
create a KeyFrame for handler
set cycle count indefinite
play animation
resume animation
pause animation

- A **handler is created** to change the text to empty **(lines 24–26)** if it is not empty or to Progrmming is fun if it is empty (lines 27–29).
- A **KeyFrame** is created to run an action event in every half second **(line 34)**. A Timeline animation is created to contain a key frame **(lines 33 and 34)**. The animation is set to run indefinitely (line 35).
- The mouse clicked event is set for the text **(lines 39–46)**. A mouse click on the text resumes the animation if the animation is paused **(lines 40–42)**, and a mouse click on the text pauses the animation if the animation is running **(lines 43–45)**

Animation

- JavaFX provides easy to use animation API (javafx.animation package).
- There are some predefined animation that can be used out of the box or you can implement custom animations using KeyFrames.
- Following are the main predefined animations in JavaFX.
  - **TranslateTransition**: Translate transition allows to create movement animation from one point to another within a duration. Using TranslateTransition#**setByX** / TranslateTransition#**setByY**, you can set how much it should move in x and y axis respectively. It also possible to set precise destination by using TranslateTransition#**setToX** / TranslateTransition#**setToY**.
  - **ScaleTransition**: Scale transition is another JavaFX animation which can be used out of the box that allows to animate the scale / zoom of the given object. The object can be enlarged or minimized using this animation.
  - **RotateTransition**: Rotate transition provides animation for rotating an object. We can provide upto what angle the node should rotate by **toAngle.** Using **byAngle** we can specify how much it should rotate from current angle of rotation.
  - **FadeTransition**: Fade transition creates a fade in / fade out effect by controlling opacity of the object. We can make fade in transition or fade out transition in JavaFX by setting the to and from value.
  - **PathTransition**: Path transition provides option to move object through a specified path. The path can be anything from simple straight line to complex quadratic curves.

Example Animation

```
@Override
public void start(Stage primaryStage) throws Exception {
    // TODO Auto-generated method stub
    Rectangle rect = new Rectangle(200,400);
    rect.setX(400);
    rect.setY(150);

    Group root = new Group(rect);

    Scene s = new Scene(root, 1000,700);

    Duration dt = new Duration(2500);
    RotateTransition rt = new RotateTransition(dt,rect);
    rt.setByAngle(180);
    rt.play();

    primaryStage.setScene(s);
    primaryStage.show();

}
```

Exercise

- Creating a Calculator using JavaFX
- Write a program that displays a tic-tac-toe board. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are images in the files X.gif and O.gif.
- Write a program that displays the color of a circle as red when the mouse button is pressed and as white when the mouse button is released.

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-08

JavaFX UI Controls & Multimedia

JavaFx™

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

Jayesh.vagadiya@darshan.ac.in
9537133260

Outline

- ✔ Label
- ✔ Button
- ✔ CheckBox
- ✔ RadioButton
- ✔ TextField
- ✔ TextArea
- ✔ ComboBox
- ✔ ListView
- ✔ ScrollBar
- ✔ Slider
- ✔ Video
- ✔ Audio

Label

- Label is used to display a short text or an image, it is a non-editable text control.
- It is useful for displaying text that is required to fit within a specific space.
- Label can only display text or image and it cannot get focus.
- Constructor for the Label class are:

Constructor	Description
Label()	Creates an empty Label
Label(String text)	Creates Label with supplied text
Label(String text, Node graphics)	Creates a Label with the supplied text and graphic.

Label Example

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

public class LabelExample extends Application  {
    @Override
    public void start(Stage primaryStage) throws Exception {

        Label label = new Label("Welcome to JavaFX");

        Scene scene = new Scene(label, 200, 200);
    primaryStage.setTitle("JavaFX Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

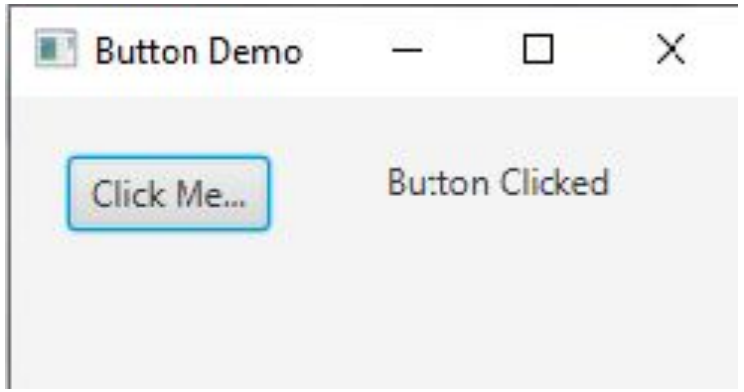
Button

- Button control enables an application to have some action executed when the application user clicks the button.
- The button control can contain text and/or a graphic.
- When a button is pressed and released a ActionEvent is sent. Some action can be performed based on this event by implementing an EventHandler to process the ActionEvent.
- Buttons can also respond to mouse events by implementing an EventHandler to process the MouseEvent.
- Constructor for the Button class are:

Constructor	Description
Button()	Creates a button with an empty string for its label.
Button(String text)	Creates a button with the specified text as its label.
Button(String text, Node graphic)	Creates a button with the specified text and icon for its label.

Button Example

```
@Override
public void start(Stage primaryStage) {
    Button btn = new Button();
    Label lbl = new Label();
    btn.setText("Click Me...");
    btn.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            lbl.setText("Button Clicked");
        }
    });
    HBox root = new HBox();
    root.setMargin(btn, new Insets(20,20,20,20));
    root.setMargin(lbl, new Insets(20,20,20,20));
    root.getChildren().add(btn);
    root.getChildren().add(lbl);
    primaryStage.setTitle("Button Demo");
    primaryStage.setScene(new Scene(root, 250, 100));
    primaryStage.show();
}
```



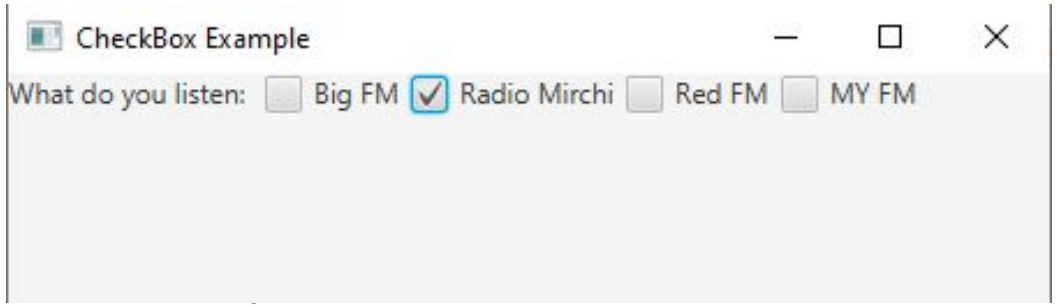
Checkbox

- The Check Box is used to provide more than one choices to the user.
- It can be used in a scenario where the user is prompted to select more than one option.
- It is different from the radiobutton in the sense that, we can select more than one checkboxes in a scenerio.
- Constructor for the Checkbox class are:

Constructor	Description
CheckBox()	Creates a check box with an empty string for its label.
CheckBox(String text)	Creates a check box with the specified text as its label.

Checkbox Example

```
    @Override
    public void start(Stage primaryStage) throws Exception {
        Label l = new Label("What do you listen: ");
        CheckBox c1 = new CheckBox("Big FM");
        CheckBox c2 = new CheckBox("Radio Mirchi");
        CheckBox c3 = new CheckBox("Red FM");
        CheckBox c4 = new CheckBox("MY FM");
        HBox root = new HBox();
        root.getChildren().addAll(l,c1,c2,c3,c4);
        root.setSpacing(5);
        Scene scene=new Scene(root,450,100);
        primaryStage.setScene(scene);
        primaryStage.setTitle("CheckBox Example");
        primaryStage.show();
    }
```



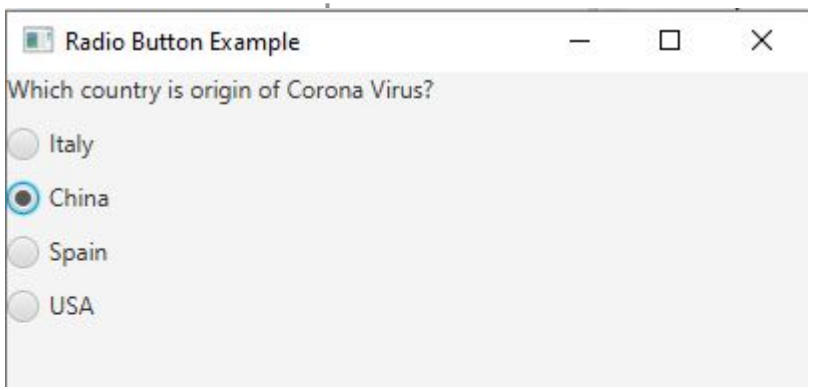
RadioButton

- The Radio Button is used to provide various options to the user.
- The user can only choose one option among all.
- A radio button is either selected or deselected.
- It can be used in a scenario of multiple choice questions in the quiz where only one option needs to be chosen by the student.
- Constructor for the RadioButton class are:

Constructor	Description
RadioButton()	Creates a radio button with an empty string for its label.
RadioButton(String text)	Creates a radio button with the specified text as its label.

Checkbox Example

```
public void start(Stage primaryStage) throws Exception {
    Label lbl = new Label("Which country is origin of Corona Virus?");
    ToggleGroup group = new ToggleGroup();
    RadioButton button1 = new RadioButton("Italy");
    RadioButton button2 = new RadioButton("China");
    RadioButton button3 = new RadioButton("Spain");
    RadioButton button4 = new RadioButton("USA");
    button1.setToggleGroup(group);
    button2.setToggleGroup(group);
    button3.setToggleGroup(group);
    button4.setToggleGroup(group);
    VBox root=new VBox();
    root.setSpacing(10);
    root.getChildren().addAll(lbl,button1,button2,button3,button4);
    Scene scene=new Scene(root,400,300);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Radio Button Example");
    primaryStage.show();
}
```

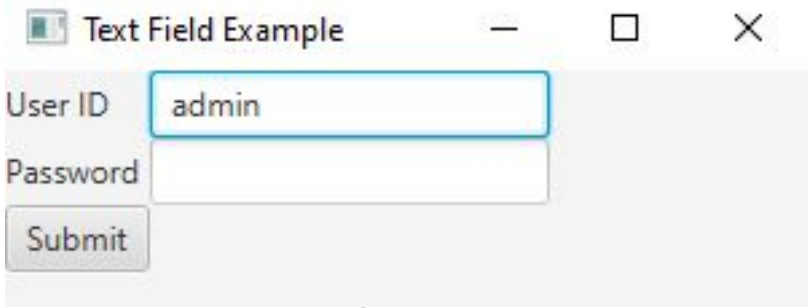


TextField

- Text input component that allows a user to enter a single line of unformatted text.
- Constructor for the TextField class are:

Constructor	Description
TextField()	Creates a TextField with empty text content.
TextField(String text)	Creates a TextField with initial text content.

```
Label user_id=new Label("User ID");
Label password = new Label("Password");
TextField tf1=new TextField();
TextField tf2=new TextField();
Button b = new Button("Submit");
GridPane root = new GridPane();
root.addRow(0, user_id, tf1);
root.addRow(1, password, tf2);
root.addRow(2, b);
Scene scene=new Scene(root,300,200);
primaryStage.setScene(scene);
primaryStage.setTitle("Text Field Example");
primaryStage.show();
```

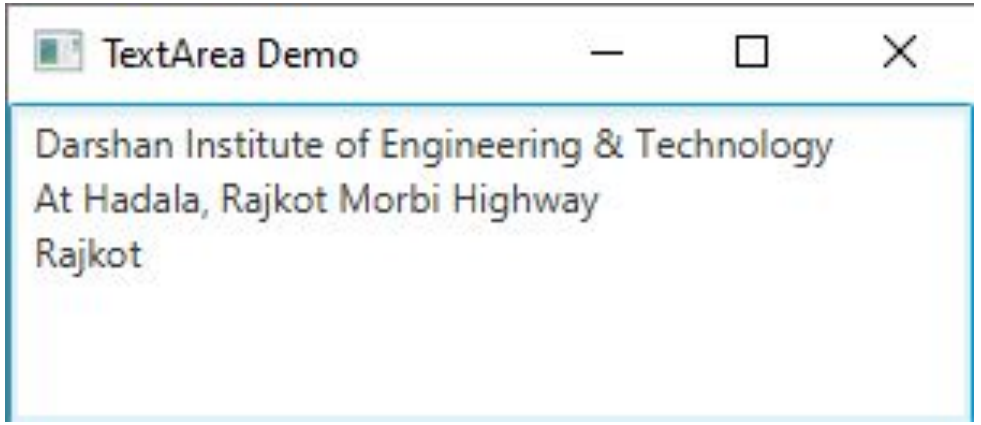


TextArea

- Text input component that allows a user to enter multiple lines of plain text.
- Constructor for the TextArea class are:

Constructor	Description
TextArea()	Creates a TextArea with empty text content.
TextArea(String text)	Creates a TextArea with initial text content.

```
@Override
public void start(Stage primaryStage) throws Exception {
TextArea textArea = new TextArea();
VBox vbox = new VBox(textArea);
Scene scene = new Scene(vbox, 300, 100);
primaryStage.setTitle("TextArea Demo");
primaryStage.setScene(scene);
primaryStage.show();
}
```



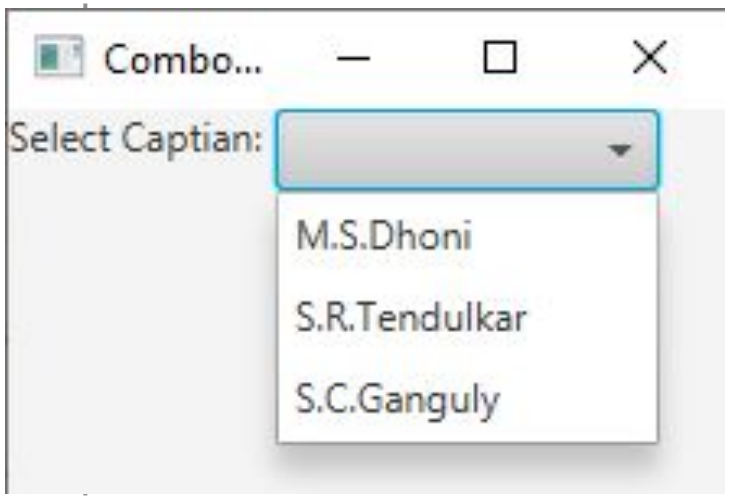
ComboBox

- A combo box is a typical element of a user interface that enables users to choose one of several options.
- A combo box is helpful when the number of items to show exceeds some limit, because it can add scrolling to the drop down list.
- Constructor for the ComboBox class are:

Constructor	Description
ComboBox()	Creates a default ComboBox instance with an empty items list and default selection model.
ComboBox(ObservableList<T> items)	ComboBox(ObservableList<T> items) Creates a default ComboBox instance with the provided items list and a default selection model.

ComboBox Example

```
    @Override
    public void start(Stage primaryStage) throws Exception {
        Label lbl = new Label("Select Captain: ");
       ObservableList<String> options =
FXCollections.observableArrayList(
                "M.S.Dhoni",
                "S.R.Tendulkar",
                "S.C.Ganguly"
            );
        ComboBox<String> comboBox = new ComboBox<String>(options);
        HBox hbox = new HBox();
        hbox.getChildren().addAll(lbl,comboBox);
        Scene scene = new Scene(hbox, 200, 120);
        primaryStage.setTitle("ComboBox Experiment 1");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```



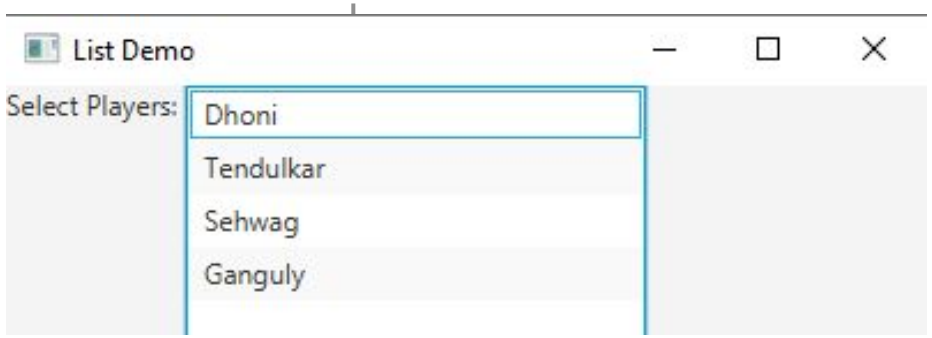
ListView

- A ListView displays a horizontal or vertical list of items from which the user may select, or with which the user may interact.
- Constructor for the ListView class are:

Constructor	Description
ListView()	Creates a default ListView which will display contents stacked vertically.
ListView(ObservableList<T> items)	Creates a default ListView which will stack the contents retrieved from the provided ObservableList vertically.

ListView Example

```
@Override
public void start(Stage primaryStage) throws Exception {
    Label lbl = new Label("Select Players: ");
    ObservableList<String> options =
            FXCollections.observableArrayList(
                "Dhoni","Tendulkar","Sehwag","Ganguly"
            );
    ListView<String> list = new ListView<String>(options);
    list.setPrefSize(200, 50);
    HBox hbox = new HBox();
    hbox.getChildren().addAll(lbl,list);
    Scene scene = new Scene(hbox, 400, 300);
    primaryStage.setTitle("List Demo");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```



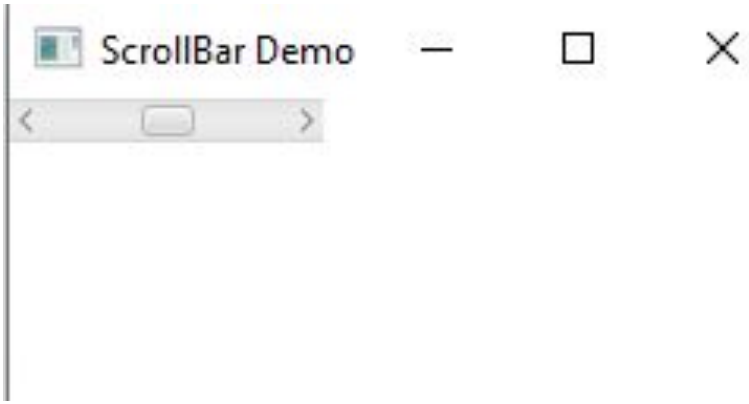
ScrollBar

- JavaFX Scroll Bar is used to provide a scroll bar to the user so that the user can scroll down the application pages.
- Either a horizontal or vertical bar with increment and decrement buttons and a "thumb" with which the user can interact. Typically not used alone but used for building up more complicated controls such as the ScrollPane and ListView.
- Constructor for the ScrollBar class are:

Constructor	Description
ScrollBar()	Creates a new horizontal ScrollBar.

ScrollBar Example

```
    @Override
    public void start(Stage stage) {
        ScrollBar sc = new ScrollBar();
        sc.setMin(0);
        sc.setMax(100);
        sc.setValue(50);
        Group root = new Group();
        Scene scene = new Scene(root, 250, 100);
        stage.setScene(scene);
        root.getChildren().add(sc);
        stage.setTitle("ScrollBar Demo");
        stage.show();
    }
```



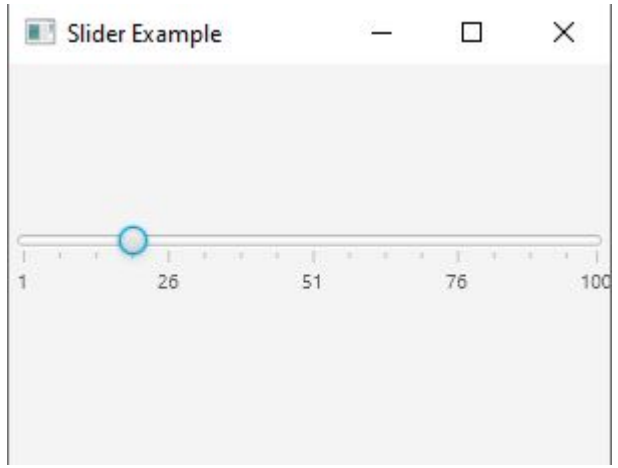
Slider

- The Slider Control is used to display a continuous or discrete range of valid numeric choices and allows the user to interact with the control.
- It is typically represented visually as having a "track" and a "knob" or "thumb" which is dragged within the track. The Slider can optionally show tick marks and labels indicating the different slider position values.
- The three fundamental variables of the slider are min, max, and value. The value should always be a number within the range defined by min and max. min should always be less than or equal to max. min defaults to 0, whereas max defaults to 100.
- Constructor for the Slider class are:

Constructor	Description
Slider()	Creates a default Slider instance.
Slider(double min, double max, double value)	Constructs a Slider control with the specified slider min, max and current value values.

Slider Example

```
@Override
public void start(Stage primaryStage) throws Exception {
    Slider slider = new Slider(1,100,20);
    slider.setShowTickLabels(true);
    slider.setShowTickMarks(true);
    StackPane root = new StackPane();
    root.getChildren().add(slider);
    Scene scene = new Scene(root,300,200);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Slider Example");
    primaryStage.show();
}
```



Video

- In the case of playing video, we need to use the MediaView node to display the video onto the scene.
- For this purpose, we need to instantiate the MediaView class by passing the Mediaplayer object into its constructor. Due to the fact that, MediaView is a JavaFX node, we will be able to apply effects to it.

```
    public void start(Stage primaryStage) throws Exception {
        String path = "G:\\demo.mp4";
        Media media = new Media(new File(path).toURI().toString());
        MediaPlayer mediaPlayer = new MediaPlayer(media);
        MediaView mediaView = new MediaView(mediaPlayer);
        mediaPlayer.setAutoPlay(true);
        Group root = new Group();
        root.getChildren().add(mediaView);
        Scene scene = new Scene(root,1366,768);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Playing video");
        primaryStage.show();
    }
```

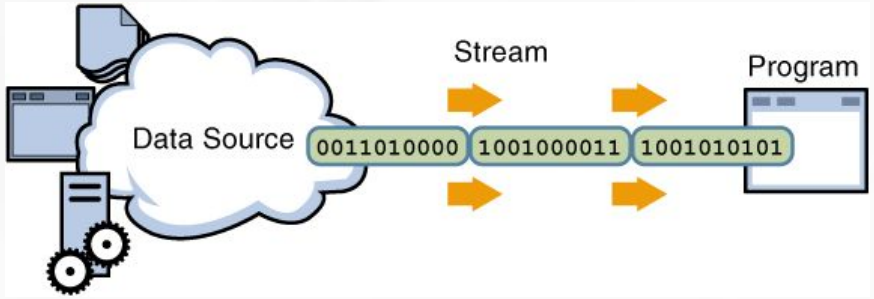
Audio

- We can load the audio files with extensions like .mp3,.wav and .aifff by using JavaFX Media API. We can also play the audio in HTTP live streaming format.
- Instantiate javafx.scene.media.Media class by passing the audio file path in its constructor to play the audio files.

```
    public void start (Stage primaryStage) throws Exception {
        String path = "G://demo.mp3";
        Media media = new Media(new File(path).toURI().toString());
        MediaPlayer mediaPlayer = new MediaPlayer(media);
        mediaPlayer.setAutoPlay(true);
        primaryStage.setTitle("Playing Audio");
        primaryStage.show();
    }
```

Unit-09

IO Programming



Outline

- ✔ File class
- ✔ Stream
- ✔ Byte Stream
- ✔ Character Stream

File class

- Java `File` class represents the files and directory pathnames in an **abstract manner**. This class is used for **creation of files** and **directories**, **file searching**, **file deletion** etc.
- The File object represents the actual file/directory on the disk. Below given is the list of constructors to create a File object.
- Constructors :

Sr.	Constructor
1	File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
2	File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.
3	File(URI uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Methods of File Class

Sr.	Method
1	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
2	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
3	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
4	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
5	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

Methods of File Class (Cont.)

Sr.	Method
6	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
7	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria
8	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
9	public long length() Returns the length of the file denoted by this abstract pathname.
10	public boolean delete() Deletes the file or directory.
11	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

File Class Example

```
import java.io.File;
class FileDemo {
    public static void main(String args[]) {
        File f1 = new File("FileDemo.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println (f1.canRead () ? "is readable" : "is not readable");
        System.out.println ("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }
}
```

Stream

- A stream can be defined as a sequence of data.
- All streams represent an input source and an output destination.
- There are two kinds of Streams
  - **Byte Stream**
  - **Character Stream**
- The **java.io** package contains all the classes required for input-output operations.
- The stream in the java.io package supports all the datatype including primitive.

Byte Streams

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.

FileOutputStream

- Java `FileOutputStream` is an output stream for writing data to a file.
- `FileOutputStream` will create the file before opening it for output.
- On opening a read only file, it will throw an exception.

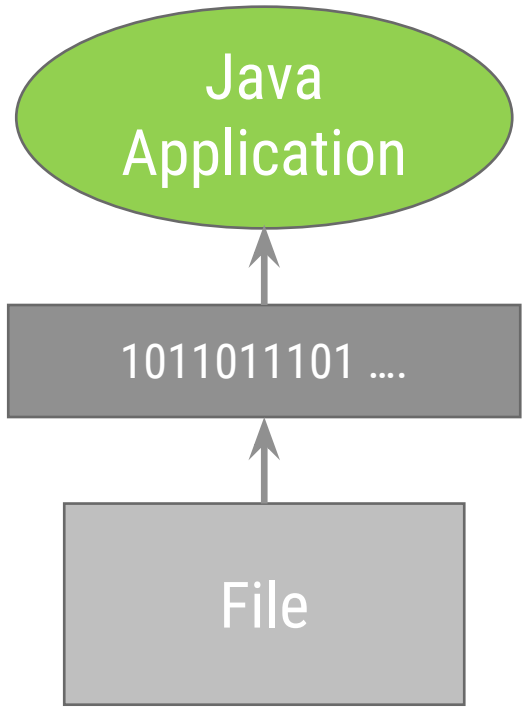
Sr.	Method
1	**void write(byte[] b)** This method writes b.length bytes from the specified byte array to this file output stream.
2	**void write(byte[] b, int off, int len)** This method writes len bytes from the specified byte array starting at offset off to this file output stream.
3	**void write(int b)** This method writes the specified byte to this file output stream.
4	**void close()** This method closes this file output stream and releases any system resources associated with this stream.

FileOutputStream Example

```
class FileOutDemo {
    public static void main(String args[]) {
        try {
            FileOutputStream fout = new FileOutputStream("abc.txt");
            String s = "Sourav Ganguly is my favorite player";
            byte b[] = s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Success...");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

FileInputStream

- `FileInputStream` class is used to read bytes from a file.
- It should be used to read byte-oriented data for example to read image, audio, video etc.



Sr.	Method
1	**public int read()**
2	**public int read(byte[] b)** b - the buffer into which the data is read. Returns: the total number of bytes read into the buffer, or -1.
3	**public int read(byte[] b, int off, int len)** b - the buffer into which the data is read. off - the start offset in the destination array b len - the maximum number of bytes read. Returns: the total number of bytes read into the buffer, or -1
4	**public long skip(long n)** n - the number of bytes to be skipped. Returns: the actual number of bytes skipped.
5	**public int available()** an estimate of the number of remaining bytes that can be read
6	**public void close()** Closes this file input stream and releases any system resources associated.

FileInputStream Example

```
class SimpleRead {
    public static void main(String args[]) {
        try {
            FileInputStream fin = new FileInputStream("abc.txt");
            int i = 0;
            while ((i = fin.read()) != -1) {
                System.out.println((char) i);
            }
            fin.close();
        } catch (Exception e) {
             System.out.println(e);
        }
    }
}
```

Example of Byte Streams

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```

Character Streams

- Character Streams provide a convenient means for handling input and output of characters.
- Internationalization is possible as it uses Unicode.
- For character streams we have two base classes
  - Reader
  - Writer

Reader

- The Java `Reader` class is the base class of all Reader's in the IO API.
- Subclasses include a `FileReader`, `BufferedReader`, `InputStreamReader`, `StringReader` and several others.
- Here is a simple Java IO Reader example:

```
Reader reader = new FileReader("c:\\data\\myfile.txt");
int data = reader.read();
while (data != -1) {
    char dataChar = (char) data;
    data = reader.read();
}
```

- Combining Readers with InputStream

```
Reader reader = new InputStreamReader("c:\\data\\myfile.txt");
```

Writer

- The Java `Writer` class is the base class of all Writers in the I-O API.
- Subclasses include `BufferedWriter`, `PrintWriter`, `StringWriter` and several others.
- Here is a simple Java IO Writer example:

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");
writer.write("Hello World Writer");
writer.close();
```

- Combining Readers With OutputStreams

```
Writer writer = new OutputStreamWriter("c:\\data\\file-output.txt");
```

BufferedReader

- The `java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Following are the important points about `BufferedReader`:
  - The buffer size may be specified, or the default size may be used.
  - Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.
- Constructors :

Sr.	Constructor
1	BufferedReader(Reader in) This creates a buffering character-input stream that uses a default-sized input buffer.
2	BufferedReader(Reader in, int sz) This creates a buffering character-input stream that uses an input buffer of the specified size.

BufferedReader (Methods)

Sr.	Methods
1	void close() This method closes the stream and releases any system resources associated with it.
2	int read() This method reads a single character.
3	int read(char[] cbuf, int off, int len) This method reads characters into a portion of an array.
4	String readLine() This method reads a line of text.
5	void reset() This method resets the stream.
6	long skip(long n) This method skips characters.

BufferedReader – Example

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("input.txt");
        BufferedReader br = new BufferedReader(fr);
        char c[] = new char[20];
        br.skip(8);
        if (br.ready()) {
            System.out.println(br.readLine());
            br.read(c);
            for (int i = 0; i < 20; i++) {
                System.out.print(c[i]);
            }
        }
    }
}
```

Unit-10 &11

Collection Framework

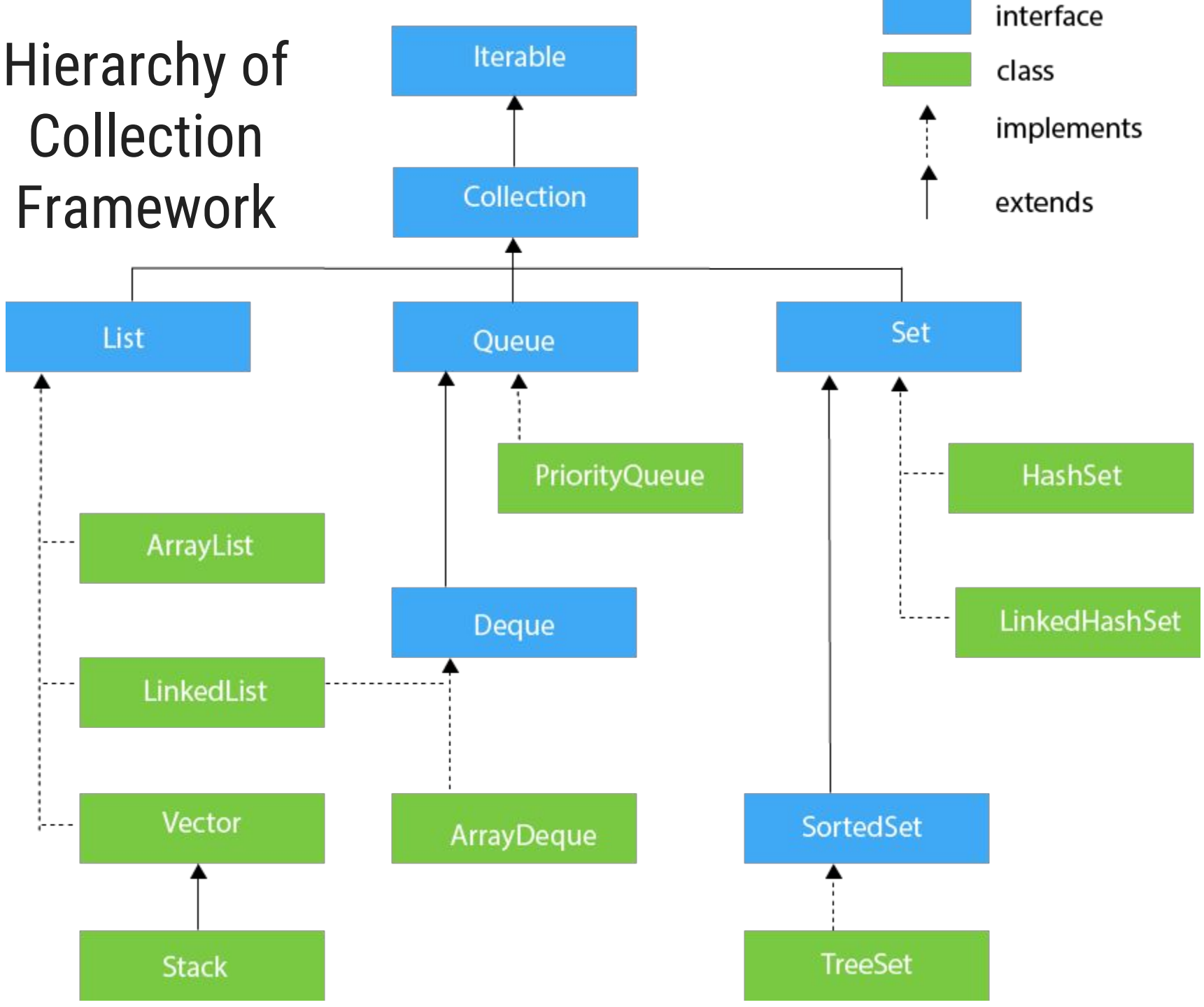
Outline

- ✔ Collection
- ✔ List Interface
- ✔ Iterator
- ✔ Comparator
- ✔ Vector Class
- ✔ Stack
- ✔ Queue
- ✔ List v/s Sets
- ✔ Map Interfaces

Collection

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework



Collection Interface - Methods

Sr.	Method & Description
1	boolean add(E e) It is used to insert an element in this collection.
2	boolean addAll(Collection<? extends E> c) It is used to insert the specified collection elements in the invoking collection.
3	void clear() It removes the total number of elements from the collection.
4	booelan contains(Object element) It is used to search an element.
5	boolean containsAll(Collection<?> c) It is used to search the specified collection in the collection.
6	boolean equals(Object obj) Returns true if invoking collection and obj are equal. Otherwise returns false.
7	int hashCode() Returns the hashcode for the invoking collection.

Collection Interface – Methods (Cont.)

Sr.	Method & Description
8	boolean isEmpty() Returns true if the invoking collection is empty. Otherwise returns false.
9	Iterator iterator() It returns an iterator.
10	boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	boolean removeAll(Collection<?> c) It is used to delete all the elements of the specified collection from the invoking collection.
12	boolean retainAll(Collection<?> c) It is used to delete all the elements of invoking collection except the specified collection.
13	int size() It returns the total number of elements in the collection.

List Interface

- The **`List`** interface extends **`Collection`** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- List is a generic interface with following declaration

```
interface List<E>
```

where `E` specifies the type of object.

List Interface - Methods

Sr.	Method & Description
1	void add(int index, Object obj) Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	Object get(int index) Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
5	int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

List Interface – Methods (Cont.)

Sr.	Method & Description
6	ListIterator listIterator( ) Returns an iterator to the start of the invoking list.
7	ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index.
8	Object remove(int index) Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list.
10	List subList(int start, int end) Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

List Interface (example)

```java
import java.util.*;
public class CollectionsDemo {
   public static void main(String[] args) {
      List a1 = new ArrayList();
      a1.add("Sachin");
      a1.add("Sourav");
      a1.add("Shami");
      System.out.println("ArrayList Elements");
      System.out.print("\t" + a1);

      List l1 = new LinkedList();
      l1.add("Mumbai");
      l1.add("Kolkata");
      l1.add("Vadodara");
      System.out.println();
      System.out.println("LinkedList Elements");
      System.out.print("\t" + l1);
   }
}
```

Here **ArrayList** & **LinkedList** implements **List** Interface

```
ArrayList Elements
        [Sachin, Sourav, Shami]
LinkedList Elements
        [Mumbai, Kolkata, Vadodara]
```

Iterator

- `Iterator` interface is used to cycle through elements in a collection, eg. displaying elements.
- `ListIterator` extends `Iterator` to allow bidirectional traversal of a list, and the modification of elements.
- Each of the collection classes provides an `iterator( )` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection, follow these steps:
  1. Obtain an iterator to the start of the collection by calling the collection's `iterator( )` method.
  2. Set up a loop that makes a call to `hasNext( ).` Have the loop iterate as long as `hasNext( )` returns true.
  3. Within the loop, obtain each element by calling `next( ).`

Iterator - Methods

Sr.	Method & Description
1	boolean hasNext() Returns true if there are more elements. Otherwise, returns false.
2	E next() Returns the next element. Throws NoSuchElementException if there is not a next element.
3	void remove() Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Iterator - Example

```
import java.util.*;
public class IteratorDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        System.out.print("Contents of list: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
    }
}
```

```
Contents of list: C A E B D F
```

Comparator

- Comparator interface is used to set the sort order of the object to store in the sets and lists.
- The Comparator interface defines two methods: compare( ) and equals( ).
- `int compare(Object obj1, Object obj2)`

  `obj1` and `obj2` are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if `obj1` is greater than `obj2`. Otherwise, a negative value is returned.

- `boolean equals(Object obj)`

  `obj` is the object to be tested for equality. The method returns true if `obj` and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Comparator Example

```
public class ComparatorDemo {
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student("Vijay",23));
        al.add(new Student("Ajay",27));
        al.add(new Student("Jai",21));
        System.out.println("Sorting by age");
        Collections.sort(al,new AgeComparator());
        Iterator<Student> itr2=al.iterator();
        while(itr2.hasNext()){
            Student st=(Student)itr2.next();
            System.out.println(st.name+" "+st.age);
        }
    }
}
```

```
class AgeComparator implements
Comparator<Object>{
    public int compare(Object o1,Object o2){
        Student s1=(Student)o1;
        Student s2=(Student)o2;
        if(s1.age==s2.age) return 0;
        else if(s1.age>s2.age) return 1;
        else return -1;
    }
}
```

```
import java.util.*;
class Student {
    String name;
    int age;
    Student(String name, int
    age){
        this.name = name;
        this.age = age;
    }
}
```

Vector Class

- **`Vector`** implements a dynamic array.
- It is similar to **`ArrayList`**, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collection framework
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.
- Vector is declared as follows:

```
Vector<E> = new Vector<E>;
```

Vector - Constructors

Sr.	Constructor & Description
1	Vector( ) This constructor creates a default vector, which has an initial size of 10
2	Vector(int size) This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	Vector(int size, int incr) This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	Vector(Collection c) creates a vector that contains the elements of collection c

Vector - Methods

Sr.	Method & Description
7	boolean containsAll(Collection c) Returns true if this Vector contains all of the elements in the specified Collection.
8	Enumeration elements() Returns an enumeration of the components of this vector.
9	Object firstElement() Returns the first component (the item at index 0) of this vector.
10	Object get(int index) Returns the element at the specified position in this Vector.
11	int indexOf(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
12	boolean isEmpty() Tests if this vector has no components.

Vector – Method (Cont.)

Sr.	Method & Description
13	Object lastElement() Returns the last component of the vector.
14	int lastIndexOf(Object elem) Returns the index of the last occurrence of the specified object in this vector.
15	Object remove(int index) Removes the element at the specified position in this Vector.
16	boolean removeAll(Collection c) Removes from this Vector all of its elements that are contained in the specified Collection.
17	Object set(int index, Object element) Replaces the element at the specified position in this Vector with the specified element.
18	int size() Returns the number of components in this vector.

Stack

- `Stack` is a subclass of `Vector` that implements a standard last-in, first-out stack.
- `Stack` only defines the default constructor, which creates an empty stack.
- `Stack` includes all the methods defined by `Vector` and adds several of its own.
- `Stack` is declared as follows:

```
Stack<E> st = new Stack<E>();
```

where `E` specifies the type of object.

Stack - Methods

- Stack includes all the methods defined by Vector and adds several methods of its own.

Sr.	Method & Description
1	boolean empty() Returns true if the stack is empty, and returns false if the stack contains elements.
2	E peek() Returns the element on the top of the stack, but does not remove it.
3	E pop() Returns the element on the top of the stack, removing it in the process.
4	E push(E element) Pushes element onto the stack. Element is also returned.
5	int search(Object element) Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

Queue

- `Queue` interface extends `Collection` and declares the behaviour of a queue, which is often a first-in, first-out list.
- `LinkedList` and `PriorityQueue` are the two classes which implements Queue interface
- `Queue` is declared as follows:

```
Queue<E> q = new LinkedList<E>();
Queue<E> q = new PriorityQueue<E>();
```

where `E` specifies the type of object.

Queue - Methods

Sr.	Method & Description
1	E element() Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
2	boolean offer(E obj) Attempts to add obj to the queue. Returns true if obj was added and false otherwise.
3	E peek() Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
4	E poll() Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
5	E remove() Returns the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Queue Example

```
import java.util.*;
public class QueueDemo {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<String>();
        q.add("Tom");
        q.add("Jerry");
        q.add("Mike");
        q.add("Steve");
        q.add("Harry");
        System.out.prin
        System.out.prin
        System.out.prin
        System.out.println("poll(): "+q.poll());
        System.out.println("peek(): "+q.peek());
        System.out.println("Elements in Queue:"+q);
    }
}
```

```
Elements in Queue:[Tom, Jerry, Mike, Steve, Harry]
Removed element: Tom
Head: Jerry
poll(): Jerry
peek(): Mike
Elements in Queue:[Mike, Steve, Harry]
```

PriorityQueue

- `PriorityQueue` extends `AbstractQueue` and implements the `Queue` interface.
- It creates a queue that is prioritized based on the queue's comparator.
- `PriorityQueue` is declared as follows:

```
PriorityQueue<E> = new PriorityQueue<E>;
```

- It builds an empty queue with starting capacity as 11.

PriorityQueue - Example

```
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new
PriorityQueue<>();
        numbers.add(750);
        numbers.add(500);
        numbers.add(900);
        numbers.add(100);
        while (!numbers.isEmpty()) {
            System.out.println(numbers.remove());
        }
    }
}
```

```
100
500
750
900
```

List v/s Sets

List	Set
Lists allow duplicates.	Sets allow only unique elements.
List is an ordered collection.	Sets is an unordered collection.
Popular implementation of List interface includes ArrayList, Vector and LinkedList.	Popular implementation of Set interface includes HashSet, TreeSet and LinkedHashSet.

When to use List and Set?

Lists - If insertion order is maintained during insertion and allows duplicates.

Sets – If unique collection without any duplicates without maintaining order.

Maps

- A map is an object that stores associations between keys and values, or key/value pairs.
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.
- Maps don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can’t obtain an iterator to a map.

Map Interfaces

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends Map so that the keys are maintained in ascending order.

Map Classes

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterators.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface.
- It uses a hash table to store the map. This allows the execution time of get() and put() to remain constant even for large sets.
- HashMap is a generic class that has declaration:

```
class HashMap<K,V>
```

HashMap - Constructors

Sr.	Constructor & Description
1	HashMap() Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
2	HashMap(int initialCapacity) Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
3	HashMap(int initialCapacity, float loadFactor) Constructs an empty HashMap with the specified initial capacity and load factor.
4	HashMap(Map<? extends K,? extends V> m) Constructs a new HashMap with the same mappings as the specified Map.

```java
import java.util.*;
class HashMapDemo {
    public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();
        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));
        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        // Display the set.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();
        //Deposit 1000 into John Doe's account.
        double balance = hm.get("John Doe");
        hm.put("John Doe", balance + 1000);
        System.out.println("John Doe's new balance: " +
        hm.get("John Doe"));
    }
}
```

```
Tod Hall: 99.22
John Doe: 3434.34
Ralph Smith: -19.08
Tom Smith: 123.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

Unit-12

Concurrency / Multithreading

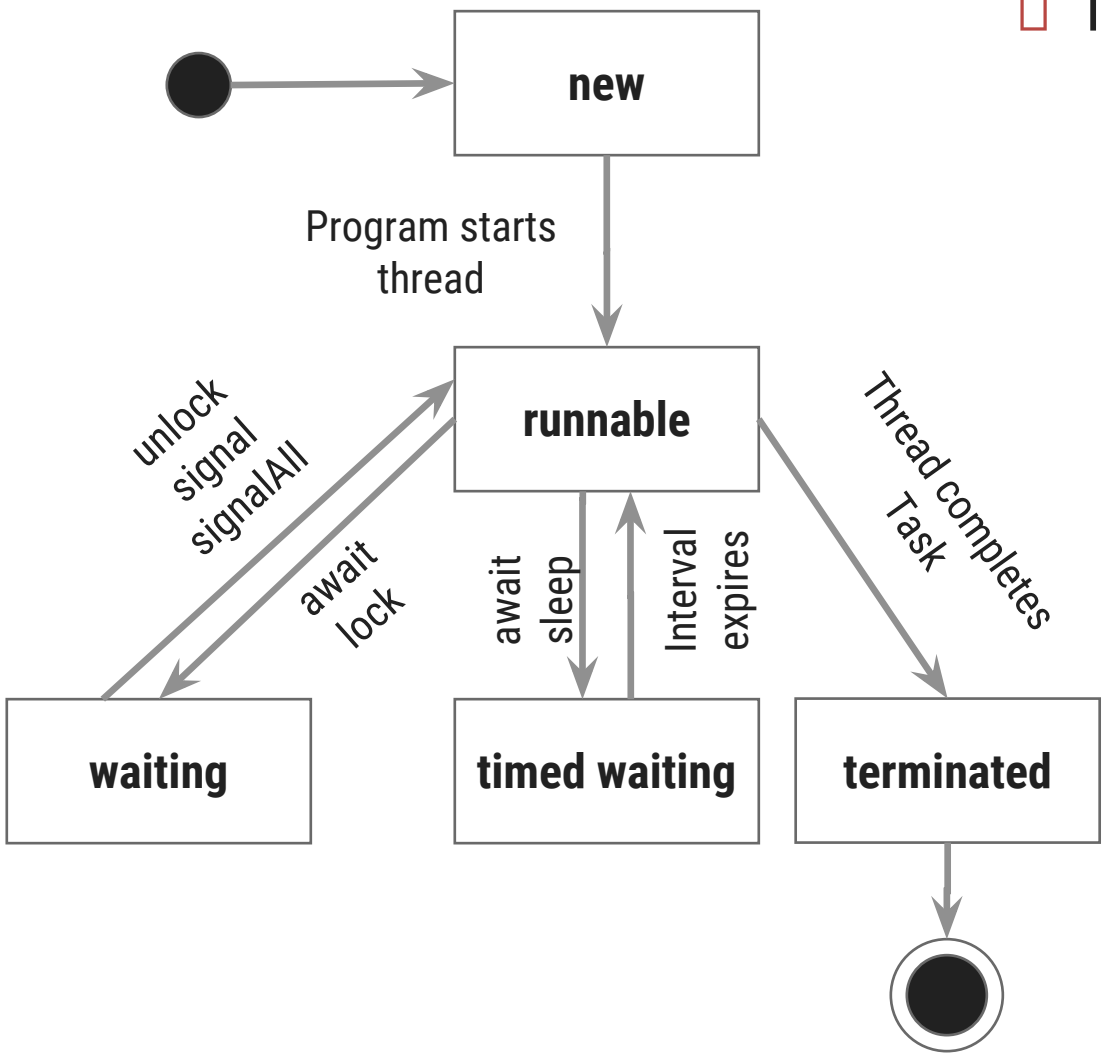
Outline

- ✔ Multithreading
- ✔ Life Cycle of Thread
- ✔ Creating a Thread
  - ✔ Using Thread Class
  - ✔ Using Runnable Interface
- ✔ Join
- ✔ Synchronized

What is Multithreading?

- Multithreading in Java is a process of *executing multiple threads* simultaneously.
- A thread is a *lightweight sub-process*, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- Threads use a *shared memory area*. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.
- Lets see the life cycle of the thread.

Life cycle of a Thread



- There are **5** stages in the life cycle of the Thread
  - **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
  - **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
  - **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals waiting thread to continue.
  - **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
  - **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating a Thread in Java

- There are two ways to create a Thread
  1. **extending** the **Thread class**
  2. **implementing** the **Runnable interface**

1) Extending Thread Class

- One way to create a thread is to create a new class that extends `Thread`, and then to create an instance of that class.
- The extending class must override the `run( )` method, which is the entry point for the new thread.
- It must also call `start( )` to begin execution of the new thread.

```
class NewThread extends Thread {
    NewThread() {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }}
```

```
class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread
        interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

2) Implementing Runnable Interface

- To implement thread using `Runnable` interface, `Runnable` interface needs to be implemented by the class.

```
class NewThread implements Runnable
```

- Class which implements `Runnable` interface should override the `run()` method which containts the logic of the thread.

```
public void run( )
```

- Instance of `Thread` class is created using following constructor.

```
Thread(Runnable threadOb, String threadName);
```

- Here `threadOb` is an instance of a class that implements the `Runnable` interface and the name of the new thread is specified by `threadName`.
- `start()` method of Thread class will invoke the `run()` method.

Example Runnable Interface

Thread using Executor Framework

- Steps to execute thread using Executor Framework are as follows:

1. Create a task (Runnable Object) to execute
2. Create Executor Pool using Executors
3. Pass tasks to Executor Pool
4. Shutdown the Executor Pool

Example Executable Framework

```java
class Task implements Runnable {
     private String name;
     public Task(String s) {
          name = s;
     }
     public void run() {
          try {
               for (int i = 1; i<=5; i++) {
                    System.out.println(name+
                    " - task number - "+i);
                    Thread.sleep(1000);
               }
               System.out.println(name+"
               complete");
          }
          catch(InterruptedException e) {
               e.printStackTrace();
          }
     }
}
```

```java
import java.util.concurrent.*;

public class ExecutorThreadDemo {
     public static void main(String[] args)
     {
          Runnable r1 = new Task("task 1");
          Runnable r2 = new Task("task 2");
          Runnable r3 = new Task("task 3");
          Runnable r4 = new Task("task 4");
          Runnable r5 = new Task("task 5");
          ExecutorService pool =
          Executors.newFixedThreadPool(3);
          pool.execute(r1);
          pool.execute(r2);
          pool.execute(r3);
          pool.execute(r4);
          pool.execute(r5);
          pool.shutdown();
     }
}
```

```
task 1 - task number - 1
task 2 - task number - 1
task 3 - task number - 1
task 1 - task number - 2
task 2 - task number - 2
task 3 - task number - 2
task 2 - task number - 3
task 1 - task number - 3
task 3 - task number - 3
task 1 - task number - 4
task 2 - task number - 4
task 3 - task number - 4
task 1 - task number - 5
task 2 - task number - 5
task 3 - task number - 5
task 2 complete
task 1 complete
task 4 - task number - 1
task 3 complete
task 5 - task number - 1
task 4 - task number - 2
task 5 - task number - 2
task 4 - task number - 3
task 5 - task number - 3
task 4 - task number - 4
task 5 - task number - 4
task 4 - task number - 5
task 5 - task number - 5
task 4 complete
task 5 complete
```

Thread Synchronization

- When we start *two or more threads* within a program, there may be a situation when multiple threads try to *access the same resource* and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.
- Java programming language provides a very handy way of creating threads and synchronizing their task by using *synchronized methods & synchronized blocks.*

Problem without synchronization (Example)

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 100 200 10 300 15 400 20 500 25
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

Solution with synchronized method

```
class Table {
 synchronized void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```

Solution with synchronized blocks

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(5);
    }
  }
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(100);
    }
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```