Rathindra Nath Dutta

Rathindra Nath Dutta

Computer Graphics using Java Swing

Rathindra Nath Dutta

Contents

Rathindra Nath Dutta

Rathindra Nath Dutta

Acknowledgment

SK sir!

Rathindra Nath Dutta

Rathindra Nath Dutta

List of Figures

Rathindra Nath Dutta

Rathindra Nath Dutta

List of Tables

Rathindra Nath Dutta

Rathindra Nath Dutta

Listings

Rathindra Nath Dutta

Rathindra Nath Dutta

Chapter 1

Introduction to Java Swing

Computer graphics is a field which studies the art of drawing pictures on the computer screen with the help of programming. With the help of computation one can create a synthetic digital image. Thus computer graphics deals with the computer-generated imagery (CGI) in both static (image) or variable (motion pictures or movies) form.

Computer generated graphics can range from a simple point to complicated textures to interactive graphical user interface (GUI). Currently many sophisticated software libraries directly allows us to generate visual elements like stylized text, window, buttons etc and even allows us to manipulate them during execution thus making the interface more interactive to the user.

1.1 About Swing

Java is one of the popular and yet easy to learn language. It is free and platform independent. Java is Object-Oriented, and library rich.

Swing was developed to address the deficiencies present in the Java's original GUI subsystem: the Abstract Window Toolkit (AWT). The AWT provides a basic set of controls, visual components like windows, buttons, dialog boxes etc. all of which had some limitations in their design and/or implementations. One such limitation is that visual element created using AWT are translated to platform-specific equivalents. In other words, look and feels of these components is defined by the underlying platform (mainly by the OS). Even the behavior of some components vary from one platform to another. This violates Java's core philosophy of platform-independence.

Before going any further it is absolute necessary to state that: although Swing was developed to eliminate the inherent limitations of AWT, Swing does not replaces it. Instead, Swing is built on top of AWT. For example, the event handling in Swing still uses the mechanisms implemented in AWT. Swing also introduces some new components apart from existing ones in AWT.

Swing solves the limitations present in AWT by its two key features. First one is Swing components are *lightweight*. AWT uses implementation code of the

Rathindra Nath Dutta

native platform to display its components. Hence AWT components are considered heavyweight. In contrast to this, Swing components are written entirely in Java. Therefore, Swing components look and feel and most importantly behave in consistently across all platforms. Secondly, Swing supports *pluggable look and feel*. This separates the look and feel of a component from the logic code that uses this components. It allows us to create custom look and feel as well as we can dynamically modify look and feel of a component even at runtime.

Swing closely follows the *MVC design pattern* [1]. MVC or Model-View-Controller framework defines Models, Views, Controller and makes their logic independent of each other. Model corresponds to the state of a component (e.g. in case of check-box, whether it is checked or not). View deals with how a component is displayed on the screen. View always reflect current state of the model for a component. Controller determines how a component reacts to some user interaction (e.g. in case of check-box, clicking on it updates its model, i.e. toggles its state between checked and unchecked). As soon as the state changes in Model, View almost instantaneously reflects the same on the screen.

Table 1.1 summarizes the above discussion. Beginning with Java 1.2, Swing was fully integrated into Java as part of the Java Foundation Classes (JFC) [2].

Table 1.1: AWT vs Swing

AWT	Swing
components are platform-dependent	components are platform-independent
components are heavyweight	components are lightweight
doesn't support pluggable look and feel	does support pluggable look and feel
provides less components than Swing	provides more [3] powerful components
doesn't follows MVC framewwork	based on MVC pattern

Rathindra Nath Dutta

[1]Note that Swing does not implement the classical MVC model.

[2]Java Foundation Classes, encompass a group of features for building GUIs and adding rich graphics functionality and interactivity to Java applications. It is defined as containing: Swing GUI Components, Pluggable Look-and-Feel Support, Accessibility API (for enabling assistive technologies, such as screen readers and Braille displays, to get information from the user interface), Java 2D API (for high-quality 2D graphics, text, and images), Internationalization (support for worldwide languages event those uses thousands of different characters, such as Japanese, Chinese, or Korean).

[3]More than 250 new classes and 75 interfaces were introduced in Swing; twice as many as was in AWT.

1.2 Components and Containers

A Swing GUI is made up of two key items: components and containers. In Swing almost every class name begins with the letter 'J' to denote they are Swing version of an old AWT class. Figure 1.1[4,5] gives an overview of available components and containers classes in Java. As it is evident from the class hierarchy in figure 1.1 a container is nothing but a special type of components and the distinction is based on their intended purpose.

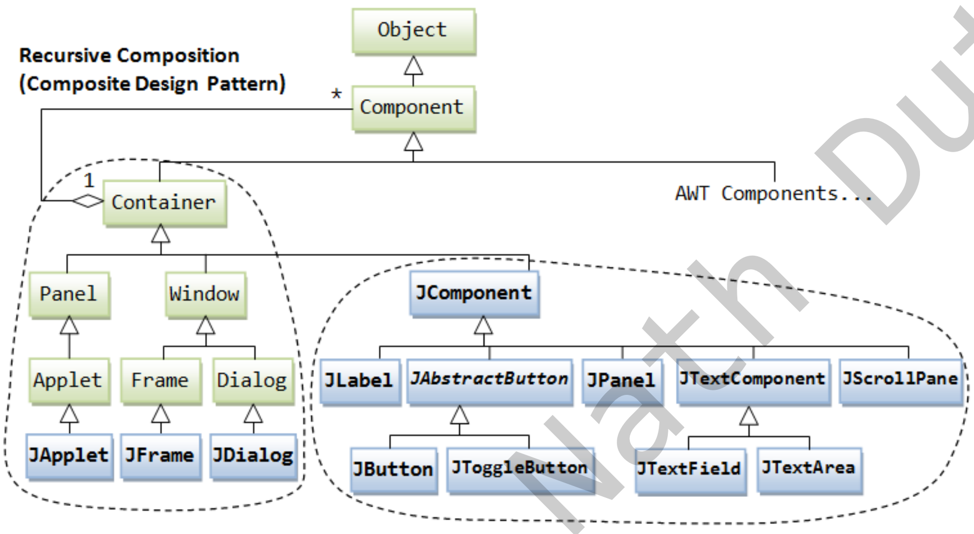


Figure 1.1: Components and Containers in Java Swing

Components

In Swing every components are derived from the `JComponent` class which inherits the AWT `Container` (and thus also `Component`). Figure 1.2 covers most of the components available in Swing.

Containers

Swing defines four top-level containers: `JFrame`, `JApplet`, `JWindow`, and `JDialog` as shown in the figure 1.3. These classes inherit from AWT `Container` (and thus also `Component`) and not from `JComponent`, thus they are heavyweight. as the name suggest these containers are not contained in any other container, and stays

Rathindra Nath Dutta

[4]Like any other class in Java the `Component` class also implicitly inherits the `Object` class.

[5]The Swing class names, the ones staring with 'J' are written in boldface, and the rest classes are part of AWT.

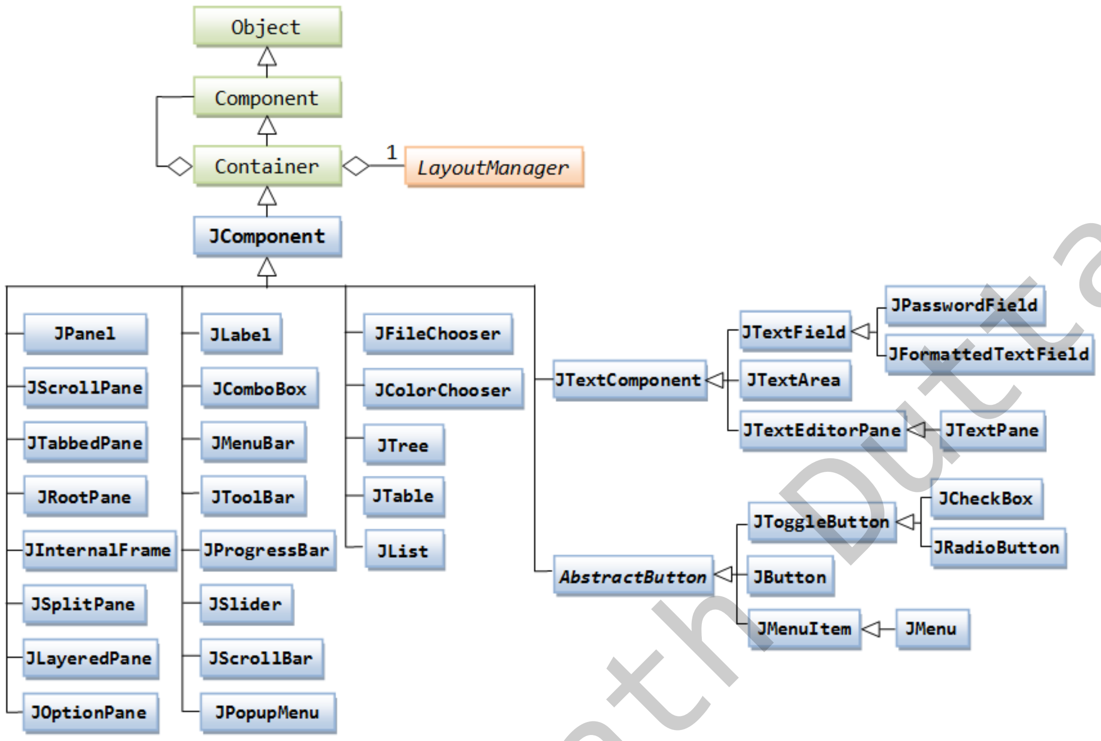


Figure 1.2: Class hierarchy of components in Java Swing

at the top-level of the containment hierarchy of the GUI design. For applications we commonly use `JFrame` whereas `JApplet` is used for applets[6].

Swing also defines a second type of containers that are lightweight as they inherits from `JComponent`. `JPanel` is one such popular container. These lightweight containers are used to contain other components and they itself in turn contained within some top-level container.

Packages

All of the Swing components and containers are organized under the main package[7] called `javax.swing`. For some functionalities we may also need to import from `java.awt` package.

[6]An applet is a special Java program that is embedded in a web page and thus runs in the Web Browser at client side.

[7]A Java package is a group of similar types of classes, interfaces and sub-packages. Packages are generally named in reverse domain notation (e.g. org.apache.commons.math) and reflects the directory hierarchy.

Rathindra Nath Dutta

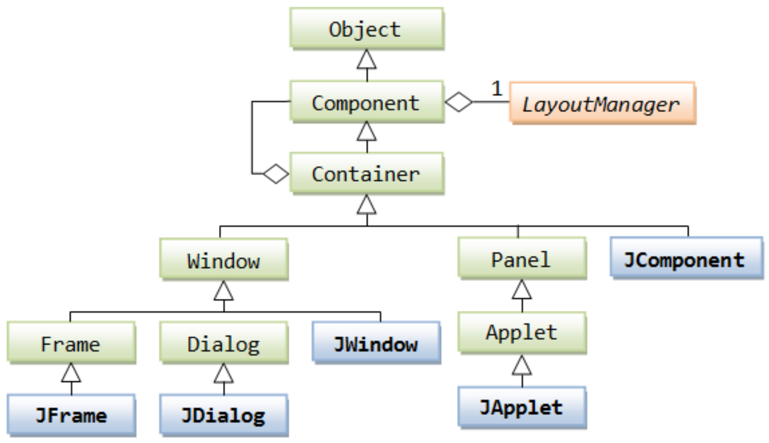


Figure 1.3: Class hierarchy of components in Java Swing

1.3 Coding a Swing Application

Consider that we want to build an application that shows the text ”Hello World” on a window. As mentioned earlier `JFrame` is used to create window based applications. Therefore, we need to create an object of `JFrame`.

```
JFrame myFrame = new JFrame();
```

Also we need another object of `JLabel` to display our text. Here we can directly pass the require text to be shown as a parameter in the constructor, which implicitly invokes the `setText()` method of the label.

```
JLabel myLabel = new JLabel("Hello World");
```

Now we need to put our `JLabel` object to our top-level container `myFrame`, the object of `JFrame` class. The `add()` method of any container obejct puts the component `myLabel`, passed as an argument, into the content pane of top-level container `myFrame`.

```
myFrame.add(myLabel);
```

We also need to set the frame size so that our window has the required width(200) and height(100).

```
myFrame.setSize(200, 100);
```

By default, a newly created `JFrame` object does not show up any window on the screen, the window stays hidden. So we need to make it visible.

```
myFrame.setVisible(true);
```

Rathindra Nath Dutta

Finally, we might want to use the close button (at the top right corner of the window) to close and dispose the window.

```
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Here `EXIT_ON_CLOSE` is a integer constant defined in the `JFrame` class a static[8] member, so we can directly access it from the class name itself.

Finally, we put all these inside a `main()` method of a demo class[9] `SwingTest` as shown in listing 1.1.

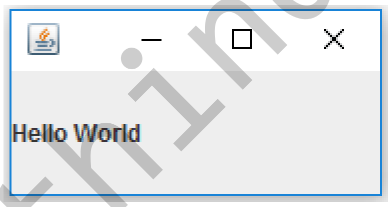
SwingTest.java

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class SwingTest {
    public static void main(String[] args) {
        JFrame myFrame = new JFrame();
        JLabel myLabel = new JLabel("Hello World");
        myFrame.add(myLabel);
        myFrame.setSize(200, 100);
        myFrame.setVisible(true);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Listing 1.1: Simple Swing Application

The output is shown below.



```
Hello World
```

[8]An object member defined in a class must be accessed (if permitted) through an intance of that class. In contrast, a static member (a member with the `static` access modifier keyword) behaves as a class member and thus directly accessible thought the class name itself, no instantiation required. This is why `main()` method is declared as static. Another example will be the methods available in the `Math` class.

[9]In Java class name should start with a capital letter, and member name should start with a small letter. They both generally uses camel case to increase readability. Java also recommends that the file name also should be same as the public class name.

Rathindra Nath Dutta

1.3.1 Coding Techniques and Best Practices

Although, the above Java code does serves the intended purpose, it is not a good coding style to follow for Swing application development. The following code depicts the standard coding practice.

MyFrame.java

```java
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {
    JLabel myLabel;

    public MyFrame() { //initialize the frame
        myLabel = new JLabel("Hello World");
        this.add(myLabel);
        this.setSize(200, 100);
        this.setVisible(true);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String args[]) {
        new MyFrame(); // instantiate our frame
    }
}
```

Listing 1.2: Standard Coding Style in Practice for Swing Applications

The first main change is that we are extending the `JFrame` class to create our own frame class. This class will contains all the components to be added on the frame as its members. In this example we have only a single component, a label, with the default access modifier[10]. One can use different access modifiers as required. Secondly, we are initializing our components and placing them onto our frame with in the constructor of our frame class. In this way user of `MyFrame` class is relieved from the burden of declaring the specifics of the frame and can simply instantiate the frame. If required, certain things can easily be customizable

[10]Java has four access modifiers: `private` (accessible from only within that class), `protected` (same as private but inheritable, thus accessible from only within that class and its subclasses), `default` (its the default one, accessible from all classes only within same package), and `public` accessible from anywhere within that application

Rathindra Nath Dutta

by passing arguments inside the constructor and utilizing them in order to meet specific user requirements. In this example, we have put the `main()` method inside our frame class, but in many cases a separate driver class is created for the main method.

Rathindra Nath Dutta

Chapter 2

Event Handling in Swing

Suppose we want develop an application which has a text-box a button and a label. When a user write something in the text-box and click on that button, the text is copied on to the label. In order to achieve this effect, we need to somehow specify the behavior of the button. This is achieved though event handling[1] in Java.

In simple words an *event* resembles happening of something. Like many other programming languages, in Java occurrence on a event causes some object to chance its state. In Java clicking mouse button, dragging mouse, pressing a key on keyboard etc. are considered as events.

2.1 The Delegation Event Model

Starting from Java 1.1, a new and better event handling approach was introduced. Here events are handled based on *delegation event model*, which is a standard and consistent mechanism to generate and process those events. A *source* generates an event and sends that to one or more *listener*(s). The job of a listener is to simply wait until it receives an event. Once an event is received, the listener processes it and returns.

The beauty of this approach is that, the event handling (processing) mechanism is completely separate from the event generation logic, and thus independent of each other. Thus an element of the user interface is able to 'delegate'[2] the processing of an event to a separate piece of code

In this model, an event is an object that describes change in state of the event source. This generally corresponds to some user activity like click, key press etc. A source must register at least some listener to listen for the desired event. This event handling flow is depicted later in figure **??**.

[1]Here we will focus on the GUI based events only. The event handling not related to GUI systems is done in similar fashion.

[2]Delegate(noun): to give authority or control to someone, entrust (a task or responsibility) to another person

Rathindra Nath Dutta

Table 2.1: AWT Events and Listeners

Event class	User Action/ Cause	Listener Interface
ActionEvent	button press, double click, selecting a menu item	ActionListener
AdjustmentEvent	scroll bar manipulated	AdjustmentListener
ComponentEvent	position, size, or visibility of a componet modified	ComponentListener
ContainerEvent	addition or removal of a component from a container	ContainerListener
FocusEvent	getting keyboarad focus or defocued	FocusListener
ItemEvent	check-box or cheackable menu item clicked or a list item is clicked	ItemListener
KeyEvent	keyborad input	KeyListener
MouseEvent	mouse click, drag, move, hover etc	MouseListener, MouseMotionListener
MouseWheelEvent	mouse wheel rotated	MouseWheelListener
TextEvent	value of the text field or text area changed	TextListener
WindowEvent	activating, deactivating, opening, closing, iconfication, deiconification, quiting a window	WindowListener, WindowFocusListener

2.2 Different Events in Java

Java provides a set of event classes and their corresponding listeners interfaces[3] in the package `java.awt.event`, few of which are described below. As shown in figure 2.1 all these event classes are part of AWT and inherits from `EventObject` of `java.util`. Although these classes and interfaces covers most of the events and their handling mechanisms, one can also define custom events and their listeners. Table 2.1 lists different AWT event classes, along with the user interactions that causes that particular event. The table also lists corresponding listener classes

[3]See Appendix A.1 for more detials.

Rathindra Nath Dutta

that must be registered by the source in order to listen for that particular event.

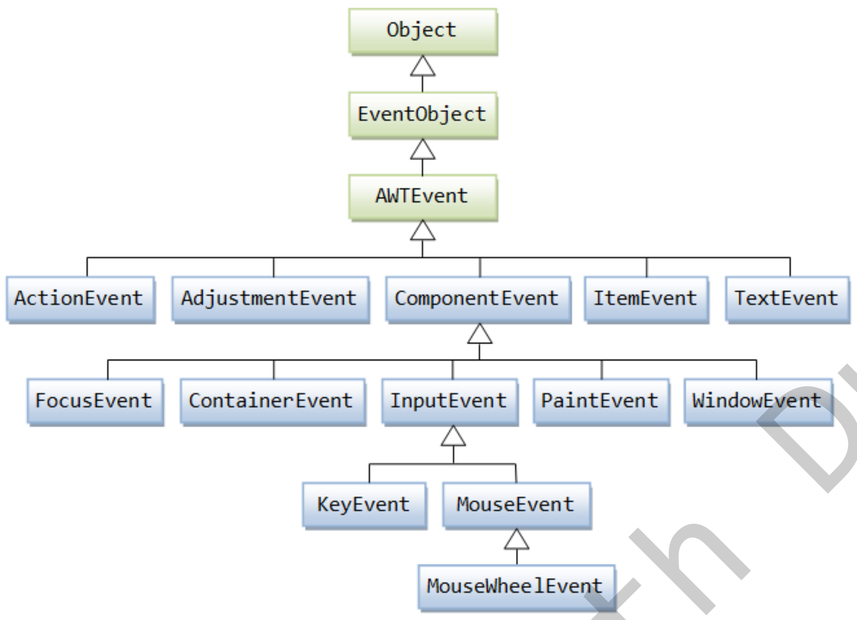


Figure 2.1: Class hierarchy of components in Java Swing

2.3 Coding to Handle Events

Consider the previous example, where we wanted to click a button to copy the text from a text-box into a label. Since we want to handle the button click event we can use the `ActionEvent` class. The source(button in this case) first registers a `ActionListener` for this event. To register this listener we need to call `addActionListener()` method as pass an `ActionListener` instance. We can make our `MyFrame` class to implement the `ActionListener` interface and pass its current instance as an action listener object into the `addActionListener()` method as follows:

```
JButton button = new JButton("Copy");
button.addActionListener(this);
```

`ActionListener` interface has a single unimplemented method `actionPerformed()` in it, which now needs to be implemented in our `MyFrame` class. Whenever the button click action is performed this method will automatically be triggered.

```
@Override
public void actionPerformed(ActionEvent e) {
```

Rathindra Nath Dutta

```
    //code to be executed when an ActionEvent occurs
}
```

The @Override is an *annotation* to indicate that this method declaration is intended to override a method declaration in a supertype. It is not required to put this annotation before a overridden method declaration. If a method is annotated with this annotation type, compilers are required to generate an error message unless there exists an override-able method in the superclass.

Since we wanted to copy the text from our text-box into our label we may write the following inside our `actionPerformed()` method.

```
label.setText(textbox.getText());
```

By default a `JFrame` instance has `BorderLayout`[4] manager set on it. So we will use an instance of `JPanel` which has the `FlowLayout`[5] as its default layout manager[6]. Finally, we put these all together as shown in listing 2.1.

MyFrame.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class MyFrame extends JFrame implements ActionListener {
    JPanel panel;
    JTextField textbox;
    JButton button;
    JLabel label;

    public MyFrame() {
```

[4]BorderLayout divides the content pane into five sections: TOP, BOTTOM, LEFT, RIGHT, and CENTER. Components added to a container having BorderLayout can be only aligned to these five regions.

[5]FlowLayout simply lays out components in a single row from left to right, starting a new row if its container is not sufficiently wide and the rows are filled from top to bottom.

[6]Java has various layouts for variour needs. For example GridLayout can be used to organize buttons in a calculator. All layouts implement the LayoutManagaer interface. For more details refer to: `https://ratcoinc.github.io/Java_Swing/oracle_tutorial/uiswing/layout/`

Rathindra Nath Dutta

```java
        panel = new JPanel();

        textbox = new JTextField(10); //sets the width to 10
        ↪ columns/characters
        panel.add(textbox);

        button = new JButton("Copy");
        panel.add(button);
        button.addActionListener(this);

        label = new JLabel();
        panel.add(label);

        this.add(panel);

        this.setSize(400, 100);
        this.setVisible(true);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        new MyFrame();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        label.setText(textbox.getText());
    }
}
```

Listing 2.1: A Simple Event Handling

2.4 Handling Events From Multiple Sources

Consider another example, where we have to change a text color. We have two buttons one to change the color to red another for blue. We can create a class which extends `JFrame` and implements `ActionListener` as well as override the `actionPerformed()` method as we did in the previous example. Both buttons must register a listener. We can use current instance of our class as the listener for

Rathindra Nath Dutta

both. Now clicking on either of the button invokes the same `actionPerformed()` method. Thus, we need some way to distinguish between the two button clicks. We can use the `getActionCommand()`[7] method of the action event, and test the command name against the label of the button as shown below. The `Color` class provides some predefined instances for the common colors like `Color.RED`, `Color.BLUE` etc. The `Color` class is described later in section 3.2.

```java
@Override
public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equalsIgnoreCase("red")) {
        label.setForeground(Color.RED);
    } else {
        label.setForeground(Color.BLUE);
    }
}
```

Alternatively, we can use the `getSource()`[8] of the action event and compare it against the available sources(buttons in this case).

```java
@Override
public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == red) {
        label.setForeground(Color.RED);
    } else {
        label.setForeground(Color.BLUE);
    }
}
```

Finally, we put all these together into listing 2.2.

MyFrame.java

```java
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

Rathindra Nath Dutta

[7]`getActionCommand()` returns the command string associated with this action. It is component specific and should be carefully used.

[8]`getSource()` returns the object on which the Event initially occurred. It is more general and preferable.

```java
import javax.swing.JPanel;
public class MyFrame extends JFrame implements ActionListener {
    JPanel panel;
    JButton red, blue;
    JLabel label;

    public MyFrame() {
        panel = new JPanel();

        red = new JButton("Red");
        panel.add(red);
        red.addActionListener(this);
        blue = new JButton("Blue");
        panel.add(blue);
        blue.addActionListener(this);

        label = new JLabel("This is a text");
        panel.add(label);

        this.add(panel);
        this.pack();
        this.setVisible(true);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        new MyFrame();
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if (ae.getSource() == red) {
            label.setForeground(Color.RED);
        } else {
            label.setForeground(Color.BLUE);
        }
    }
}
```

Rathindra Nath Dutta

Listing 2.2: Handling Events From Multiple Sources

2.5 Event Handling With Anonymous Inner Classes

Being an inner class, any *anonymous inner class* has access to all of the members of its enclosing class. Moreover, any method definition provided inside the body of the anonymous inner class is local to the instance created by that expression. This becomes very handy when using event listeners as shown below.

MyFrame.java

```java
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MyFrame extends JFrame {
    JPanel panel;
    JButton red, blue;
    JLabel label;

    public MyFrame() {
        panel = new JPanel();
        red = new JButton("Red");
        panel.add(red);
        red.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setForeground(Color.RED);
            }
        });
        blue = new JButton("Blue");
        panel.add(blue);
        blue.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
```

Rathindra Nath Dutta

```
                    label.setForeground(Color.BLUE);
                }
            });

            label = new JLabel("This is a text");
            panel.add(label);

            this.add(panel);
            this.pack();
            this.setVisible(true);
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        }

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

Listing 2.3: Event Handling With Anonymous Inner Classes

Here we are using anonymous inner class to create instances of `ActionListener` interface for our two buttons. Recall that the `addActionListener()` method requires an `ActionListener` instance as its argument. Previously we were making our frame class to implement the listener so that we can pass `this` as the argument to `addActionListener()`. Instead, here we are creating anonymous inner class instances for the listener and passing that as the argument. In this way the design also becomes simpler. Each anonymous inner class instance of `ActionListener` has its own `addActionListener()` method where we have put the code for action to be performed corresponding to that button.

Since, each instance of `ActionListener` is registered only to a single source(a button in this example), we no longer need to worry about the source of event, as we did earlier. Thus, using anonymous inner class has become a standard practice specially for handling events from multiple sources.

Further Reading

There are many type of events only some of which are discussed in this document. To know more about them you may refer to chapter 23 of [Schildt, 2011] and `http://ratcoinc.github.io/Java_Swing/oracle_tutorial/uiswing/events/`.

Rathindra Nath Dutta

Rathindra Nath Dutta

CHAPTER 3

Working with Graphics

Java AWT provides a rich set of graphics APIs that serves almost every need of a UI programmer. Swing uses the same graphics methods available in AWT. Every component in Swing has its own *graphics context*. Drawing(output using graphics context) on any component is done relative to its top-left corner. The top-left cornet is considered origin (0, 0), going down increases x-coordinate and going left increases y-coordinate.

3.1 Drawing with Graphics

Graphics context of any component can be acquired by calling `getGraphics()` method of that component. The `getGraphics()` method return an instance of the `Graphics` class defined in `java.awt` package.

The `repaint()` method is provided by any component inhering `Component` class from `java.awt`. The `repaint()` method repaints this component by calling either `paint()` method (for a lightweight component) or `update()` (for a heavy-weight component) as soon as possible. Both `paint()` and `update()` methods are get invoked with the current graphics context as their parameter.

In swing, every component inherits from the `JComponent`, which provides a `paintComponent()` method. It is recommended to override `paintComponent()` instead of `paint()`.

3.1.1 Drawing and Filling Primitive Shapes

The `Graphics` class defines a number of drawing methods to draw and fill different basic shapes discussed below. All these objects are drawn or filled with the current color of the graphics context, which is black by default. The component's dimension defines the viewport and any object having portion outside the boundary of the component gets automatically clipped out.

Rathindra Nath Dutta

Lines

A line segment is defined by its two end points (x_1, y_1) and (x_2, y_2). Such a line can be drawn by `void drawLine(int x1, int y1, int x2, int y2)` method.

Rectangles

A rectangle can be defined by specifying its top-left corner along with its width and height. Such a rectangle can be drawn or filled by the following methods

```
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)
```

If the width and height value are same then the drawn rectangle is a square.

Ovals

An ellipse can be drawn or filled by the following methods

```
void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)
```

The ellipse is actually fit into the rectangle specified by the parameters. If the width and height value are same then the drawn ellipse is actually a circle. Unfortunately, there is no such method available to draw ia point, `fillOval()` method is often used to plot a point.

Arcs

A rectangle can be defined by specifying its top-left corner along with its width and height. Such a rectangle can be drawn or filled by the following methods

```
void drawArc(int x, int y, int width, int height,
                int startAngle, int arcAngle)
void filleArc(int x, int y, int width, int height,
                int startAngle, int arcAngle)
```

They draw the outline of a circular or elliptical arc within the specified rectangle. The resulting arc begins at startAngle and extends for arcAngle degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation. The center of the arc is the center of the rectangle specified.

Rathindra Nath Dutta

Polygons

A polygon is a bounded region which can be difed by its bounding lines. Alternatively, a set of point can also define a polygon, where each consecutive pair of points denoted a bounding line of the polygon. Any arbitrary shaped polygon can be drawn or filled by the following methods

```
void drawPolygon(int xPoints[], int yPoints[], int nPoints)
void fillPolygon(int xPoints[], int yPoints[], int nPoints)
```

Here the points are denoted by $(xPoints[i], yPoints[i])$ with $0 \leq i < nPoints$.

3.2 Colors

The `Color` class is defined in the `java.awt` package, and uses the sRGB model to render the colors. The `Color` class contains some predefined instances common colors (e.g. instance for red color can be accessed by writing `Color.RED`). These instances are defined as `static`(class members) and `final`(constant) with the `public` access modifier. One can define any color using the one of the following constructors.

```
Color c = new Color(int r, int g, int b);
Color c = new Color(int r, int g, int b, int a);
```

The parameters `r`, `g`, and `b` corresponds to the color values of Red, Green, and Blue respectively. The parameter `a` denotes alpha value which is used for transparency.

`Color` class provides several methods to create color using HSB color model, or to get a particular color component, or to convert between RGB and HSB color models etc. The full documentation of `Color` class is available at `https://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/Color.html`

3.2.1 Changing Graphics Color

The color of the graphics context for current component can be changed by the following method.

```
void setColor(Color c)
```

The next shape drawn on this graphics context will have the specified color (already draws shaped won't be affected). There is also another method to retrieve current color the graphics context.

```
Color getColor()
```

Rathindra Nath Dutta

3.3 Code Example: Translating and Scaling a Line

The above discussions can be better explained by the following example. This example also demonstrates usage of various mouse events and their listeners.

Suppose we want develop an application that can draw a line based on user input. Then it allows to select the line and drag it on the screen using mouse and it also allows to zoom/shrink the line by scrolling the mouse wheel.

As it clearly evident that here the we need to save the user inputs (for the line) in order to transform it later. We can think that the line as an object that gets updated whenever the line is transformed (translated or scaled). So, we need a class for this as shown below.

MyLine.java

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;

public class MyLine {
    Point start, end;
    Color c;

    public MyLine(int startX, int startY, int endX, int endY,
     ↪ Color c) {
        this.start = new Point(startX, startY);
        this.end = new Point(endX, endY);
        this.c = c;
    }

    public void translate(int tx, int ty) {
        //A line can be translated by simply translating its
         ↪ two extremities
        start.x += tx;
        start.y += ty;
        end.x += tx;
        end.y += ty;
    }

    public void scale(double sx, double sy) {
```

Rathindra Nath Dutta

```
        //here scaling is done with respect to the start point
        int x = start.x, y = start.y;
        translate(-x, -y); //translate to (0,0)
        //scale the line
        //since start is at (0,0) scaling wont have any effect
        ↪  on it, only the end point needs to be updated
        //end.x *= sy is syntactic sugar for end.x =
        ↪  (int)(end.x * sy)
        end.x *= sx;
        end.y *= sy;
        translate(x, y); //back translate
    }

    /* This method tests whether the given point
     * is on the line segment or not
     *
     * There are many ways to check that
     * one simple way to test is by calculating Euclidean
↪   distances
     * iff dist(start,p) + dist(p,end) = dist(start,end)
     * then the pont p is on the line segment
     */
    public boolean doesContain(Point p) {
        double d1,d2,d3;
        d1 = Math.sqrt((start.x - p.x)*(start.x - p.x) +
        ↪  (start.y - p.y)*(start.y - p.y));
        d2 = Math.sqrt((end.x - p.x)*(end.x - p.x) + (end.y -
        ↪  p.y)*(end.y - p.y));
        d3 = Math.sqrt((start.x - end.x)*(start.x - end.x) +
        ↪  (start.y - end.y)*(start.y - end.y));
        return d1+d2==d3 ? true : false;
    }

    public void draw(Graphics g) {
        g.setColor(c);
        g.drawLine(start.x, start.y, end.x, end.y);
    }
}
```

Rathindra Nath Dutta

This class uses the Point class provided by AWT, which simply represents a point located at (x, y). This class also defines a few methods that will be required in the following classes.

Now we extend the `JPanel` class to create our special panel for drawing. Although is not required to create this seperate class, but it will simplify the design and makes it modular. Here the only thing to draw is a line, so we keep an instance of `MyLine` and re-implement the `paint()` method to display the line. As discussed earlier in section 3.1, that calling `repaint()` for some component invokes its `paint()` method, and thus our line will get drawn. When a component gets initialized its `paint()` is called for once, so we need to be careful that it then doesn't try to paint the `MyLine` instance that is yet to be initialized (i.e. set to `null`). The class definition looks like as follows.

DrawPanel.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JPanel;

public class DrawPanel extends JPanel {
    MyLine line;

    public DrawPanel() {
        //creating a dummy line
        line = new MyLine(10, 10, 100, 100, Color.BLACK);

        this.setBackground(Color.WHITE);
        //setting our canvas size to be 500x500
        this.setPreferredSize(new Dimension(500, 500));
    }

    @Override
    public void paint(Graphics g) {
        //super.paint(g) should be called from the
         ↪  reimplemented method
        //so that lightweight components are properly rendered
        super.paint(g);

        line.draw(g); //draw our line
    }
```

Rathindra Nath Dutta

```
}
```

Finlay, we design our UI by extending `JFrame`. Since also want to take input from user we create four text-boxes, a few radio buttons and a button. This class also registers different listeners. The code for this class is given below.

MyFrame.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JTextField;

public class MyFrame extends JFrame {
    JTextField x1, y1, x2, y2;
    JRadioButton black, blue;
    ButtonGroup colors;
    JButton draw;
    JPanel inputPanel;
    DrawPanel dp;

    private Color currentColor;
    private Point prevPoint = null;
    private boolean lineSelected = false;

    public MyFrame() {
        initComponents();
        setListeners();
```

Rathindra Nath Dutta

```

        this.pack();
        this.setVisible(true);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void initComponents() {
        inputPanel = new JPanel();

        x1 = new JTextField(5);
        y1 = new JTextField(5);
        x2 = new JTextField(5);
        y2 = new JTextField(5);
        inputPanel.add(x1);
        inputPanel.add(y1);
        inputPanel.add(x2);
        inputPanel.add(y2);
        //set tool tips, so that a message is displays
        //when the cursor lingers over the component
        x1.setToolTipText("startX");
        y1.setToolTipText("startY");
        x2.setToolTipText("endX");
        y2.setToolTipText("endY");

        //using ButtonGroup causes only one of the
        //radio buttons to be selected at a time
        colors = new ButtonGroup();
        black = new JRadioButton("Black");
        blue = new JRadioButton("Blue");
        colors.add(black);
        colors.add(blue);
        inputPanel.add(black);
        inputPanel.add(blue);

        draw = new JButton("Draw");
        inputPanel.add(draw);

        //put the inputPanel to the top side of our frame
        this.add(inputPanel, BorderLayout.NORTH);

```

Rathindra Nath Dutta

```
        dp = new DrawPanel();
        //put our drawPanel to the bottom side of the frame
        this.add(dp, BorderLayout.SOUTH);
    }

    private void setListeners() {
        black.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                currentColor = Color.BLACK;
            }
        });
        blue.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                currentColor = Color.BLUE;
            }
        });

        draw.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                dp.line = new MyLine(
                    Integer.parseInt(x1.getText()),
                    Integer.parseInt(y1.getText()),
                    Integer.parseInt(x2.getText()),
                    Integer.parseInt(y2.getText()),
                    currentColor);
                repaint();
            }
        });

        dp.addMouseListener(new MouseListener() {
            @Override
            public void mouseReleased(MouseEvent e) {
                lineSelected = false;
                prevPoint = null;
            }
            @Override
            public void mousePressed(MouseEvent e) {
```

Rathindra Nath Dutta

```java
                Point clickPoint = e.getPoint();
                if (dp.line.doesContain(clickPoint)) {
                    prevPoint = clickPoint;
                    lineSelected = true;
                }
            }
            @Override
            public void mouseExited(MouseEvent e) {}
            @Override
            public void mouseEntered(MouseEvent e) {}
            @Override
            public void mouseClicked(MouseEvent e) {}
        });

        dp.addMouseMotionListener(new MouseMotionListener() {
            @Override
            public void mouseMoved(MouseEvent e) {
            }
            @Override
            public void mouseDragged(MouseEvent e) {
                if (lineSelected) {
                    Point currentPoint = e.getPoint();
                    dp.line.translate(currentPoint.x - prevPoint.x,
                            currentPoint.y - prevPoint.y);
                    repaint();
                    prevPoint = currentPoint;
                }
            }
        });

        dp.addMouseWheelListener(new MouseWheelListener() {
            @Override
            public void mouseWheelMoved(MouseWheelEvent e) {
                if(e.getWheelRotation() < 0) //up scroll
                    dp.line.scale(1.2, 1.2); //zoom
                else //down scroll
                    dp.line.scale(1/1.2, 1/1.2); //shrink
                repaint();
            }
```

Rathindra Nath Dutta

```
        });
    }

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

Listing 3.1: Translating and Scaling a Line Using Mouse

The `MyFrame` class has two `private` methods. In `initComponents()` method we initialize various UI components and organize then on the frame. Whereas, `setListeners()` method registers different listeners. First we register action listeners for the two radio-buttons to select the color. These two list listeners simply updates a class field called `currentColor`. We also register another action listeners for the draw-button. This one reads the values from the text-fields and creates a new line.

Now, let us see how the transformation actually works. In order to do the translation we need to click exctly on the line(which is drawn on our draw-panel) and drag it. First, we register a `MouseListener` for our draw panel. Now when we press the mouse (left) button, we first check that whether the mouse pointer is actually on the line or not. We call the `getPoint()` method of the `MouseEvent` instance to get the current position(a point) of the mouse pointer. Then we invoke our `doesContain()` method of the `MyLine` class. If the point is on the line we set a flag, `lineSelected` to true, and set the `prevPoint` to this point. Moreover, we reset these two variables when the mouse button is released.

We also need to register a `MouseMotionListener` in oder to translate the line as the mouse is dragged over the draw panel. To do this, we get the current mouse position(similarly as above), calculate the t_x and t_y values as the coordinate difference between current and previous position of the mouse. Then we invoke the `translate()` method of `MyLine`. Now the line is translated but we need to call `repaint()` in order to see the effect. Finally, we update the previous point to be the current point. Note that we need to do all of these only if we had actually clicked on the line. This can easily be determined from the `lineSelected` flag.

Scaling is taken care by the `MouseWheelListener`. Lets assume that when we scroll the mouse wheel up we want to zoom the line, and scrolling down shrinks the line. The We invoke the `getWheelRotation()` of `MouseWheelEvent` to detect the direction of scrolling. Then we simply invoke the `scale()` of `MyLine` with $s_x = 1.2$, $s_y = 1.2$ for zooming and $s_x = 1/1.2$, $s_y = 1/1.2$ for shrinking the line. Finally, we invoke `repaint()` to see the scaled line.

Rathindra Nath Dutta

For more details on various mouse events see section B.1.

3.4 Paint Mode

By default, whenever we draw a new object it overwrites anything beneath it. However it is possible to draw in XOR mode.

```
void setXORMode(Color xorColor)
```

It sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified *xorColor*. When drawing operations are performed, pixels which are the current color are changed to the specified color, and vice versa. This guarantees that the new object is always visible. To return to default overwrite mode we have the following one.

```
void setPaintMode()
```

3.5 Showing Texts

The `Graphics` class also provides a method to display on a component.

```
void drawString(String str, int x, int y)
```

here (x, y) denotes the position to display the given string *str*. There is also a `Font` class available to change the font and style of the graphics context. `https://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/Font.html` provides the full details of the font API.

Rathindra Nath Dutta

CHAPTER 4

Working with 2D Graphics

Java provides more sophisticated control over the graphics elements in the 2D space. The 2D API provides following capabilities:

- A uniform rendering model for display devices and printers
- A wide range of geometric primitives, such as curves, rectangles, and ellipses, as well as a mechanism for rendering virtually any geometric shape
- Mechanisms for performing hit detection on shapes, text, and images
- A compositing model that provides control over how overlapping objects are rendered
- Enhanced color support that facilitates color management
- Support for printing complex documents
- Control of the quality of the rendering through the use of rendering hints

The 2D API maintains a two coordinate spaces:

User space is a device-independent logical coordinate system, the coordinate space that the program uses. All geometries passed into Java 2D rendering routines are specified in user-space coordinates.

Device space is a device-dependent coordinate system that varies according to the target rendering device such as a screen, window, or a printer.

The coordinate system for a window or screen might be very different from the coordinate system of a printer, these differences should be transparent to programmers. Thus the necessary conversions between user space and device space are performed automatically during rendering.

By default the origin of user space is the upper-left corner of the component's drawing area. The x coordinate increases to the right, and the y coordinate increases downward. The top-left corner of a window is 0,0. All coordinates are specified using integers, which is usually sufficient. However, in some cases require floating point or even double precision which are also supported.

Rathindra Nath Dutta

4.1 The `Graphics2D` Class

The Java 2D API includes the `Graphics2D`[1] class, which extends the `Graphics` class to provide access to the enhanced graphics and rendering features of the Java 2D API. These features include:

- Rendering the outline of any geometric primitive, using the stroke and paint attributes using `draw` method.
- Rendering any geometric primitive by filling its interior with the color or pattern specified by the paint attributes using `fill` method.
- Rendering any text string using the `drawString` method.
- Rendering the specified image using the `drawImage` method.

The methods available in Graphics2D class can be divided into two groups: methods to draw or fill a shape and methods that affect rendering. The second group of the methods serves the following purposes in the `Graphics2D` context:

- Vary the stroke width
- Change how strokes are joined together
- Set a clipping path to limit the area that is rendered
- Translate, rotate, scale, or shear objects when they are rendered
- Define colors and patterns to fill shapes with
- Specify how to compose multiple graphics objects

It was mentioned earlier that, the current graphics context for any component can be obtained through its `paint` or `update` method. In order to get the `Graphics2D` context of that component we can simply use type casting as follows:

```
public void paint (Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    ...
}
```

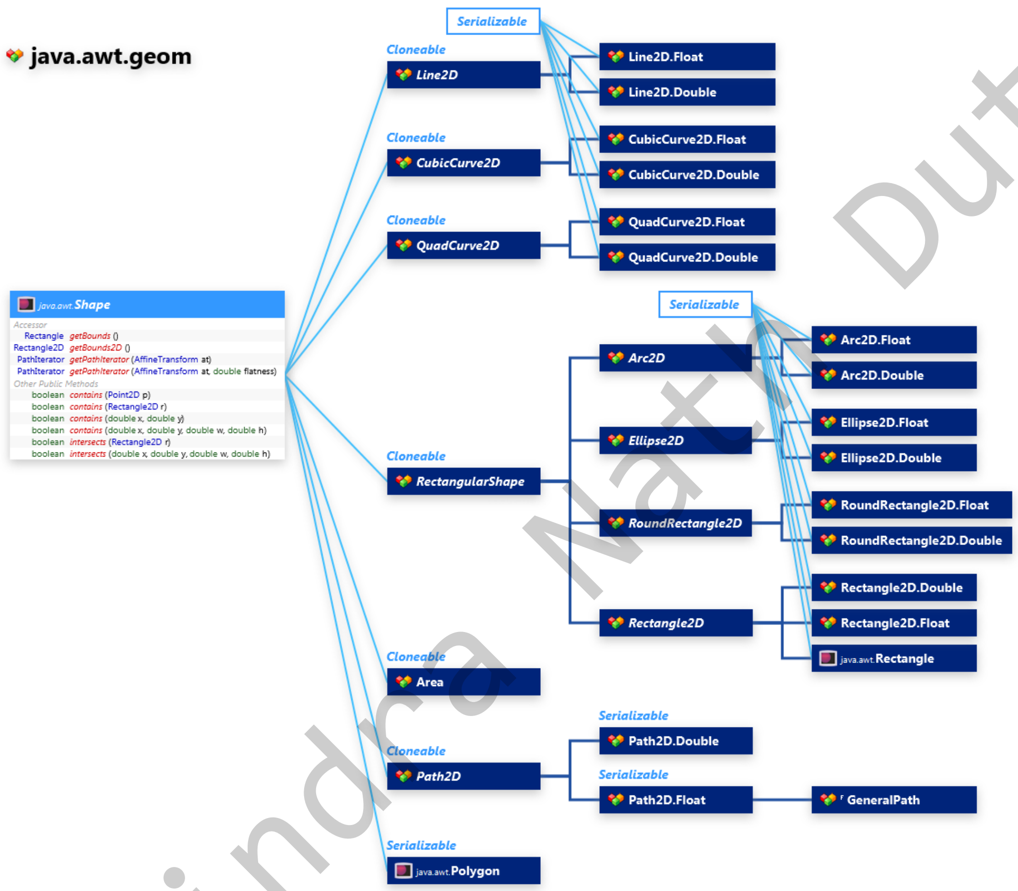
For more details on `Graphics2D` class and available methods see `http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/Graphics2D.html`

[1]The `Graphics2D` class is available in `java.awt` package.

Rathindra Nath Dutta

4.1.1 2D Geometric Primitives

The Java 2D API provides a useful set of standard shapes such as points, lines, rectangles, arcs, ellipses, and curves mainly in the `java.awt.geom` package. Arbitrary shapes can be represented by combinations of these standard shapes. Figure 4.1 shows a list classes and interfaces corresponding to these shapes.



```
java.awt.Shape
Accessor
   Rectangle  getBounds ()
 Rectangle2D  getBounds2D ()
PathIterator  getPathIterator (AffineTransform at)
PathIterator  getPathIterator (AffineTransform at, double flatness)
Other Public Methods
     boolean  contains (Point2D p)
     boolean  contains (Rectangle2D r)
     boolean  contains (double x, double y)
     boolean  contains (double x, double y, double w, double h)
     boolean  intersects (Rectangle2D r)
     boolean  intersects (double x, double y, double w, double h)
```

Figure 4.1: Hierarchy of 2D Geometric Primitives

Rathindra Nath Dutta

Shape

The `Shape` interface represents any geometric shape having an outline and an interior. This interface provides a common set of methods for describing and inspecting two-dimensional geometric objects and supports curved line segments and multiple sub-shapes. It contains the following methods:

```
Rectangle getBounds()
Rectangle2D getBounds2D()
```

The getBounds() returns an (integer) Rectangle instance representing the smallest bounding box that completely encloses the shape. Whereas, getBounds2D() gives a high precision and more accurate (floating point) bounding box as an Rectangle2D instance.

```
boolean contains(double x, double y)
boolean contains(Point2D p)
boolean contains(double x, double y, double w, double h)
boolean contains(Rectangle2D r)
```

The method contains(double x, double y) tests if the specified (x, y) point is inside shape and contains(Point2D p) does the same for a point represented by a Point2D instance. The contains(double x, double y, double w, double h) method tests if the shape entirely contains the specified rectangular area defined by its top-left corner point (x, y) along with the width and height of the rectangle. The contains(Rectangle2D r) method does the same for a Rectangle2D instance.

```
boolean intersects(double x, double y, double w, double h)
boolean intersects(Rectangle2D r)
```

The intersects() methods test whether the interior of the Shape intersects the interior of a specified rectangular area.

For more details see http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/Shape.html.

Points

The Point2D class defines a point representing a location (x, y). Note that the term "point" here is not the same as a pixel. A point has no area, does not contain a color, and cannot be rendered. Points are used to create other shapes. The Point2D class also includes methods for calculating the distance between two points. This is an abstract class thus cannot be instantiated directly. It contains two *static nested classes* Point2D.Double and Point2D.Float to instantiate a Point2D object in double and float precision respectively. For more details see http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/Point2D.html.

Lines

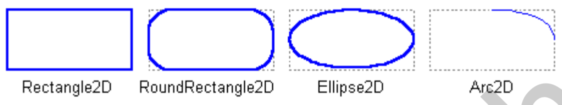
The Line2D class, a subclass of Shape, is an abstract class that represents a line. This is an abstract class and contains two *static nested classes* Line2D.Double and

Rathindra Nath Dutta

Line2D.Float to instantiate a Line2D object in double and float precision respectively. For more details see http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/Line2D.html

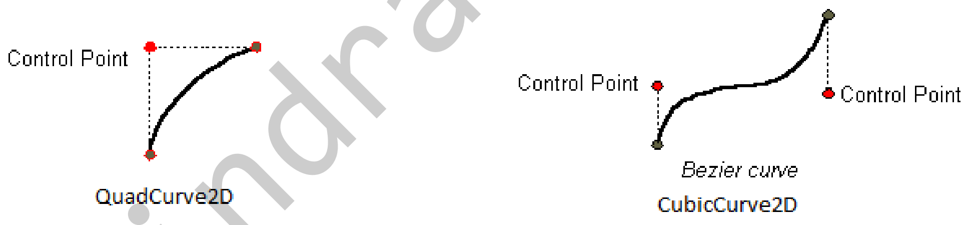
Rectangles

The RectangularShape is an abstract base class that defines rectangular frame or bounding boxes for other shapes. The Rectangle2D, RoundRectangle2D, Arc2D, and Ellipse2D classes are all derived from the RectangularShape class which is a subclass of Shape. These four classes are again abstract classes and contain two *static nested classes* to create instances in double or float precision. For more details see http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/RectangularShape.html.



Curves

The QuadCurve2D class enables us to create quadratic parametric curve segments where a quadratic curve is defined by two endpoints and one control point.The CubicCurve2D class enables us to create cubic parametric curve segments where a cubic curve is defined by two endpoints and two control points. These two classes



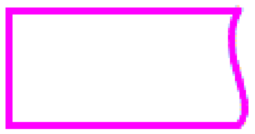
are again abstract classes and contain two *static nested classes* to create instances in double or float precision. For more details see http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/QuadCurve2D.html and http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/CubicCurve2D.html.

Arbitrary Shapes

The GeneralPath class enables you to construct an arbitrary shape by specifying a series of positions along the shape's boundary. These positions can be

Rathindra Nath Dutta

connected by line segments, quadratic curves, or cubic (Bézier) curves. The following shape can be created with three line segments and a cubic curve. For more details see `http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/GeneralPath.html`.



Area

With the `Area` class, one can perform boolean operations, such as *union*, *intersection*, *subtraction*, and *exclusive or* on any two Shape objects. This technique, often referred to as *constructive area geometry*, enables us to easily create complex shapes. For more details see `http://ratcoinc.github.io/Java_Swing/oracle_docs/api/java/awt/geom/Area.html`.

Rathindra Nath Dutta

Appendix A

Java Basics

A.1 Interfaces

An interface is a special type of class which is fully abstract. The followings describe an interface:

- The keyword `interface` is used instead of `class` when defining an interface. File containing an interface still has the .java extension, and when compiled produces .class file(s) like any other java classes.
- Every method in an interface are abstract.
- An interface can only contain members qualified with both `static` and `final`.
- One cannot instantiate an interface, they are designed only to be inherited by some other class or interface.
- A class `implements` an interface, while `extends` keyword is used for inheriting classes.
- Interface allows multiple inheritance.
- A class that implements an interface, must also define all of the unimplemented(abstract) methods in that interface.
- Sometimes one might wish to leave some of the methods to be implemented later by some other derived class, in such cases this class must be declared either as an `interface` or as a `abstract` class depending on the situation.

Shape.java

```
public interface Shape {
        public double getArea();
        public double getPerimeter();
```

Rathindra Nath Dutta

```
}
```

Rectangle.java

```
public class Rectangle implements Shape {
        private int w,h;

        public Rectangle(int w, int h) {
                this.w = w;
                this.h = h;
        }

        @Override
        public double getArea() {
                return w*h;
        }

        @Override
        public double getPerimeter() {
                return 2*(w+h);
        }
}
```

Circle.java

```
public class Circle implements Shape {
    private int r;

    public Circle(int r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return Math.PI*r*r;
    }

    @Override
    public double getPerimeter() {
```

Rathindra Nath Dutta

```
            return 2*Math.PI*r;
        }
}
```

Listing A.1: Interface

A.2 Inner Class

In Java it is possible to define a class nested within any other block. The scope of the nested class is limited by scope of of the enclosing block. Such a nested class can be declared directly within the enclosing braces of a class. These type of nested class can access members(even private ones) of the outer class. However, outer class cannot directly access member of the nested class. A nested class can be declared either as static or non-static. A static nested class (declared with the `static` keyword) requires an instance of the outer class to access its members. A non-static nested class is called an ***inner*** class. It can directly access any non-static member of its enclosing class. Followings are compelling reasons for using a nested classes.

- **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such ”helper classes” makes their package more streamlined.
- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A’s members can be declared private and B can still access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

SortedLinkedList.java

```

public class SortedLinkedList {
    private class Node { //inner class
        int data;
        Node next;
```

Rathindra Nath Dutta

```java

        public Node(int data) {
            this.data = data;
            next = null;
        }
    }

    private Node root;

    public SortedLinkedList() {
        root = null;
    }

    public void insert(int value) {
        //scan and insert a new node
    }

    public int search(int value) {
        //search and return position, -1 if not found
    }

    public void delete(int value) {
        //search and delete the node
    }
}
```

Listing A.2: Using an Inner Class in Java

A.2.1 Anonymous Inner Class

An ***anonymous inner class*** is a special type of inner class that has no name assigned to it. It is mainly used to easily generate instances for some interface. To be clear, an interface cannot be instantiated and these instances are of a class that implements the interface. Syntax for an anonymous inner class looks as follows.

```java
new InterfaceName(){
    @Override
    public void aUnimplementedMethod(args){
        //method body
    }
```

Ratindra Nath Dutta

```
    //definitions of other unimplemented methods
}
```

The syntax `new InterfaceName(){...}` tells compiler that the code between the braces defines an anonymous inner class. This class implements the interface specified by `InterfaceName`, and gets automatically instantiated when this expression is evaluated.

Being an inner class, any anonymous inner class has access to all of the members of its enclosing class. Moreover, the method definition provided inside the body of the anonymous inner class is local to the instance created by the expression. This becomes very handy when using event listeners as shown earlier in section 2.5.

Rathindra Nath Dutta

Rathindra Nath Dutta

Appendix B

Mouse and Keyboard Events

Rathindra Nath Dutta

Mouse and keyboard are the two primary input devices in a system. Any good user interface must be designed is such a away that the UI is responsive to both mouse interactions and key stokes. While the mouse can be used to select, highlight, move various elements on the UI, a keyboard can be used to type in simple texts, numbers or a key combination (shortcut) to trigger certain things.

B.1 Handling Mouse Events

The `MouseEvent` class, which extends the `InputEvent` class, defines eight types of mouse events: click, drag, enter, exit, move, press, release, and wheel. Any instance of this class provides the following methods.

```
int getX() //returns the x-coordinate of the mouse
int getY() //returns the y-coordinate of the mouse
Point getPoint() //returns current location the mouse as a point
 ↪  instance
void translatePoint(int tx, int ty) //changes the location of the
 ↪  event
int getClickCount() //returns the number of mouse clicks
 ↪  associated with this event
boolean isPopupTrigger() //tests whether the event causes a pop-up
 ↪  menu to appear
Point getLocationOnScreen() //returns the absolute position of the
 ↪  event on the entire screen
int getXOnScreen() //returns the absolute x-coordinate of the
 ↪  event on the entire screen
int getYOnScreen() //returns the absolute y-coordinate of the
 ↪  event on the entire screen
int getButton() //returns the button number
```

For a three button mouse the returned button numbers range from 0 to 3, where 0 or `NOBUTTON` means no button was pressed or released, 1 or `BUTTON1` means left mouse button, 2 or `BUTTON2` means middle mouse button, and 3 or `BUTTON3` means right mouse button. For mouse with five buttons the return value ranges from 0 to 5. The `SwingUtilities` class supplies three methods: `boolean isLeftMouseButton()`, `boolean isMiddleMouseButton()`, `boolean isRightMouseButton()` to easily determine which of the Mouse buttons is pressed.

B.1.1 Listening Mouse Events

The `MouseListener` interface listens to five mouse events defined in `MouseEvent` class, and it defines five methods corresponding to these events.

`MOUSE_PRESSED` event occurs when the mouse button is pressed. Corresponding method is `void mousePressed(MouseEvent e)`

`MOUSE_RELEASED` event occurs when the mouse button is released. Corresponding method is `void mouseReleased(MouseEvent e)`

`MOUSE_CLICKED` event occurs when the mouse button is pressed and released at the same point. Corresponding method is `void mouseClicked(MouseEvent e)`

`MOUSE_ENTERED` event occurs when the mouse pointer enters a component. Corresponding method is `void mouseEntered(MouseEvent e)`

`MOUSE_EXITED` event occurs when the mouse pointer exits a component. Corresponding method is `void mouseExited(MouseEvent e)`

The `MouseMotionListener` interface listens to two more mouse events.

`MOUSE_MOVED` event occurs when the mouse pointer is moved on a component. The corresponding method `void mouseMoved(MouseEvent e)` gets multiple times as the mouse moves.

`MOUSE_DRAGGED` event occurs when a mouse button is pressed on a component and then dragged (mouse moved while pressing the mouse button) on the screen. The corresponding method `void mouseDragged(MouseEvent e)` gets multiple times as the mouse moves.

The `MouseWheelEvent` class extends the `MouseEvent` class. If a mouse do have the wheels this class is used to handle the following event.

`MOUSE_WHEEL` event occurs when the mouse wheel is moved. The corresponding method is `void mouseWheelMoved(MouseWheelEvent e)`.

The `MouseWheelEvent` class further divides this event into two sub types.

Rathindra Nath Dutta

`WHEEL_BLOCK_SCROLL`: a page-up/page-down scroll event

`WHEEL_UNIT_SCROLL`: a line-up/line-down scroll event

It also defines some new methods, some of them are

```
int    getScrollType() //returns either WHEEL_BLOCK_SCROLL or
 ↪ WHEEL_UNIT_SCROLL
int    getWheelRotation() //returns the number of "clicks" the
 ↪ mouse wheel was rotated, as an integer. The value if positive
 ↪ for counterclockwise rotation (down scroll), and negative for
 ↪ clockwise rotation (up scroll)
double getPreciseWheelRotation() //returns the number of "clicks"
 ↪ the mouse wheel was rotated, as a double. It works same as the
 ↪ above, used for high resolution mouse wheels
```

B.2 Handling Key Events

The class `KeyEvent` covers the various key events that occurs due to some keyboard input. The `KeyEvent` is a subclass of `InputEvent` and defines various integer constants, called *key codes*: `VK_0` through `VK_9`, `VK_A` through `VK_Z`, `VK_ALT`, `VK_CONTROL`, `VK_ENTER`, `VK_ESCAPE`, `VK_SHIFT`, `VK_UP`, `VK_DOWN`, `VK_LEFT`, `VK_RIGHT`, `VK_PAGE_UP`, `VK_PAGE_DOWN` etc. They corresponds to the keys as per their names. The prefix **VK** means *virtual keys*, and these constants are independent of any modifiers such as control, shift, or alt. The `KeyEvent` defines several methods, of which the following two are most common.

```
char getKeyChar() //returns the character equivalent of the key
 ↪ that was entered
int getKeyCode() //returns the key code
```

B.2.1 Listening to Key Events

There are three types of key events and their corresponding methods are defined in the `KeyListener` interface.

`KEY_PRESSED` event occurs whenever a key is pressed. This event invokes the `void keyPressed(KeyEvent e)` method.

`KEY_RELEASED` event occurs whenever a key is released. This event invokes the `void keyReleased(KeyEvent e)` method.

`KEY_TYPED` event occurs whenever a key is pressed and released. This event invokes the `void keyTyped(KeyEvent e)` method.

Rathindra Nath Dutta

The `keyTyped()` method is invoked only if a character is entered. When a user presses and releases a key, three events are generated in this sequence: `KEY_PRESSED`, `KEY_TYPED`, `KEY_RELEASED`. When a user presses one of the special keys like *HOME* key, then only `KEY_PRESSED` and `KEY_RELEASED` are generated.

Rathindra Nath Dutta

Bibliography

[Schildt, 2011] Schildt, H. (2011). *Java The Complete Reference, 8th Edition*. The Complete Reference. Mcgraw-hill.

Rathindra Nath Dutta