

## CHAPTER 6



### Understanding Nodes

In this chapter, you will learn:

- What a node is in JavaFX
- About the Cartesian coordinate system
- About the bounds and bounding box of nodes
- How to set the size of a node and how to position a node
- How to store user data in a node
- What a managed node is
- How to transform node's bounds between coordinate spaces

#### What Is a Node?

Chapter 5 introduced you to scenes and scene graphs. A scene graph is a tree data structure. Every item in a scene graph is called a *node*. An instance of the `javafx.scene.Node` class represents a node in the scene graph. Note that the `Node` class is an abstract class, and several concrete classes exist to represent specific type of nodes.

A node can have subitems (also called children), and these node are called branch nodes. A branch node is an instance of the `Parent`, whose concrete subclasses are `Group`, `Region`, and `WebView`. A node that does cannot have subitems is called a *leaf node*. Instances of classes such as `Rectangle`, `Text`, `ImageView`, and `MediaView` are examples of leaf nodes. Only a single node within each scene graph tree will have no parent, which is referred to as the *root node*. A node may occur at the most once anywhere in the scene graph.

A node may be created and modified on any thread if it is not yet attached to a scene. Attaching a node to a scene and subsequent modification must occur on the JavaFX Application Thread.

A node has several types of bounds. Bounds are determined with respect to different coordinate systems. The next section will discuss the Cartesian coordinate system in general; the following section explains how Cartesian coordinate systems are used to compute the bounds of a node in JavaFX.

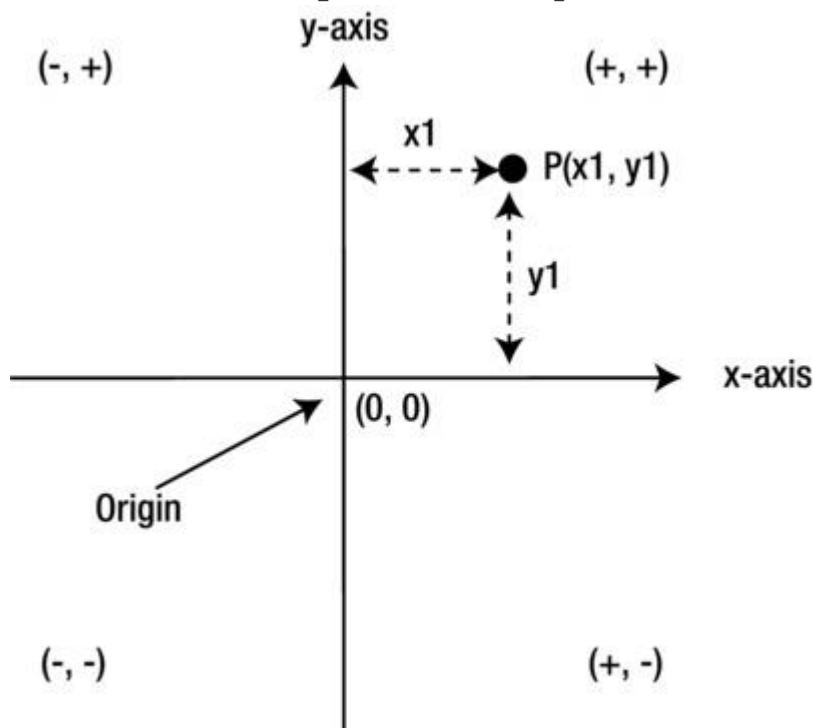
#### The Cartesian Coordinate System

If you have studied (and still remember) the Cartesian coordinate system from your coordinate geometry class in high school, you may skip this section.

The Cartesian coordinate system is a way to define each point on a 2D plane uniquely. Sometimes it is also known as a *rectangular coordinate system*. It consists of two perpendicular, directed lines known as the x axis and the y axis. The point where the two axes intersect is known as the *origin*.

A point in a 2D plane is defined using two values known as its x and y coordinates. The x and y coordinates of a point are its perpendicular distances from the y axis and x axis, respectively. Along an axis, the distance is measured as positive on one side from the origin and as negative on the other side. The origin has (x, y) coordinates, such as (0, 0). The axes divide the plane into four quadrants. Note that the 2D plane itself is infinite and so are the four quadrants. The set of all points in a Cartesian coordinate system defines the *coordinate space* of that system.

Figure 6-1 shows an illustration of the Cartesian coordinate system. It shows a point P having x and y coordinates of  $x_1$  and  $y_1$ . It shows the type of values for the x and y coordinates in each quadrant. For example, the upper right quadrant shows (+, +), meaning that both x and y coordinates for all points in this quadrant will have positive values.



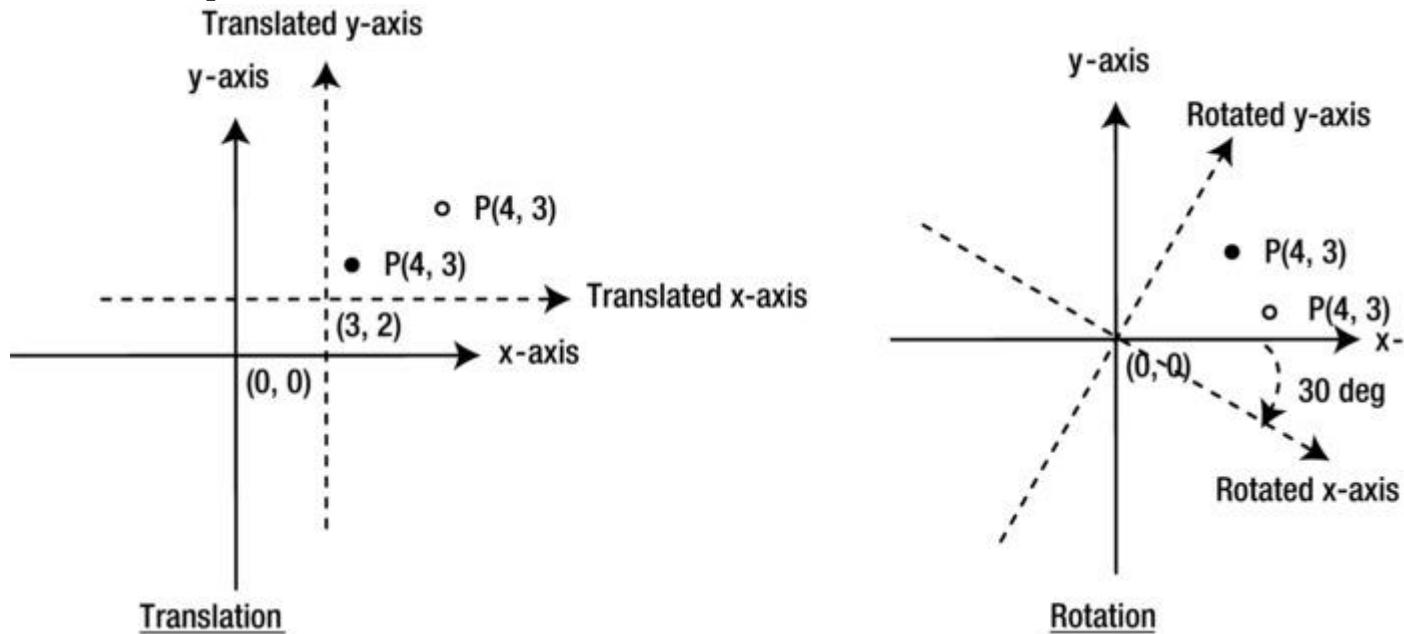
**Figure 6-1.** A two-dimensional Cartesian coordinate system used in coordinate geometry

A transformation is a mapping of points in a coordinate space to themselves, preserving distances and directions between

them. Several types of transformations can be applied to points in a coordinate space. Some examples of transformation types are *translation*, *rotation*, *scaling*, and *shearing*.

In a translation transformation, a fixed pair of numbers is added to the coordinates of all points. Suppose you want to apply translation to a coordinate space by  $(a, b)$ . If a point had coordinates  $(x, y)$  before translation, it will have the coordinate of  $(x + a, y + b)$  after translation.

In a rotation transformation, the axes are rotated around a pivot point in the coordinate space and the coordinates of points are mapped to the new axes. Figure 6-2 shows examples of translation and rotation transformations.



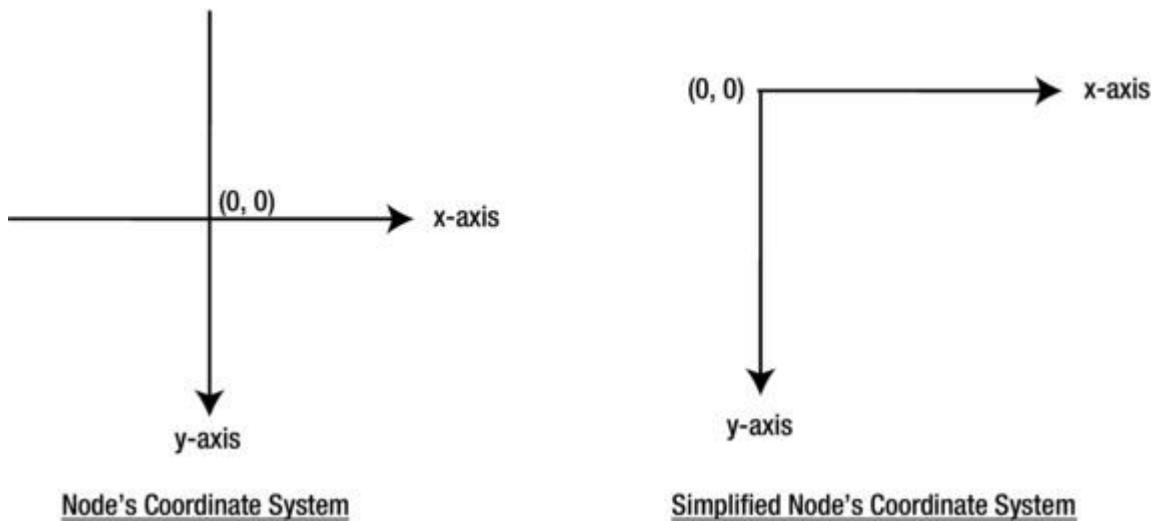
**Figure 6-2.** Examples of translation and rotation transformations

In Figure 6-2, axes before the transformations are shown in solid lines, and axes after the transformations are shown in dashed lines. Note that the coordinates of the point  $P$  at  $(4, 3)$  remains the same in the translated and rotated coordinate spaces.

However, the coordinates of the point relative to the original coordinate space change after the transformation. The point in the original coordinate space is shown in a solid black fill color, and in the transformed coordinate space, it is shown without a fill color. In the rotation transformation, you have used the origin as the pivot point. Therefore, the origins for the original and the transformed coordinate space are the same.

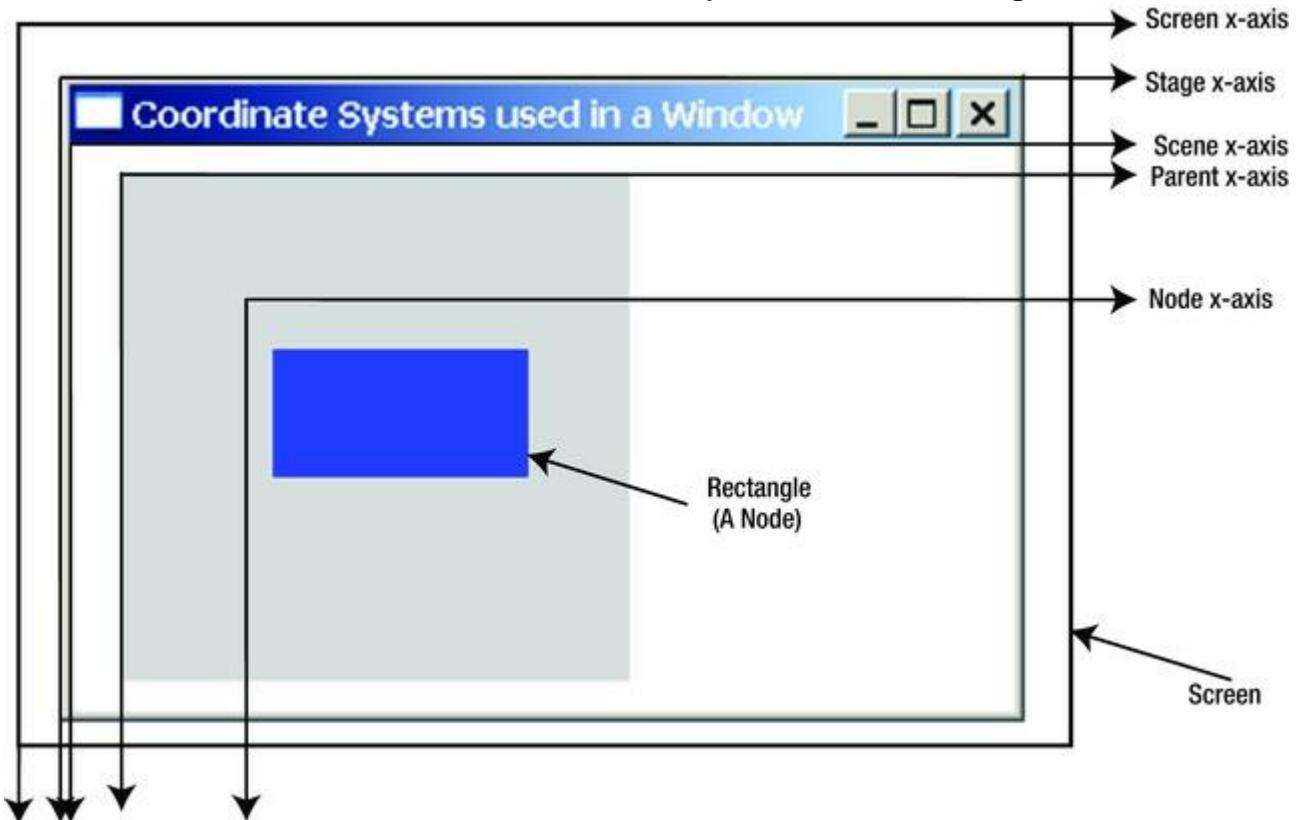
### Cartesian Coordinate System of a Node

Each node in a scene graph has its own coordinate system. A node uses a Cartesian coordinate system that consists of an x axis and a y axis. In computer systems, the values on the x axis increase to the right and the values on y axis increase downward, as shown in Figure 6-3. Typically, when showing the coordinate system of nodes, the negative sides of the x axis and y axis are not shown, even though they always exist. The simplified version of the coordinate system is shown on the right part of Figure 6-3. A node can have negative x and y coordinates.



**Figure 6-3.** The coordinate system of nodes

In a typical GUI application, nodes are placed within their parents. A root node is the ultimate parent of all nodes and it is placed inside a scene. The scene is placed inside a stage and the stage is placed inside a screen. Each element comprising a window, from nodes to the screen, has its own coordinate system, as shown in Figure 6-4.



**Figure 6-4.** Coordinate systems of all elements comprising a GUI window

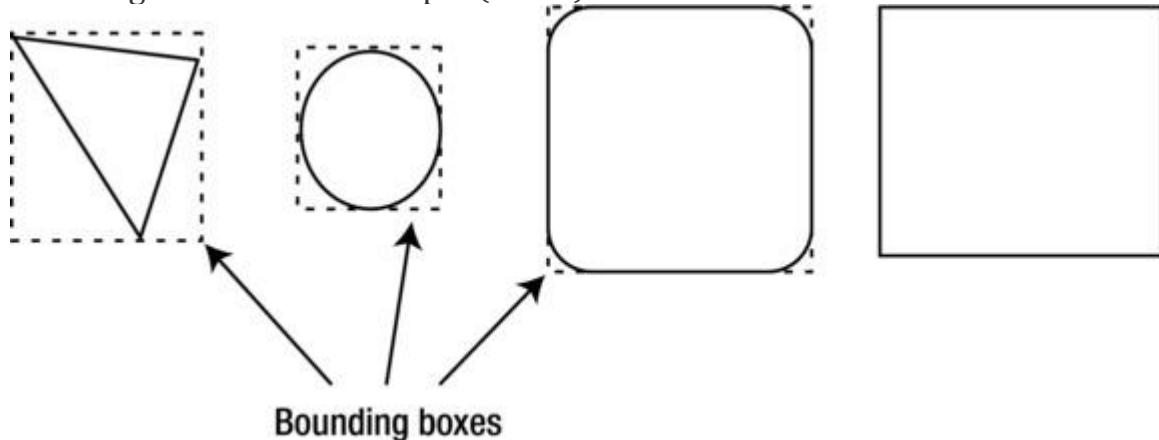
The outermost rectangular area with a thick black border is the screen. The rest is a JavaFX stage with a region and a rectangle. The region has a light-gray background color and a rectangle has a blue background color. The region is the parent of the rectangle. This simple window uses five coordinate spaces as indicated in Figure 6-4.

I have labeled only the x axes. All y axes are vertical lines meeting the respective x axes at their origins.

What are the coordinates of the upper left corner of the rectangle? The question is incomplete. The coordinates of a point are defined relative to a coordinate system. As shown in Figure 6-4, you have five coordinate systems at play, and hence, five coordinate spaces. Therefore, you must specify the coordinate system in which you want to know the coordinates of the upper left corner of the rectangle. In a node's coordinate system, they are (10, 15); in a parent's coordinate system, they are (40, 45); in a scene's coordinate system, they are (60, 55); in a stage's coordinate system, they are (64, 83); in a screen's coordinate system, they are (80, 99).

## The Concept of Bounds and Bounding Box

Every node has a geometric shape and it is positioned in a coordinate space. The size and the position of a node are collectively known as its *bounds*. The bounds of a node are defined in terms of a bounding rectangular box that encloses the entire geometry of the node. Figure 6-5 shows a triangle, a circle, a rounded rectangle, and a rectangle with a solid border. Rectangles around them, shown with a dashed border, are the bounding boxes for those shapes (nodes).

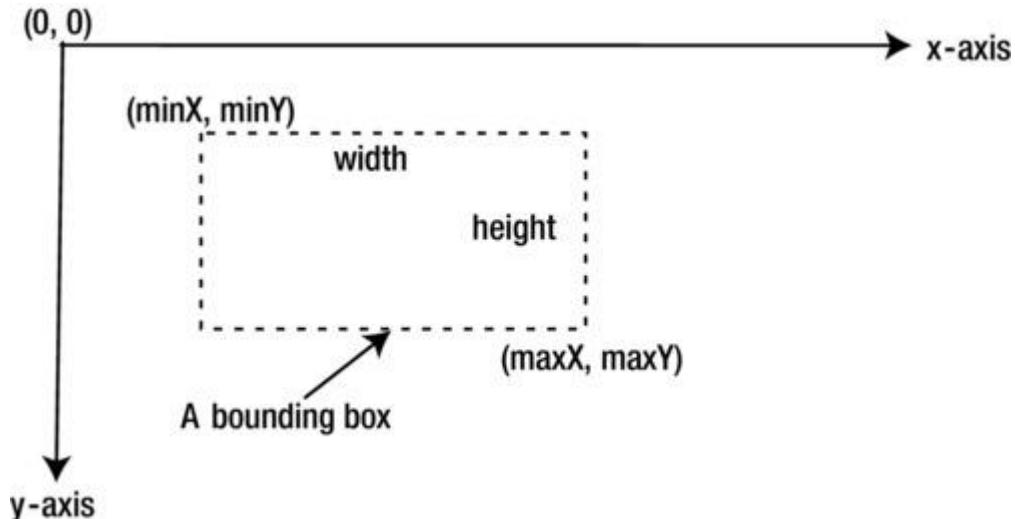


**Figure 6-5.** The bounding rectangular box defining the geometric shape of nodes

The area (area in a 2D space and volume in a 3D space) covered by the geometric shape of a node and its bounding box may be different. For example, for the first three nodes in Figure 6-5, counting from the left, the areas of the nodes and their bounding boxes are different. However, for the last rectangle, without rounded corners, its area and that of its bounding box are the same.

An instance of the `javafx.geometry.Bounds` class represents the bounds of a node. The `Bounds` class is an abstract class. The `BoundingBox` class is a concrete implementation of the `Bounds` class. The `Bounds` class is designed to handle bounds in a 3D space. It encapsulates the coordinates of the upper left corner with the minimum depth in the bounding box and the width, height, and depth of the bounding box. The methods `getMinX()`, `getMinY()`, and `getMinZ()` are used to get the coordinates. The three dimensions of the bounding box are accessed using the `getWidth()`, `getHeight()`, and `getDepth()` methods. The `Bounds` class contains the `getMaxX()`, `getMaxY()`, and `getMaxZ()` methods that return the coordinates of the lower right corner, with the maximum depth, in the bounding box.

In a 2D space, the `minX` and `minY` define the x and y coordinates of the upper left corner of the bounding box, respectively, and the `maxX` and `maxY` define the x and y coordinates of the lower right corner, respectively. In a 2D space, the values of the z coordinate and the depth for a bounding box are zero. Figure 6-6 shows the details of a bounding box in a 2D coordinate space.



**Figure 6-6.** The makings of a bounding box in a 2D space

The `Bounds` class contains `isEmpty()`, `contains()`, and `intersects()` utility methods. The `isEmpty()` method returns true if any of the three dimensions (width, height, or depth) of a `Bounds` is negative. The `contains()` method lets you check if a `Bounds` contains another `Bounds`, a 2D point, or a 3D point. The `intersects()` method lets you check if the interior of a `Bounds` intersects the interior of another `Bounds`, a 2D point, or a 3D point.

### Knowing the Bounds of a Node

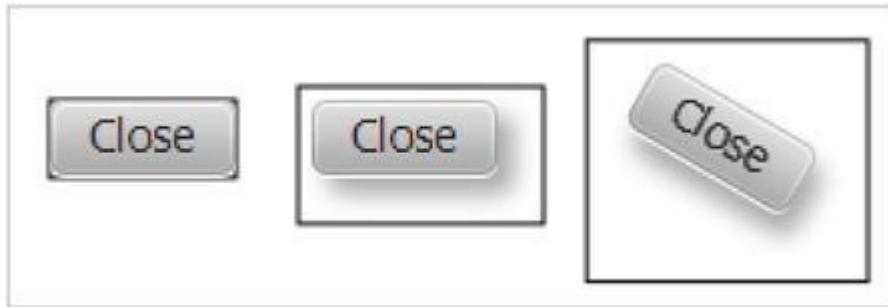
So far, I have covered topics such as coordinate systems, bounds, and bounding boxes related to a node. That discussion was to prepare you for this section, which is about knowing the bounds of a node. You might have guessed (though incorrectly) that the `Node` class should have a `getBounds()` method to return the bounds of a node. It would be great if it were that simple! In this section, I will discuss the details of different types of bounds of a node. In the next section, I will walk you through some examples.

Figure 6-7 shows a button with the text “Close” in three forms.



**Figure 6-7.** A button with and without an effect and a transformation

The first one, starting from the left, has no effects or transformations. The second one has a drop shadow effect. The third one has a drop shadow effect and a rotation transformation. Figure 6-8 shows the bounding boxes representing the bounds of the button in those three forms. Ignoring the coordinates for now, you may notice that the bounds of the button change as effects and transformations are applied.



**Figure 6-8.** A button with and with an effect and a transformation with bounding boxes

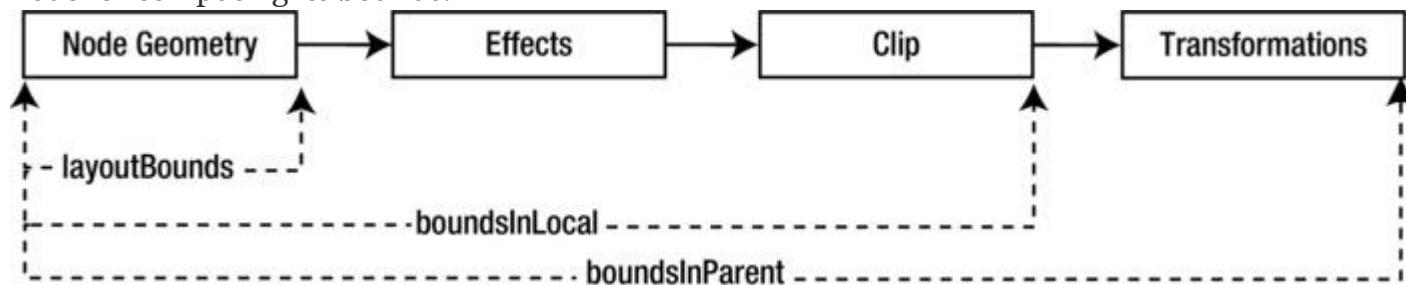
A node in a scene graph has three types of bounds defined as three read-only properties in the `Node` class:

- `layoutBounds`
- `boundsInLocal`
- `boundsInParent`

When you are trying to understand the three types of the bounds of a node, you need to look for three points:

- How the  $(\min X, \min Y)$  values are defined. They define the coordinates of the upper left corner of the bounding box described by the `Bounds` object.
- Remember that coordinates of a point are always defined relative to a coordinate space. Therefore, pay attention to the coordinate space in which the coordinates, as described in the first step, are defined.
- What properties of the node—geometry, stroke, effects, clip, and transformations—are included in a particular type of bounds.

Figure 6-9 shows the properties of a node contributing to the bounds of a node. They are applied from left to right in order. Some node types (e.g., `Circle`, `Rectangle`) may have a nonzero stroke. A nonzero stroke is considered part of the geometry of a node for computing its bounds.



**Figure 6-9.** Factors contributing to the size of a node

Table 6-1 lists the properties that contribute to a particular type of the bounds of a node and the coordinate space in which the bounds are defined.

The `boundsInLocal` and `boundsInParent` of a node are also known as its *physical bounds* as they correspond to the physical properties of the node.

The `layoutBounds` of a node is known as the *logical bounds* as it is not necessarily tied to the physical bounds of the node. When the geometry of a node is changed, all bounds are recomputed.

**Table 6-1.** Contributing Properties to the Bounds of a Node

Bounds Type	Coordinate Space	Contributors
<code>layoutBounds</code>	Node (Untransformed)	Geometry of the node Nonzero stroke
<code>boundsInLocal</code>	Node (Untransformed)	Geometry of the node Nonzero stroke Effects Clip
<code>boundsInParent</code>	Parent	Geometry of the node Nonzero stroke Effects Clip Transformations

**Tip** The `boundsInLocal` and `boundsInParent` are known as physical or visual bounds as they correspond to how the node looks visually.

The `layoutBounds` is also known as the *logical bounds* as it does not necessarily correspond to the physical bounds of the node.

### The `layoutBounds` Property

The `layoutBounds` property is computed based on the geometric properties of the node in the *untransformed* local coordinate space of the node. Effects, clip, and transformations are not included. Different rules, depending on the resizable behavior of the node, are used to compute the coordinates of the upper left corner of the bounding box described by the `layoutBounds`:

- For a resizable node (a `Region`, a `Control`, and a `WebView`), the coordinates for the upper left corner of the bounding box are always set to (0, 0). For example, the (`minX`, `minY`) values in the `layoutBounds` property are always (0, 0) for a button.

- For a nonresizable node (a Shape, a Text, and a Group), the coordinates of the upper left corner of the bounding box are computed based on the geometric properties. For a shape (a rectangle, a circle, etc.) or a Text, you can specify the (x, y) coordinates of a specific point in the node relative to the untransformed coordinate space of the node. For example, for a rectangle, you can specify the (x, y) coordinates of the upper left corner, which become the (x, y) coordinates of the upper left corner of the bounding box described by its `layoutBounds` property. For a circle, you can specify the `centerX`, `centerY`, and `radius` properties, where `centerX` and `centerY` are the x and y coordinates of the center of the circle, respectively. The (x, y) coordinates of the upper left corner of the bounding box described by the `layoutBounds` for a circle are computed as (`centerX - radius`, `centerY - radius`).

The width and height in `layoutBounds` are the width and height of the node. Some nodes let you set their width and height; but some compute them automatically for you and let you override them.

Where do you use the `layoutBounds` property of a node? Containers allocate spaces to lay out child nodes based on their `layoutBounds`. Let's look at an example as shown in Listing 6-1. It displays four buttons in a `VBox`. The first button has a drop shadow effect. The third button has a drop shadow effect and a 30-degree rotation transformation. The second and the fourth buttons have no effect or transformation. The resulting screen is shown in Figure 6-10. The output shows that irrespective of the effect and transformation, all buttons have the same `layoutBounds` values. The size (width and height) in the `layoutBounds` objects for all buttons is determined by the text of the button and the font, which is the same for all buttons. The output may differ on your platform.

### ***Listing 6-1.*** Accessing the `layoutBounds` of Buttons with and without Effects

```
// LayoutBoundsTest.java
package com.jdojo.node;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class LayoutBoundsTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button b1 = new Button("Close");
        b1.setEffect(new DropShadow());

        Button b2 = new Button("Close");

        Button b3 = new Button("Close");
    }
}
```

```

        b3.setEffect(new DropShadow());
        b3.setRotate(30);

        Button b4 = new Button("Close");

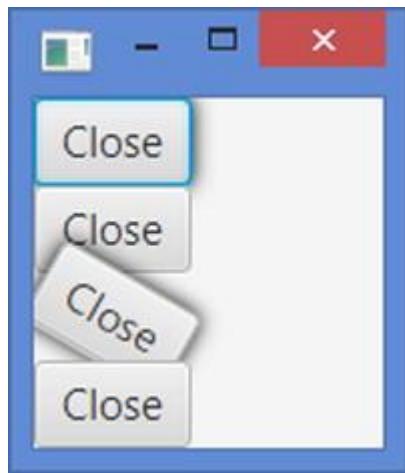
        VBox root = new VBox();
        root.getChildren().addAll(b1, b2, b3, b4);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Testing LayoutBounds");
        stage.show();

        System.out.println("b1=" + b1.getLayoutBounds());
        System.out.println("b2=" + b2.getLayoutBounds());
        System.out.println("b3=" + b3.getLayoutBounds());
        System.out.println("b4=" + b4.getLayoutBounds());
    }
}

b1=BoundingBox [minX:0.0, minY:0.0, minZ:0.0, width:57.0, height:23.0,
depth:0.0, maxX:57.0, maxY:23.0, maxZ:0.0]
b2=BoundingBox [minX:0.0, minY:0.0, minZ:0.0, width:57.0, height:23.0,
depth:0.0, maxX:57.0, maxY:23.0, maxZ:0.0]
b3=BoundingBox [minX:0.0, minY:0.0, minZ:0.0, width:57.0, height:23.0,
depth:0.0, maxX:57.0, maxY:23.0, maxZ:0.0]
b4=BoundingBox [minX:0.0, minY:0.0, minZ:0.0, width:57.0, height:23.0,
depth:0.0, maxX:57.0, maxY:23.0, maxZ:0.0]

```



**Figure 6-10.** The `layoutBounds` property does not include the effects and transformations

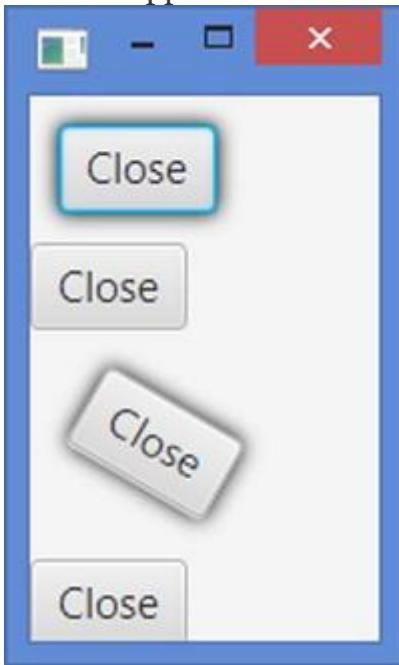
Sometimes you may want to include the space needed to show the effects and transformations of a node in its `layoutBounds`. The solution for this is easy. You need to wrap the node in a `Group` and the `Group` in a container. Now the container will query the `Group` for its `layoutBounds`. The `layoutBounds` of a `Group` is the union of the `boundsInParent` for all its children. Recall that (see Table 6-1) the `boundsInParent` of a node includes the space needed for showing effects and transformation of the node. If you change the statement

```
root.getChildren().addAll(b1, b2, b3, b4);
```

in Listing 6-1 to

```
root.getChildren().addAll(new Group(b1), b2, new Group(b3), b4);
```

the resulting screen is shown in Figure 6-11. This time, `VBox` allocated enough space for the first and the third groups to account for the effect and transformation applied to the wrapped buttons.



**Figure 6-11.** Using a `Group` to allocate space for effects and transformations of a node

**Tip** The `layoutBounds` of a node is computed based on the geometric properties of a node. Therefore, you should not bind such properties of a node to an expression that includes the `layoutBounds` of the node.

### The `boundsInLocal` Property

The `boundsInLocal` property is computed in the untransformed coordinate space of the node. It includes the geometric properties of the node, effects, and clip. Transformations applied to a node are not included.

**Listing 6-2** prints the `layoutBounds` and `boundsInLocal` of a button.

The `boundsInLocal` property includes the drop shadow effect around the button. Notice that the coordinates of the upper left corner of the bounding box defined by the `layoutBounds` are (0.0, 0.0) and they are (-9.0, -9.0) for the `boundsInLocal`. The output may be a bit different on different platforms as the size of nodes is computed automatically based on the platform running the program.

### **Listing 6-2.** Accessing the `boundsInLocal` Property of a Node

```
// BoundsInLocalTest.java
package com.jdojo.node;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BoundsInLocalTest extends Application {
```

```

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

@Override
public void start(Stage stage) {
    Button b1 = new Button("Close");
    b1.setEffect(new DropShadow());

    VBox root = new VBox();
    root.getChildren().addAll(b1);

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Testing LayoutBounds");
    stage.show();

    System.out.println("b1(layoutBounds)=" + b1.getLayoutBounds());
    System.out.println("b1(boundsInLocal)=" + b1.getBoundsInLocal());
}
}
b1(layoutBounds)=BoundingBox [minX:0.0, minY:0.0, minZ:0.0, width:57.0,
height:23.0, depth:0.0, maxX:57.0, maxY:23.0, maxZ:0.0]
b1(boundsInLocal)=BoundingBox [minX:-9.0, minY:-9.0, minZ:0.0, width:75.0,
height:42.0, depth:0.0, maxX:66.0, maxY:33.0, maxZ:0.0]

```

When do you use the `boundsInLocal` of a node? You would use `boundsInLocal` when you need to include the effects and the clip of a node. Suppose you have a `Text` node with a reflection and you want to center it vertically. If you use the `layoutBounds` of the `Text` node, it will only center the text portion of the node and would not include the reflection. If you use the `boundsInLocal`, it will center the text with its reflection. Another example would be checking for collisions of balls that have effects. If a collision between two balls occurs when one ball moves inside the bounds of another ball that include their effects, use the `boundsInLocal` for the balls. If a collision occurs only when they intersect their geometric boundaries, use the `layoutBounds`.

### The `boundsInParent` Property

The `boundsInParent` property of a node is in the coordinate space of its parent. It includes the geometric properties of the node, effects, clip, and transformations. It is rarely used directly in code.

### Bounds of a Group

The computation of `layoutBounds`, `boundsInLocal`, and `boundsInParent` for a `Group` is different from that of a node. A `Group` takes on the collection bounds of its children. You can apply effects, clip, and transformations separately on each child of a `Group`. You can also apply effects, clip, and transformations directly on a `Group` and they are applied to all of its children nodes.

The `layoutBounds` of a `Group` is the union of the `boundsInParent` of all its children. It includes effects, clip, and transformations applied directly to the children. It does not include effects, clip, and transformations applied directly to the `Group`. The `boundsInLocal` of a `Group` is computed by taking its `layoutBounds` and including the effects and clip applied directly to the `Group`.

The `boundsInParent` of a `Group` is computed by taking its `boundsInLocal` and including the transformations applied directly to the `Group`.

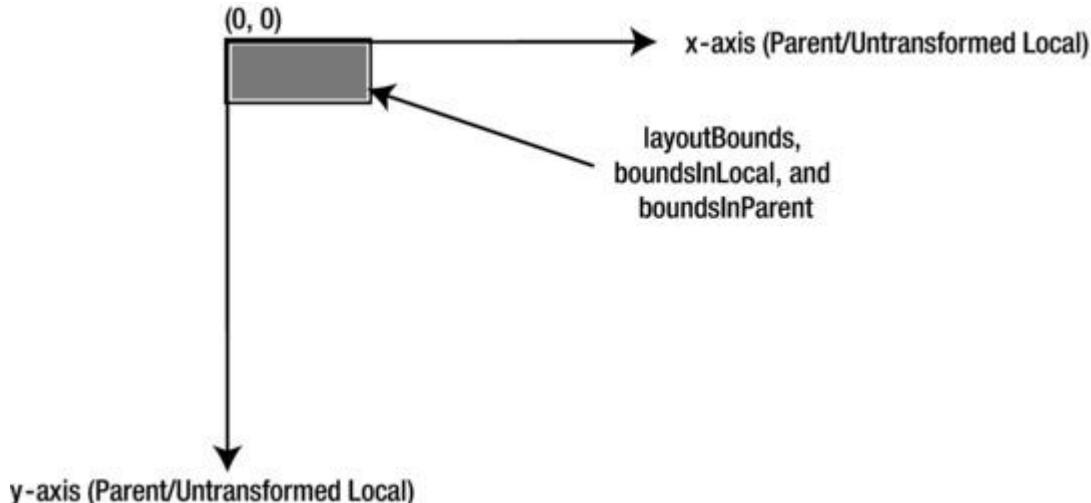
When you want to allocate space for a node that should include effects, clip, and transformations, you need to try wrapping the node in a `Group`. Suppose you have a node with effects and transformations and you only want to allocate layout space for its effects, not its transformations. You can achieve this by applying the effects on the node and wrapping it in a `Group`, and then applying the transformations on the `Group`.

### A Detailed Example on Bounds

In this section, I will walk you through an example to show how the bounds of a node are computed. You will use a rectangle and its different properties, effects, and transformations in this example.

Consider the following snippet of code that creates a 50 by 20 rectangle and places it at  $(0, 0)$  in the local coordinate space of the rectangle. The resulting rectangle is shown in Figure 6-12, which shows the axes of the parent and the untransformed local axes of the node (the rectangle in this case), which are the same at this time:

```
Rectangle r = new Rectangle(0, 0, 50, 20);
r.setFill(Color.GRAY);
```



**Figure 6-12.** A 50 by 20 rectangle placed at  $(0, 0)$  with no effects and transformations

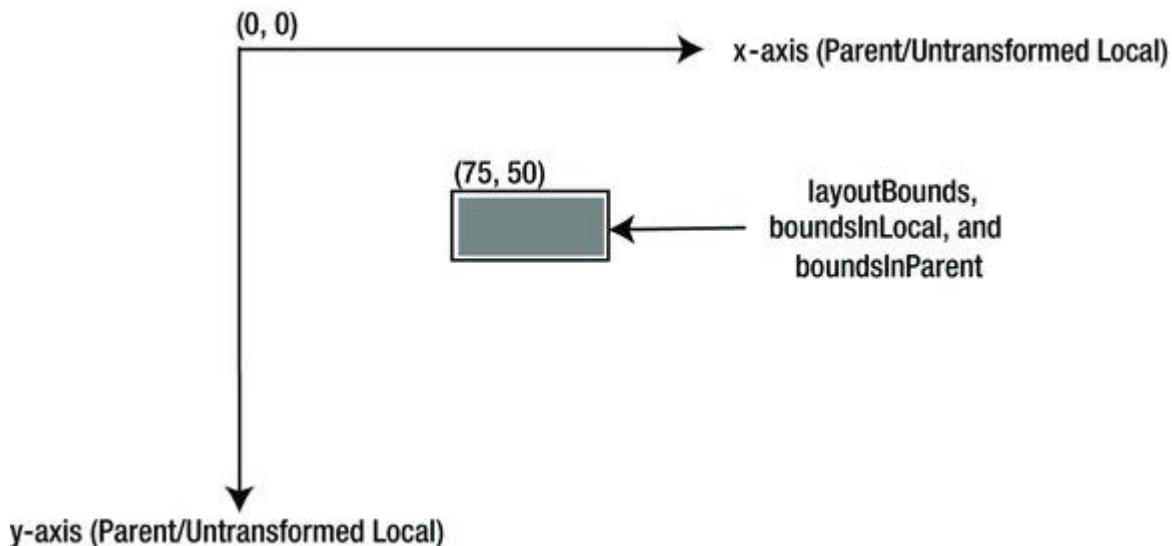
Three types of bounds of the rectangle are the same, as follows:

```
layoutBounds[minX=0.0, minY=0.0, width=50.0, height=20.0]
boundsInLocal[minX=0.0, minY=0.0, width=50.0, height=20.0]
boundsInParent[minX=0.0, minY=0.0, width=50.0, height=20.0]
```

Let's modify the rectangle to place it at  $(75, 50)$  as follows:

```
Rectangle r = new Rectangle(75, 50, 50, 20);
```

The resulting node is shown in Figure 6-13.



**Figure 6-13.** A 50 by 20 rectangle placed at  $(75, 50)$  with no effects and transformations

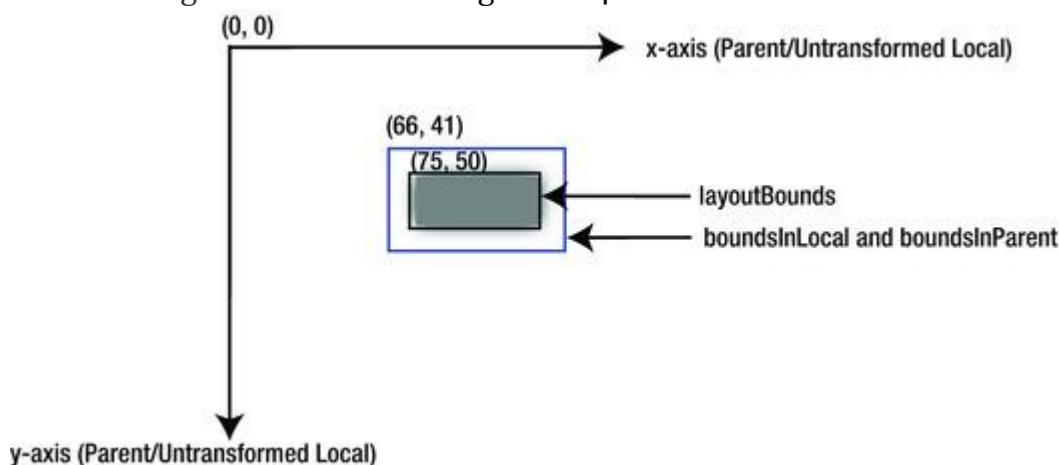
The axes for the parent and the node are still the same. All bounds are the same, as follows: The upper left corner of all bounding boxes have moved to  $(75, 50)$  with the same width and height:

```
layoutBounds[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInLocal[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInParent[minX=75.0, minY=50.0, width=50.0, height=20.0]
```

Let's modify the rectangle and give it a drop shadow effect, as follows:

```
Rectangle r = new Rectangle(75, 50, 50, 20);
r.setEffect(new DropShadow());
```

The resulting node is shown in Figure 6-14.



**Figure 6-14.** A 50 by 20 rectangle placed at  $(75, 50)$  with a drop shadow and no transformations

The axes for the parent and the node are still the same. Now, the `layoutBounds` did not change. To accommodate the drop shadow effect, the `boundsInLocal` and `boundsInParent` have changed and they have the same values. Recall that the `boundsInLocal` is defined in the untransformed coordinate space of the node and the `boundsInParent` in the coordinate space of the parent. In this case, both coordinate spaces are the same. Therefore, the same values for the two bounds define the same bounding box. The values for the bounds are as follows:

```

layoutBounds[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInLocal[minX=66.0, minY=41.0, width=68.0, height=38.0]
boundsInParent[minX=66.0, minY=41.0, width=68.0, height=38.0]

```

Let's modify the previous rectangle to have a (x, y) translation of (150, 75) as follows:

```

Rectangle r = new Rectangle(75, 50, 50, 20);
r.setEffect(new DropShadow());
r.getTransforms().add(new Translate(150, 75));

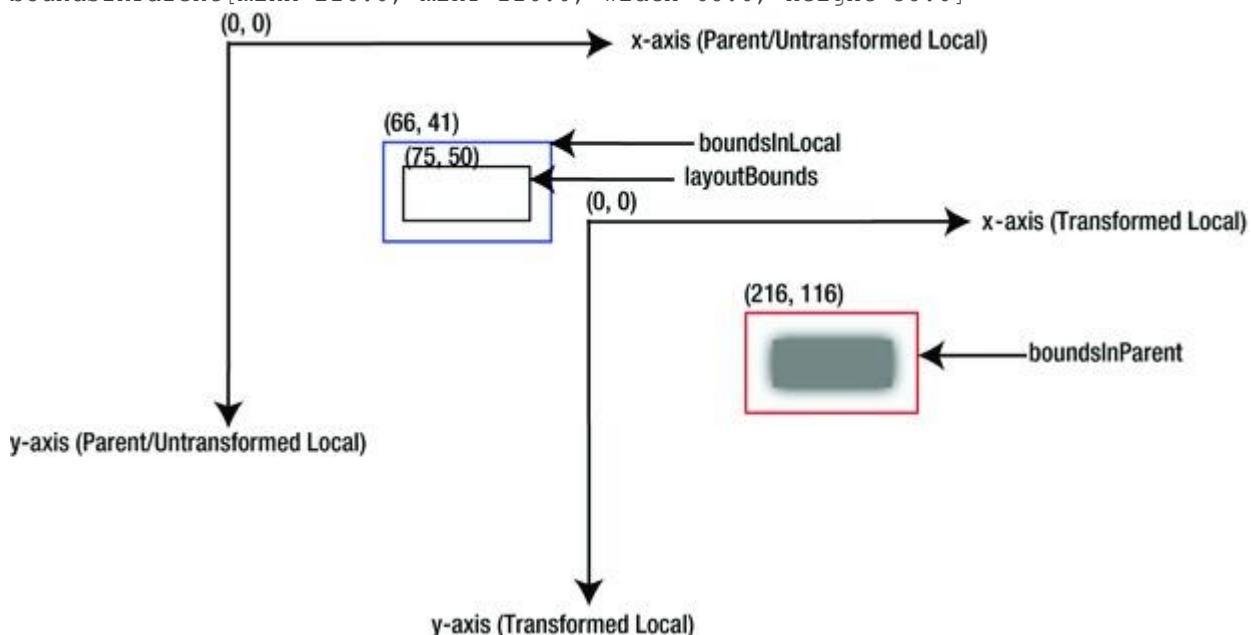
```

The resulting node is shown in Figure 6-15. A transformation (a translation, in this case) transforms the coordinate space of the node, and as a result, you see the node being transformed. In this case, you have three coordinate spaces to consider: the coordinate space of the parent, and the untransformed and transformed coordinate spaces of the node. The `layoutBounds` and `boundsInParent` are relative to the untransformed local coordinate space of the node. The `boundsInParent` is relative to the coordinate space of the parent. Figure 6-15 shows all coordinate spaces at play. The values for the bounds are as follows:

```

layoutBounds[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInLocal[minX=66.0, minY=41.0, width=68.0, height=38.0]
boundsInParent[minX=216.0, minY=116.0, width=68.0, height=38.0]

```



**Figure 6-15.** A 50 by 20 rectangle placed at (75, 50) with a drop shadow and a (150, 75) translation

Let's modify the rectangle to have a (x, y) translation of (150, 75) and a 30-degree clockwise rotation:

```

Rectangle r = new Rectangle(75, 50, 50, 20);
r.setEffect(new DropShadow());
r.getTransforms().addAll(new Translate(150, 75), new Rotate(30));

```

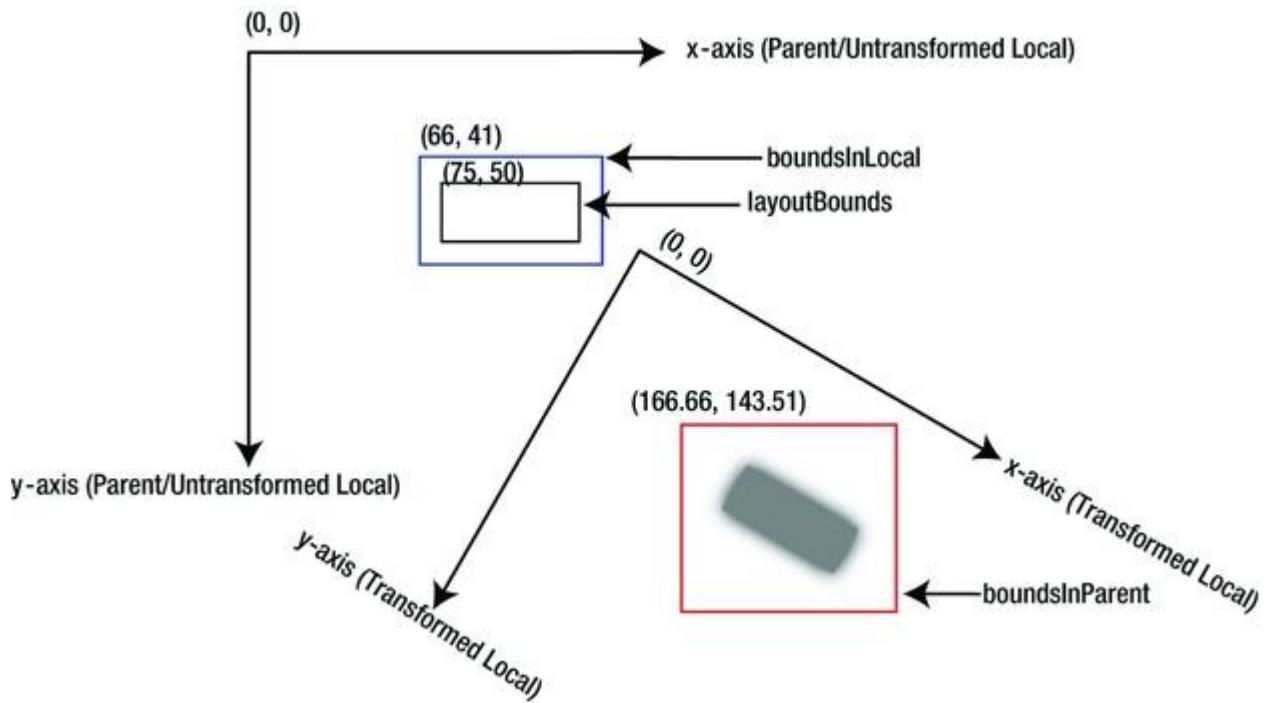
The resulting node is shown in Figure 6-16. Notice that the translation and rotation have been applied to the local coordinate space of the rectangle and the rectangle appears in the same position relative to its transformed local coordinate axes.

The `layoutBounds` and `boundsInLocal` remained the same because you did not change the geometry of the rectangle and the effects. The `boundsInParent` has changed because you added a rotation. The values for the bounds are as follows:

```

layoutBounds[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInLocal[minX=66.0, minY=41.0, width=68.0, height=38.0]
boundsInParent[minX=167.66, minY=143.51, width=77.89, height=66.91]

```

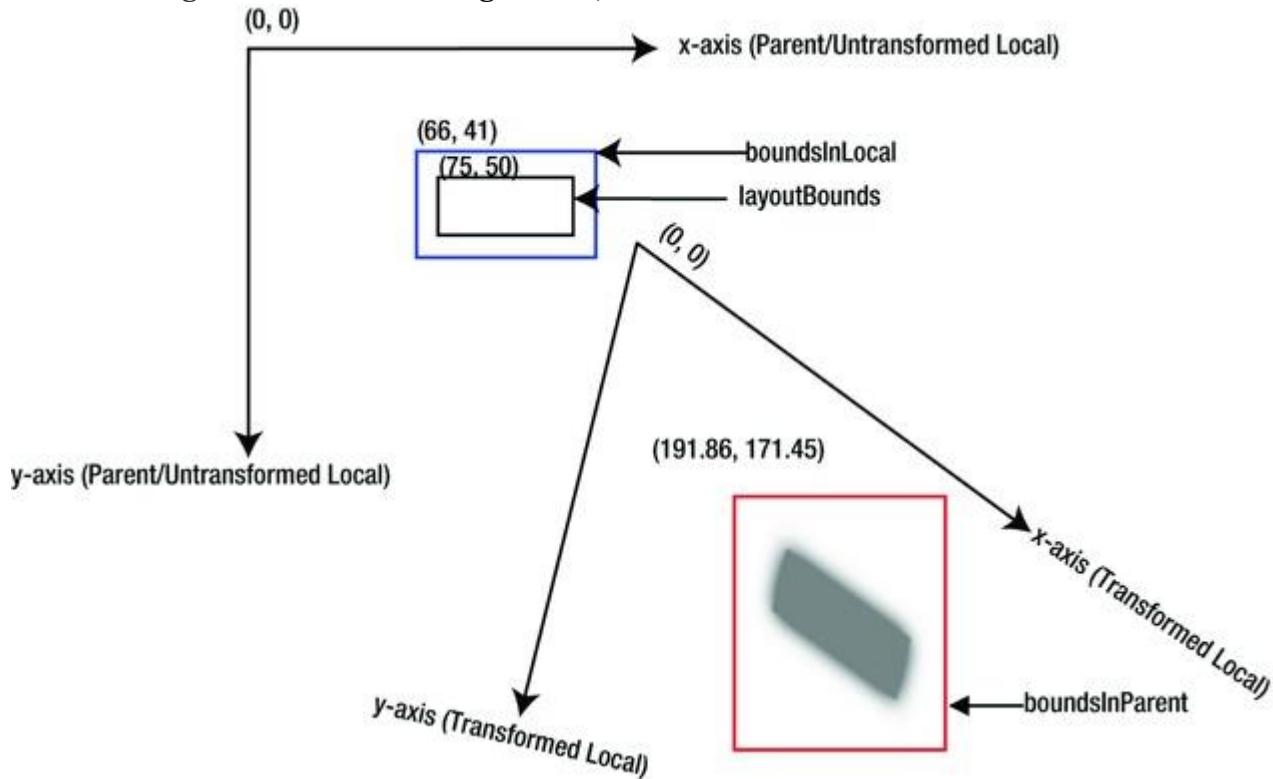


**Figure 6-16.** A 50 by 20 rectangle placed at (75, 50) with a drop shadow, a (150, 75) translation, and a 30-degree clockwise rotation

As the last example, you will add scale and shear transformations to the rectangle:

```
Rectangle r = new Rectangle(75, 50, 50, 20);
r.setEffect(new DropShadow());
r.getTransforms().addAll(new Translate(150, 75), new Rotate(30),
    new Scale(1.2, 1.2), new Shear(0.30, 0.10));
```

The resulting node is shown in Figure 6-17.



**Figure 6-17.** A 50 by 20 rectangle placed at (75, 50) with a drop shadow, a (150, 75) translation, a 30-degree clockwise rotation, a 1.2 in x and y scales, and a 0.30 x shear and 0.10 y shear

Notice that only `boundsInParent` has changed. The values for the bounds are as follows:

```
layoutBounds[minX=75.0, minY=50.0, width=50.0, height=20.0]
boundsInLocal[minX=66.0, minY=41.0, width=68.0, height=38.0]
boundsInParent[minX=191.86, minY=171.45, width=77.54, height=94.20]
```

For a beginner, it is not easy to grasp the concepts behind different types of bounds of a node. A beginner is one who is learning something for the first time. I started out as a beginner while learning about bounds. During the learning process, another beautiful concept and hence its implementation in a JavaFX program came about. The program, which is a very detailed demo application, helps you understand visually how bounds are affected by changing the state of a node. You can save the scene graph with all coordinate axes. You can run the `NodeBoundsApp` class as shown in Listing 6-3 to see all the examples in this section in action.

### **Listing 6-3.** Computing the Bounds of a Node

```
// NodeBoundsApp.java
package com.jdojo.node;
...
public class NodeBoundsApp extends Application {
    // The code for this class is not included here as it is very big.
    // Please refer to the source code. You can download the source code
    // for all programs in this book from http://www.apress.com/source-code
}
```

### Positioning a Node Using `layoutX` and `layoutY`

If you do not understand the details and the reasons behind the existence of all layout-related properties, laying out nodes in JavaFX is as confusing as it can get. The `Node` class has two properties, `layoutX` and `layoutY`, to define translation of its coordinate space along the x axis and y axis, respectively. The `Node` class has `translateX` and `translateY` properties that do the same thing. The final translation of the coordinate space of a node is the sum of the two:

```
finalTranslationX = layoutX + translateX
finalTranslationY = layoutY + translateY
```

Why do you have two properties to define translations of the same kind? The reason is simple. They exist to achieve the similar results in different situations.

Use `layoutX` and `layoutY` to position a node for a stable layout.

Use `translateX` and `translateY` to position a node for a dynamic layout, for example, during animation.

It is important to keep in mind that the `layoutX` and `layoutY` properties do not specify the final position of a node. They are translations applied to the *coordinate space* of the node. You need to factor the `minX` and `minY` values of the `layoutBounds` when you compute the value of `layoutX` and `layoutY` to position a node at a particular position. To position the upper left corner of the bounding box of a node at `finalX` and `finalY`, use the following formula:

```
layoutX = finalX - node.getLayoutBounds().getMinX()
layoutY = finalY - node.getLayoutBounds().getMinY()
```

**Tip** The `Node` class has a convenience method, `relocate(double finalX, double finalY)`, to position the node at the `(finalX, finalY)` location. The

method computes and sets the `layoutX` and `layoutY` values correctly, taking into account the `minX` and `minY` values of the `layoutBounds`. To avoid errors and misplacement of nodes, I prefer using the `relocate()` method over the `setLayoutX()` and `setLayoutY()` methods.

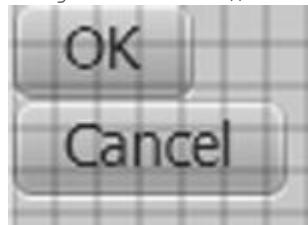
Sometimes setting the `layoutX` and `layoutY` properties of a node may not position them at the desired location inside its parent. If you are caught in this situation, check the parent type. Most parents, which are the subclasses of the `Region` class, use their own positioning policy, ignoring the `layoutX` and `layoutY` settings of their children. For example, `HBox` and `VBox` use their own positioning policy and they will ignore the `layoutX` and `layoutY` values for their children.

The following snippet of code will ignore the `layoutX` and `layoutY` values for two buttons, as they are placed inside a `VBox` that uses its own positioning policy. The resulting layout is shown in Figure 6-18.

```
Button b1 = new Button("OK");
b1.setLayoutX(20);
b1.setLayoutY(20);

Button b2 = new Button("Cancel");
b2.setLayoutX(50);
b2.setLayoutY(50);

VBox vb = new VBox();
vb.getChildren().addAll(b1, b2);
```



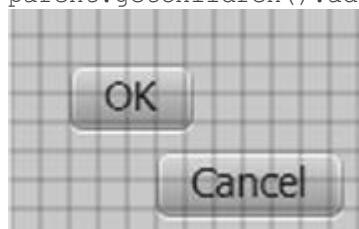
**Figure 6-18.** Two buttons using `layoutX` and `layoutY` properties and placed inside a `VBox`

If you want to have full control on positioning a node within its parent, use a `Pane` or a `Group`. A `Pane` is a `Region`, which does not position its children. You will need to position the children using the `layoutX` and `layoutY` properties. The following snippet of code will lay out two buttons as shown in Figure 6-19, which shows the coordinate grid in which lines are placed 10px apart.

```
Button b1 = new Button("OK");
b1.setLayoutX(20);
b1.setLayoutY(20);

Button b2 = new Button("Cancel");
b2.setLayoutX(50);
b2.setLayoutY(50);

Group parent = new Group(); //Or. Pane parent = new Pane();
parent.getChildren().addAll(b1, b2);
```



**Figure 6-19.** Two buttons using `layoutX` and `layoutY` properties and placed inside a `Group` or a `Pane`

## Setting the Size of a Node

Every node has a size (width and height), which may be changed. That is, every node can be resized. There are two types of nodes: *resizable* nodes and *nonresizable* nodes. Aren't the previous two sentences contradictory? The answer is yes and no. It is true that every node has the potential to be resized. However, by a resizable node, it is meant that a node can be resized by its parent during layout. For example, a button is a resizable node and a rectangle is a nonresizable node. When a button is placed in a container, for example, in an `HBox`, the `HBox` determines the best size for the button. The `HBox` resizes the button depending on how much space is needed for the button to display and how much space is available to the `HBox`. When a rectangle is placed in an `HBox`, the `HBox` does not determine its size; rather, it uses the size of the rectangle specified by the application.

**Tip** A resizable node can be resized by its parent during a layout. A nonresizable node is not resized by its parent during a layout. If you want to resize a nonresizable node, you need to modify its properties that affect its size. For example, to resize a rectangle, you need to change its `width` and `height` properties. `Regions`, `Controls`, and `WebView` are examples of resizable nodes. `Group`, `Text`, and `Shapes` are examples of nonresizable nodes.

How do you know if a node is resizable? The `isResizable()` method in the `Node` class returns `true` for a resizable node; it returns `false` for a nonresizable node.

The program in Listing 6-4 shows the behavior of resizable and nonresizable nodes during a layout. It adds a button and a rectangle to an `HBox`. After you run the program, make the stage shorter in width. The button becomes smaller up to a point when it displays an ellipsis (...). The rectangle remains the same size all the time. Figure 6-20 shows the stage at three different points during resizing.

### **Listing 6-4.** A Button and a Rectangle in an `HBox`

```
// ResizableNodeTest.java
package com.jdojo.node;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ResizableNodeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button btn = new Button("A big button");
        Rectangle rect = new Rectangle(100, 50);
        rect.setFill(Color.WHITE);
```

```

rect.setStrokeWidth(1);
rect.setStroke(Color.BLACK);

HBox root = new HBox();
root.setSpacing(20);
root.getChildren().addAll(btn, rect);

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Resizable Nodes");
stage.show();

System.out.println("btn.isResizable(): " + btn.isResizable());
System.out.println("rect.isResizable(): " + rect.isResizable());
}
}

btn.isResizable(): true
rect.isResizable(): false

```



**Figure 6-20.** A button and a rectangle shown in full size and after resizing the stage

## Resizable Nodes

The actual size of a resizable node is determined by two things:

- The sizing policy of the container in which the node is placed
- The sizing range specified by the node itself

Each container has a resizing policy for its children. I will discuss the resizing policy of containers in Chapter 10. A resizable node may specify a range for its size (width and height), which should be taken into account by an *honoringcontainer* for laying out the node. A resizable node specifies three types of sizes that constitute the range of its size:

- Preferred size
- Minimum size
- Maximum size

The *preferred size* of a node is its ideal width and height to display its contents. For example, a button in its preferred size would be big enough to display all its contents, based on the current properties such as the image, text, font, and text wrapping.

The *minimum size* of a node is the smallest width and height that it would like to have. For example, a button in its minimum size would be big enough to display the image and an ellipsis for its text. The *maximum size* of a node is the largest width and height that it would like to have. In the case of a button, the maximum size of a button is the same as its preferred size. Sometimes you may want to extend a node to an unlimited size. In those cases, the maximum width and height are set to `Double.MAX_VALUE`.

Most of the resizable nodes compute their preferred, minimum, and maximum sizes automatically, based on their contents and property settings. These sizes are known as their *intrinsic sizes*. The `Region` and `Control` classes define two constants that act as sentinel values for the intrinsic sizes of nodes. Those constants are:

- `USE_COMPUTED_SIZE`
- `USE_PREF_SIZE`

Both constants are of `double` type. The values for `USE_COMPUTED_SIZE` and `USE_PREF_SIZE` are `-1` and `Double.NEGATIVE_INFINITY`, respectively. It was not documented as to why the same constants were defined twice. Maybe the designers did not want to move them up in the class hierarchy, as they do not apply to all types of nodes.

If the size of a node is set to the sentinel value `USE_COMPUTED_SIZE`, the node will compute that size automatically based on its contents and properties settings. The `USE_PREF_SIZE` sentinel value is used to set the minimum and maximum sizes if they are the same as the preferred size.

The `Region` and `Control` classes have six properties of the `DoubleProperty` type to define preferred, minimum, and maximum values for their width and height:

- `prefWidth`
- `prefHeight`
- `minWidth`
- `minHeight`
- `maxWidth`
- `maxHeight`

By default, these properties are set to the sentinel value `USE_COMPUTED_SIZE`. That means, nodes compute these sizes automatically. You can set one of these properties to override the intrinsic size of a node. For example, you can set the preferred, minimum, and maximum width of a button to be 50 pixels as follows:

```
Button btn = new Button("Close");
btn.setPrefWidth(50);
btn.setMinWidth(50);
btn.setMaxWidth(50);
```

The above snippet of code sets preferred, minimum, and maximum widths of the button to the same value that makes the button horizontally nonresizable.

The following snippet of code sets the minimum and maximum widths of a button to the preferred width, where the preferred width itself is computed internally:

```
Button btn = new Button("Close");
btn.setMinWidth(Control.USE_PREF_SIZE);
btn.setMaxWidth(Control.USE_PREF_SIZE);
```

**Tip** In most cases, the internally computed values for preferred, minimum, and maximum sizes of nodes are fine. Use these properties to override the internally computed sizes only if they do not meet the needs of your application. If you need to bind the size of a node to an expression, you would need to bind the `prefWidth` and `prefHeight` properties.

How do you get the actual preferred, minimum, and maximum sizes of a node? You might guess that you can get them using

the `getPrefWidth()`, `getPrefHeight()`, `getMinWidth()`, `getMinHeight()`, `getMaxWidth()`, and `getMaxHeight()` methods. But you should not use these methods to get the actual sizes of a node. These sizes may be set to the sentinel values and the node will compute the actual sizes internally. These methods return the sentinel values or the override values. Listing 6-5 creates two buttons and overrides the preferred intrinsic width for one of them to 100 pixels. The resulting screen is shown in Figure 6-21. The output below proves that these methods are not very useful to learn the actual sizes of a node for layout purposes.

***Listing 6-5.*** Using `getXXXWidth()` and `getXXXHeight()` Methods of Regions and Controls

```
// NodeSizeSentinelValues.java
package com.jdojo.node;

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

public class NodeSizeSentinelValues extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        // Override the intrinsic width of the cancel button
        cancelBtn.setPrefWidth(100);

        VBox root = new VBox();
        root.getChildren().addAll(okBtn, cancelBtn);

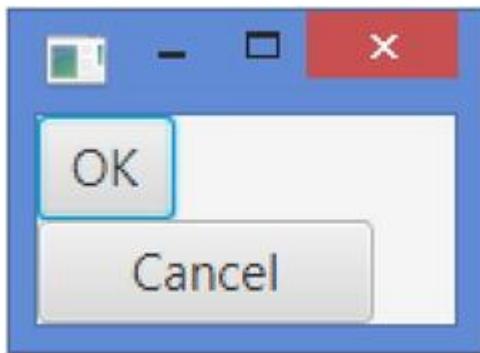
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Overriding Node Sizes");
        stage.show();

        System.out.println("okBtn.getPrefWidth(): " +
+ okBtn.getPrefWidth());
        System.out.println("okBtn.getMinWidth(): " +
+ okBtn.getMinWidth());
        System.out.println("okBtn.getMaxWidth(): " +
+ okBtn.getMaxWidth());

        System.out.println("cancelBtn.getPrefWidth(): " +
+ cancelBtn.getPrefWidth());
        System.out.println("cancelBtn.getMinWidth(): " +
+ cancelBtn.getMinWidth());
        System.out.println("cancelBtn.getMaxWidth(): " +
+ cancelBtn.getMaxWidth());
    }
}

okBtn.getPrefWidth(): -1.0
okBtn.getMinWidth(): -1.0
okBtn.getMaxWidth(): -1.0
cancelBtn.getPrefWidth(): 100.0
```

```
cancelBtn.getMinWidth(): -1.0
cancelBtn.getMaxWidth(): -1.0
```



**Figure 6-21.** Buttons using sentinel and override values for their widths

To get the actual sizes of a node, you need to use the following methods in the `Node` class. Note that the `Node` class does not define any properties related to sizes. The size-related properties are defined in the `Region`, `Control`, and other classes.

- `double prefWidth(double height)`
- `double prefHeight(double width)`
- `double minWidth(double height)`
- `double minHeight(double width)`
- `double maxWidth(double height)`
- `double maxHeight(double width)`

Here you can see another twist in getting the actual sizes of a node. You need to pass the value of its height to get its width and vice versa. For most nodes in JavaFX, width and height are independent. However, for some nodes, the height depends on the width and vice versa. When the width of a node depends on its height or vice versa, the node is said to have a *content bias*. If the height of a node depends on its width, the node has a *horizontal content bias*. If the width of a node depends on its height, the node has a *vertical content bias*. Note that a node cannot have both horizontal and vertical content biases, which will lead to a circular dependency.

The `getContentBias()` method of the `Node` class returns the content bias of a node. Its return type is the `javafx.geometry.Orientation` enum type, which has two constants: `HORIZONTAL` and `VERTICAL`. If a node does not have a content bias, for example, `Text` or `ChoiceBox`, the method returns `null`.

All controls that are subclasses of the `Labeled` class, for example, `Label`, `Button`, or `CheckBox`, have a `HORIZONTAL` content bias when they have the text wrapping property enabled. For some nodes, their content bias depends on their orientation. For example, if the orientation of a `FlowPane` is `HORIZONTAL`, its content bias is `HORIZONTAL`; if its orientation is `VERTICAL`, its content bias is `VERTICAL`.

You are supposed to use the above-listed six methods to get the sizes of a node for layout purposes. If a node type does not have a content bias, you need to pass `-1` to these methods as the value for the other dimension. For example, a `ChoiceBox` does not have a content bias, and you would get its preferred size as follows:

```
ChoiceBox choices = new ChoiceBox();
...
```

```
double prefWidth = choices.prefWidth(-1);
double prefHeight = choices.prefHeight(-1);
```

For those nodes that have a content bias, you need to pass the biased dimension to get the other dimension. For example, for a button, which has a HORIZONTAL content bias, you would pass -1 to get its width, and you would pass its width value to get its height as follows:

```
Button b = new Button("Hello JavaFX");

// Enable text wrapping for the button, which will change its
// content bias from null (default) to HORIZONTAL
b.setWrapText(true);
...
double prefWidth = b.prefWidth(-1);
double prefHeight = b.prefHeight(prefWidth);
```

If a button does not have the text wrap property enabled, you can pass -1 to both methods `prefWidth()` and `prefHeight()`, as it would not have a content bias. The generic way to get the width and height of a node for layout purposes is outlined as follows. The code shows how to get the preferred width and height, and the code would be similar to get minimum and maximum width and height of a node:

```
Node node = get the reference of of the node;
...
double prefWidth = -1;
double prefHeight = -1;

Orientation contentBias = b.getContentBias();

if (contentBias == HORIZONTAL) {
    prefWidth = node.prefWidth(-1);
    prefHeight = node.prefHeight(prefWidth);
} else if (contentBias == VERTICAL) {
    prefHeight = node.prefHeight(-1);
    prefWidth = node.prefWidth(prefHeight);
} else {
    // contentBias is null
    prefWidth = node.prefWidth(-1);
    prefHeight = node.prefHeight(-1);
}
```

Now you know how to get the specified values and the actual values for the preferred, minimum, and maximum sizes of a node. These values indicate the range for the size of a node. When a node is laid out inside a container, the container tries to give the node its preferred size. However, based on the container's policy and the specified size of the node, the node may not get its preferred size. Instead, an honoring container will give a node a size that is within its specified range. This is called the *current size*. How do you get the current size of a node?

The `Region` and `Control` classes define two *read-only* properties, `width` and `height`, that hold the values for the current width and height of a node.

Now let's see all these methods in action. Listing 6-6 places a button in an `HBox`, prints different types of sizes for the button, changes some properties, and prints the sizes of the button again. The output below shows that as the preferred width of the button becomes smaller, its preferred height becomes bigger.

### ***Listing 6-6.*** Using Different Size-Related Methods of a Node

```
// NodeSizes.java
package com.jdojo.node;

import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class NodeSizes extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button btn = new Button("Hello JavaFX!");

        HBox root = new HBox();
        root.getChildren().addAll(btn);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Sizes of a Node");
        stage.show();

        // Print button's sizes
        System.out.println("Before changing button properties:");
        printSizes(btn);

        // Change button's properties
        btn.setWrapText(true);
        btn.setPrefWidth(80);
        stage.sizeToScene();

        // Print button's sizes
        System.out.println("\nAfter changing button properties:");
        printSizes(btn);
    }

    public void printSizes(Button btn) {
        System.out.println("btn.getContentBias() = "
+ btn.getContentBias());

        System.out.println("btn.getPrefWidth() = " + btn.getPrefWidth() +
                           ", btn.getPrefHeight() = "
+ btn.getPrefHeight());

        System.out.println("btn.getMinWidth() = " + btn.getMinWidth() +
                           ", btn.getMinHeight() = " + btn.getMinHeight());

        System.out.println("btn.getMaxWidth() = " + btn.getMaxWidth() +
                           ", btn.getMaxHeight() = " + btn.getMaxHeight());

        double prefWidth = btn.prefWidth(-1);
        System.out.println("btn.prefWidth(-1) = " + prefWidth +
                           ", btn.prefHeight(prefWidth) = "
+ btn.prefHeight(prefWidth));

        double minWidth = btn.minWidth(-1);
        System.out.println("btn.minWidth(-1) = " + minWidth +
                           ", btn.minHeight(minWidth) = " + btn.minHeight(minWidth));

        double maxWidth = btn.maxWidth(-1);
        System.out.println("btn.maxWidth(-1) = " + maxWidth +
                           ", btn.maxHeight(maxWidth) = " + btn.maxHeight(maxWidth));

        System.out.println("btn.getWidth() = " + btn.getWidth() +
                           ", btn.getHeight() = " + btn.getHeight());
    }
}

```

```

    }
}

Before changing button properties:
btn.getContentBias() = null
btn.getPrefWidth() = -1.0, btn.getPrefHeight() = -1.0
btn.getMinWidth() = -1.0, btn.getMinHeight() = -1.0
btn.getMaxWidth() = -1.0, btn.getMaxHeight() = -1.0
btn.prefWidth(-1) = 107.0, btn.prefHeight(prefWidth) = 22.8984375
btn.minWidth(-1) = 37.0, btn.minHeight(minWidth) = 22.8984375
btn.maxWidth(-1) = 107.0, btn.maxHeight(maxWidth) = 22.8984375
btn.getWidth() = 107.0, btn.getHeight() = 23.0

After changing button properties:
btn.getContentBias() = HORIZONTAL
btn.getPrefWidth() = 80.0, btn.getPrefHeight() = -1.0
btn.getMinWidth() = -1.0, btn.getMinHeight() = -1.0
btn.getMaxWidth() = -1.0, btn.getMaxHeight() = -1.0
btn.prefWidth(-1) = 80.0, btn.prefHeight(prefWidth) = 39.796875
btn.minWidth(-1) = 37.0, btn.minHeight(minWidth) = 22.8984375
btn.maxWidth(-1) = 80.0, btn.maxHeight(maxWidth) = 39.796875
btn.getWidth() = 80.0, btn.getHeight() = 40.0

```

The list of methods to get or set sizes of resizable nodes is not over. There are some convenience methods that can be used to perform the same task as the methods discussed in this section. Table 6-2 lists the size-related methods with their defining classes and usage.

**Table 6-2.** Size-Related Methods of Resizable Nodes

Methods/Properties	Defining Class	Usage
<b>Properties:</b> prefWidth prefHeight minWidth minHeight maxWidth maxHeight	Region, Control	They define the preferred, minimum, and maximum set to sentinel values by default. Use them to override values.
<b>Methods:</b> double prefWidth(double h) double prefHeight(double w) double minWidth(double h) double minHeight(double w)	Node	Use them to get the actual sizes of nodes. Pass -1 as the node does not have a content bias. Pass the actual other dimension as the argument if the node has a content bias. Note that there are no corresponding properties to the methods.

Methods/Properties	Defining Class	Usage
<pre>double maxWidth(double h)  double maxHeight(double w)</pre>		
<b>Properties:</b> width height	Region, Control	These are <i>read-only</i> properties that hold the current width and height of resizable nodes.
<b>Methods:</b> void setPreferredSize(double w, double h) void setMinSize(double w, double h) void setMaxSize(double w, double h)	Region, Control	These are convenience methods to override the default width and height of nodes.
<b>Methods:</b> void resize(double w, double h)	Node	It resizes a node to the specified width and height. It is called by the parent of the node during a layout. You should not call this method directly in your code. If you need to set the size of a nonresizable node, use the <code>setMinSize()</code> , <code>setPreferredSize()</code> , or <code>setMaxSize()</code> methods instead. This method has no effect on a nonresizable node.
<b>Methods:</b> void autosize()	Node	For a resizable node, it sets the layout bounds to its current preferred width and height. It takes care of the content. This method has no effect on a nonresizable node.

## Nonresizable Nodes

Nonresizable nodes are not resized by their parents during layout. However, you can change their sizes by changing their properties. Nonresizable nodes (e.g., all shapes) have different properties that determine their sizes. For example, the width and height of a rectangle, the radius of a circle, and the `(startX, startY)` and `(endX, endY)` of a line determine their sizes.

There are several size-related methods defined in the `Node` class. Those methods have no effect when they are called on nonresizable nodes or they return their current size. For example, calling the `resize(double w, double h)` method of

the `Node` class on a nonresizable node has no effect. For a nonresizable node, the `prefWidth(double h)`, `minWidth(double h)`, and `maxWidth(double h)` methods in the `Node` class return its `layoutBounds` width; whereas `prefHeight(double w)`, `minHeight(double w)`, and `maxHeight(double w)` methods return its `layoutBounds` height. Nonresizable nodes do not have content bias. Pass `-1` to all these methods as the argument for the other dimension.

## Storing User Data in a Node

Every node maintains an observable map of user-defined properties (key/value pairs). You can use it to store any useful information. Suppose you have a `TextField` that lets the user manipulate a person's name. You can store the originally retrieved person's name from the database as the property of the `TextField`. You can use the property later to reset the name or to generate an `UPDATE` statement to update the name in the database. Another use of the properties would be to store micro help text. When a node receives the focus, you can read its `microHelp` property and display it, for example, in a status bar, to help the user understand the use of the node.

The `getProperties()` method of the `Node` class returns an `ObservableMap<Object, Object>` in which you can add or remove properties for the node. The following snippet of code adds a property "originalData" with a value "Advik" to a `TextField` node:

```
TextField nameField = new TextField();
...
ObservableMap<Object, Object> props = nameField.getProperties();
props.put("originalData", "Advik");
```

The following snippet of code reads the value of the "originalData" property from the `nameField` node:

```
ObservableMap<Object, Object> props = nameField.getProperties();
if (props.containsKey("originalData")) {
    String originalData = (String)props.get("originalData");
} else {
    // originalData property is not set yet
}
```

The `Node` class has two convenience methods, `setUserData(Object value)` and `getUserData()`, to store a user-defined value as a property for a node. The value specified in the `setUserData()` method uses the same `ObservableMap` to store the data that are returned by the `getProperties()` method. The `Node` class uses an internal `Object` as the key to store the value. You need to use the `getUserData()` method to get the value that you store using the `setUserData()` method, as follows:

```
nameField.setUserData("Saved"); // Set the user data
...
String userData = (String)nameField.getUserData(); // Get the user data
```

**Tip** You cannot access the user data of a node directly except by using the `getUserData()` method. Because it is stored in the same `ObservableMap` returned by the `getProperties()` method, you can get to it indirectly by iterating through the values in that map.

The `Node` class has a `hasProperties()` method. It tests if a node has properties. Its implementation seems to be wrong as of JavaFX version 2.2. The `Node` class

creates an `ObservableMap` to store properties lazily. It is created when you call the `getProperties()` of `setUserData()` method for the first time. The implementation of the `hasProperties()` method returns `true` if the internal `ObservableMap` object has been created. It does not check if the internal map has any key/value pair in it:

```
TextField nameField = new TextField();
System.out.println(nameField.getProperties());
ObservableMap<Object, Object> props = nameField.getProperties();
System.out.println(nameField.getProperties());
false
true
```

The above snippet of code should print `false` twice. However, it prints `true` for the second time because you have called the `getProperties()` method, which is wrong. Your `nameField` node still has no properties. Let's see if this would get fixed in the later version.

## What Is a Managed Node?

The `Node` class has a managed property, which is of type `BooleanProperty`. By default, all nodes are managed. The laying out of a managed node is managed by its parent. A Parent node takes into account the `layoutBounds` of all its managed children when it computes its own size. A Parent node is responsible for resizing its managed resizable children and positioning them according to its layout policy. When the `layoutBounds` of a managed child changes, the relevant part of the scene graph is relaid out.

If a node is unmanaged, the application is solely responsible for laying it out (computing its size and position). That is, a Parent node does not lay out its unmanaged children. Changes in the `layoutBounds` of an unmanaged node do not trigger the relayout above it. An unmanaged Parent node acts as a *layout root*. If a child node calls the `Parent.requestLayout()` method, only the branch rooted by the unmanaged Parent node is relaid out.

**Tip** Contrast the `visible` property of the `Node` class with its `managed` property. A Parent node takes into account the `layoutBounds` of all its invisible children for layout purposes and ignores the unmanaged children.

When would you use an unmanaged node? Typically, you do not need to use unmanaged nodes in applications because they need additional work on your part. However, just know that they exist and you can use them, if needed.

You can use an unmanaged node when you want to show a node in a container without the container considering its `layoutBounds`. You will need to size and position the node yourself. Listing 6-7 demonstrates how to use unmanaged nodes. It uses an unmanaged `Text` node to display a micro help when a node has the focus. The node needs to have a property named "`microHelpText`". When the micro help is shown, the layout for the entire application is not disturbed as the `Text` node to show the micro help is an unmanaged node. You place the node at an appropriate position in the `focusChanged()` method. The program registers a change listener to the `focusOwner` property of the scene, so you show or hide the micro help `Text` node when the focus inside the scene changes. The resulting screens, when two different nodes have focus, are shown in Figure 6-22. Note that positioning the `Text` node, in this example, was easy as all nodes were inside the same parent

node, a `GridPane`. The logic to position the `Text` node becomes complex if nodes are placed inside different parents.

### ***Listing 6-7. Using an Unmanaged Text Node to Show Micro Help***

```
// MicroHelpApp.java
package com.jdojo.node;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.value.ObservableValue;
import javafx.geometry.VPos;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MicroHelpApp extends Application {
    // An instance variable to store the Text node reference
    private Text helpText = new Text();

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

    @Override
    public void start(Stage stage) {
        TextField fName = new TextField();
        TextField lName = new TextField();
        TextField salary = new TextField();

        Button closeBtn = new Button("Close");
        closeBtn.setOnAction(e -> Platform.exit());

        fName.getProperties().put("microHelpText", "Enter the first
name");
        lName.getProperties().put("microHelpText", "Enter the last
name");
        salary.getProperties().put("microHelpText",
                               "Enter a salary greater than $2000.00.");

        // The help text node is unmanaged
        helpText.setManaged(false);
        helpText.setTextOrigin(VPos.TOP);
        helpText.setFill(Color.RED);
        helpText.setFont(Font.font(null, 9));
        helpText.setMouseTransparent(true);

        // Add all nodes to a GridPane
        GridPane root = new GridPane();

        root.add(new Label("First Name:"), 1, 1);
        root.add(fName, 2, 1);
        root.add(new Label("Last Name:"), 1, 2);
        root.add(lName, 2, 2);

        root.add(new Label("Salary:"), 1, 3);
        root.add(salary, 2, 3);
    }
}
```

```

root.add(closeBtn, 3, 3);
root.add(helpText, 4, 3);

Scene scene = new Scene(root, 300, 100);

// Add a change listener to the scene, so you know when the focus
owner
// changes and display the micro help
scene.focusOwnerProperty().addListener(
    (ObservableValue<? extends Node> value, Node oldNode, Node
newNode)
        -> focusChanged(value, oldNode, newNode));
stage.setScene(scene);
stage.setTitle("Showing Micro Help");
stage.show();
}

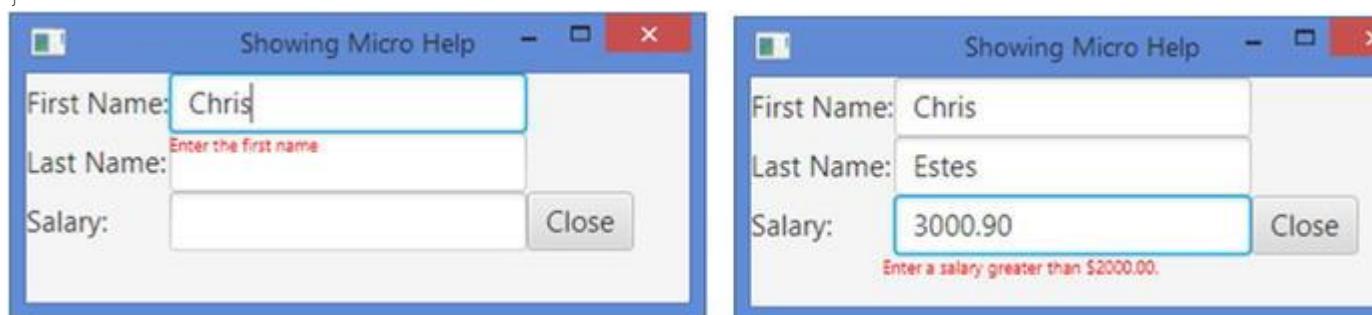
public void focusChanged(ObservableValue<? extends Node> value,
    Node oldNode, Node newNode) {
    // Focus has changed to a new node
    String microHelpText
= (String)newNode.getProperties().get("microHelpText");

    if (microHelpText != null && microHelpText.trim().length() >
0) {
        helpText.setText(microHelpText);
        helpText.setVisible(true);

        // Position the help text node
        double x = newNode.getLayoutX() +
            newNode.getLayoutBounds().getMinX() -
            helpText.getLayoutBounds().getMinX();
        double y = newNode.getLayoutY() +
            newNode.getLayoutBounds().getMinY() +
            newNode.getLayoutBounds().getHeight() -
            helpText.getLayoutBounds().getMinX();

        helpText.setLayoutX(x);
        helpText.setLayoutY(y);
        helpText.setWrappingWidth(newNode.getLayoutBounds().getWidth());
    }
    else {
        helpText.setVisible(false);
    }
}
}

```



**Figure 6-22.** Using an unmanaged Text node to show micro help

Sometimes you may want to use the space that is used by a node if the node becomes invisible. Suppose you have an `HBox`with several buttons. When one of the buttons becomes invisible, you want to slide all buttons from right to left. You can achieve a

slide-up effect in `VBox`. Achieving sliding effects in `HBox` and `VBox` (or any other containers with relative positioning) is easy by binding the `managed` property of the node to the `visible` property. Listing 6-8 shows how to achieve the slide-left feature in an `HBox`. It displays four buttons. The first button is used to make the third button, `b2`, visible and invisible. The `managed` property of the `b2` button is bound to its `visible` property:

```
b2.managedProperty().bind(b2.visibleProperty());
```

When the `b2` button is made invisible, it becomes unmanaged, and the `HBox` does not use its `layoutBounds` in computing its own `layoutBounds`. This makes the `b3` button slide to the left. Figure 6-23 shows two screenshots when the application is run.

### ***Listing 6-8.*** Simulating the Slide-Left Feature Using Unmanaged Nodes

```
// SlidingLeftNodeTest.java
package com.jdojo.node;

import javafx.application.Application;
import javafx.beans.binding.When;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class SlidingLeftNodeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button b1 = new Button("B1");
        Button b2 = new Button("B2");
        Button b3 = new Button("B3");
        Button visibleBtn = new Button("Make Invisible");

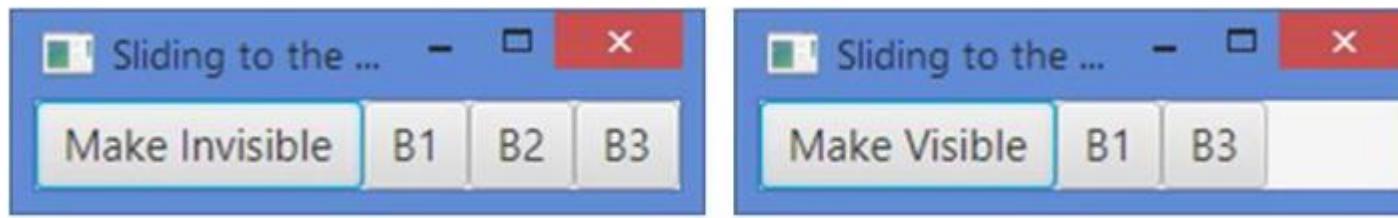
        // Add an action listener to the button to make b2 visible
        // if it is invisible and invisible if it is visible
        visibleBtn.setOnAction(e -> b2.setVisible(!b2.isVisible()));

        // Bind the text property of the button to the visible
        // property of the b2 button
        visibleBtn.textProperty().bind(new When(b2.visibleProperty())
            .then("Make Invisible")
            .otherwise("Make Visible"));

        // Bind the managed property of b2 to its visible property
        b2.managedProperty().bind(b2.visibleProperty());

        HBox root = new HBox();
        root.getChildren().addAll(visibleBtn, b1, b2, b3);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Sliding to the Left");
        stage.show();
    }
}
```



**Figure 6-23.** Simulating the slide-left feature for B2 button

## Transforming Bounds between Coordinate Spaces

I have already covered coordinate spaces used by nodes. Sometimes you may need to translate a `Bounds` or a point from one coordinate space to another. The `Node` class contains several methods to support this. The following transformations of a `Bounds` or a point are supported:

- Local to parent
- Local to scene
- Parent to local
- Scene to local

The `localToParent()` method transforms a `Bounds` or a point in the local coordinate space of a node to the coordinate space of its parent.

The `localToScene()` method transforms a `Bounds` or a point in the local coordinate space of a node to the coordinate space of its scene.

The `parentToLocal()` method transforms a `Bounds` or a point in the coordinate space of the parent of a node to the local coordinate space of the node.

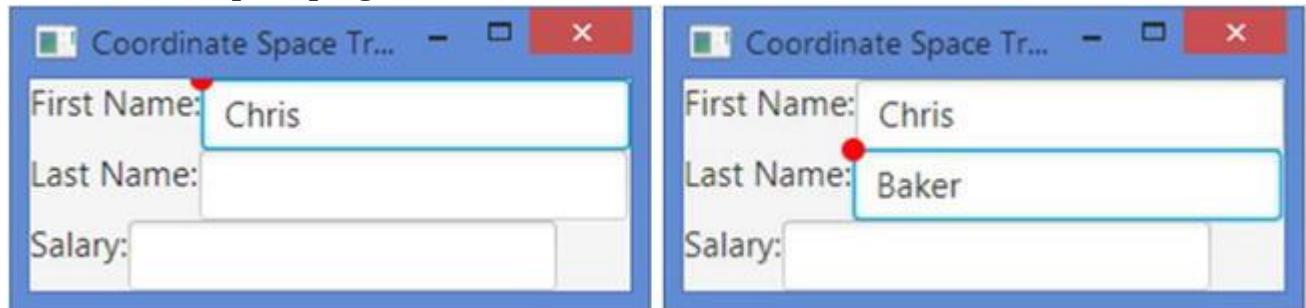
The `sceneToLocal()` method transforms a `Bounds` or a point in the coordinate space of the scene of a node to the local coordinate space of the node. All methods have three overloaded versions; one version takes a `Bounds` as an argument and returns the transformed `Bounds`; another version takes a `Point2D` as an argument and returns the transformed `Point2D`; another version takes the x and y coordinates of a point and returns the transformed `Point2D`.

These methods are sufficient to transform the coordinate of a point in one coordinate space to another within a scene graph. Sometimes you may need to transform the coordinates of a point in the local coordinate space of a node to the coordinate space of the stage or screen. You can achieve this using the `x` and `y` properties of the `Scene` and `Stage` classes. The `(x, y)` properties of a scene define the coordinates of the top left corner of the scene in the coordinate space of its stage. The `(x, y)` properties of a stage define the coordinates of the top left corner of the stage in the coordinate space of the screen. For example, if  $(x_1, y_1)$  is a point in the coordinate space of the scene,  $(x_1 + x_2, y_1 + y_2)$  defines the same point in the coordinate space of the stage, where  $x_2$  and  $y_2$  are the `x` and `y` properties of the stage, respectively.

Apply the same logic to get the coordinate of a point in the coordinate space of the screen.

Let's look at an example that uses transformations between the coordinate spaces of a node, its parent, and its scene. A scene has three `Labels` and three `TextFields` placed under different parents. A red, small circle is placed at the top left corner of the bounding box of the node that has the focus. As the focus changes, the position of the circle needs to be computed, which would be the same as the position of the top

left corner of the current node, relative to the parent of the circle. The center of the circle needs to coincide with the top left corner of the node that has the focus. Figure 6-24 shows the stage when the focus is in the first name and last name nodes. Listing 6-9 has the complete program to achieve this.



**Figure 6-24.** Using coordinate space transformations to move a circle to a focused node

The program has a scene consisting of three Labels and TextFields. A pair of a Label and a TextField is placed in an HBox. All HBoxes are placed in a VBox. An unmanaged Circle is placed in the VBox. The program adds a change listener to the focusOwner property of the scene to track the focus change. When the focus changes, the circle is placed at the top left corner of the node that has the focus.

The placeMarker() contains the main logic. It gets the (x, y) coordinates of the top left corner of the bounding box of the node in focus in the local coordinate space:

```
double nodeMinX = newNode.getLayoutBounds().getMinX();
double nodeMinY = newNode.getLayoutBounds().getMinY();
```

It transforms the coordinates of the top left corner of the node from the local coordinate space to the coordinate space of the scene:

```
Point2D nodeInScene = newNode.localToScene(nodeMinX, nodeMinY);
```

Now the coordinates of the top left corner of the node are transformed from the coordinate space of the scene to the coordinate space of the circle, which is named marker in the program:

```
Point2D nodeInMarkerLocal = marker.sceneToLocal(nodeInScene);
```

Finally, the coordinate of the top left corner of the node is transformed to the coordinate space of the parent of the circle:

```
Point2D nodeInMarkerParent = marker.localToParent(nodeInMarkerLocal);
```

At this point, the nodeInMarkerParent is the point (the top left corner of the node in focus) relative to the parent of the circle. If you relocate the circle to this point, you will place the top left corner of the bounding box of the circle to the top left corner of the node in focus:

```
marker.relocate(nodeInMarkerParent.getX(), nodeInMarkerParent.getY())
```

If you want to place the center of the circle to the top left corner of the node in focus, you will need to adjust the coordinates accordingly:

```
marker.relocate(nodeInMarkerParent.getX() + marker.getLayoutBounds().getMinX(),
                nodeInMarkerParent.getY() + marker.getLayoutBounds().getMinY());
```

### **Listing 6-9.** Transforming the Coordinates of a Point from One Coordinate Space to Another

```
// CoordinateConversion.java
package com.jdojo.node;

import javafx.application.Application;
```

```

import javafx.geometry.Point2D;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class CoordinateConversion extends Application {
    // An instance variable to store the reference of the circle
    private Circle marker;

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

    @Override
    public void start(Stage stage) {
        TextField fName = new TextField();
        TextField lName = new TextField();
        TextField salary = new TextField();

        // The Circle node is unmanaged
        marker = new Circle(5);
        marker.setManaged(false);
        marker.setFill(Color.RED);
        marker.setMouseTransparent(true);

        HBox hb1 = new HBox();
        HBox hb2 = new HBox();
        HBox hb3 = new HBox();
        hb1.getChildren().addAll(new Label("First Name:"), fName);
        hb2.getChildren().addAll(new Label("Last Name:"), lName);
        hb3.getChildren().addAll(new Label("Salary:"), salary);

        VBox root = new VBox();
        root.getChildren().addAll(hb1, hb2, hb3, marker);

        Scene scene = new Scene(root);

        // Add a focus change listener to the scene
        scene.focusOwnerProperty().addListener(
            (prop, oldNode, newNode) -> placeMarker(newNode));
    }

    public void placeMarker(Node newNode) {
        double nodeMinX = newNode.getLayoutBounds().getMinX();
        double nodeMinY = newNode.getLayoutBounds().getMinY();
        Point2D nodeInScene = newNode.localToScene(nodeMinX, nodeMinY);
        Point2D nodeInMarkerLocal = marker.sceneToLocal(nodeInScene);
        Point2D nodeInMarkerParent
        = marker.localToParent(nodeInMarkerLocal);

        // Position the circle appropriately
        marker.relocate(nodeInMarkerParent.getX()
            + marker.getLayoutBounds().getMinX(),
            nodeInMarkerParent.getY()
            + marker.getLayoutBounds().getMinY());
    }
}

```

```
}
```

## Summary

A scene graph is a tree data structure. Every item in a scene graph is called a node. An instance of the `javafx.scene.Node` class represents a node in the scene graph. A node can have subitems (also called children) and such a node is called a branch node. A branch node is an instance of the `Parent` class whose concrete subclasses are `Group`, `Region`, and `WebView`. A node that cannot have subitems is called a leaf node. Instances of classes such as `Rectangle`, `Text`, `ImageView`, and `MediaView` are examples of leaf nodes. Only a single node within each scene graph tree will have no parent, which is referred to as the root node. A node may occur at the most once anywhere in the scene graph.

A node may be created and modified on any thread if it is not yet attached to a scene. Attaching a node to a scene and subsequent modification must occur on the JavaFX Application Thread. A node has several types of bounds. Bounds are determined with respect to different coordinate systems. A node in a scene graph has three types of bounds: `layoutBounds`, `boundsInLocal`, and `boundsInParent`.

The `layoutBounds` property is computed based on the geometric properties of the node in the *untransformed* local coordinate space of the node. Effects, clip, and transformations are not included. The `boundsInLocal` property is computed in the untransformed coordinate space of the node. It includes the geometric properties of the node, effects, and clip. Transformations applied to a node are not included.

The `boundsInParent` property of a node is in the coordinate space of its parent. It includes the geometric properties of the node, effects, clip, and transformations. It is rarely used directly in code.

The computation of `layoutBounds`, `boundsInLocal`, and `boundsInParent` for a `Group` is different from that of a node. A `Group` takes on the collection bounds of its children. You can apply effects, clip, and transformations separately on each child of a `Group`. You can also apply effects, clip, and transformations directly on a `Group` and they are applied to all its children nodes. The `layoutBounds` of a `Group` is the union of the `boundsInParent` of all its children. It includes effects, clip, and transformations applied directly to the children. It does not include effects, clip, and transformations applied directly to the `Group`. The `boundsInLocal` of a `Group` is computed taking its `layoutBounds` and including the effects and clip applied directly to the `Group`. The `boundsInParent` of a `Group` is computed by taking its `boundsInLocal` and including the transformations applied directly to the `Group`.

Every node maintains an observable map of user-defined properties (key/value pairs). You can use it to store any useful information. A node can be managed or unmanaged. A managed node is laid out by its parent, whereas the application is responsible for laying out an unmanaged node.

The next chapter will discuss how to use colors in JavaFX.

## CHAPTER 7



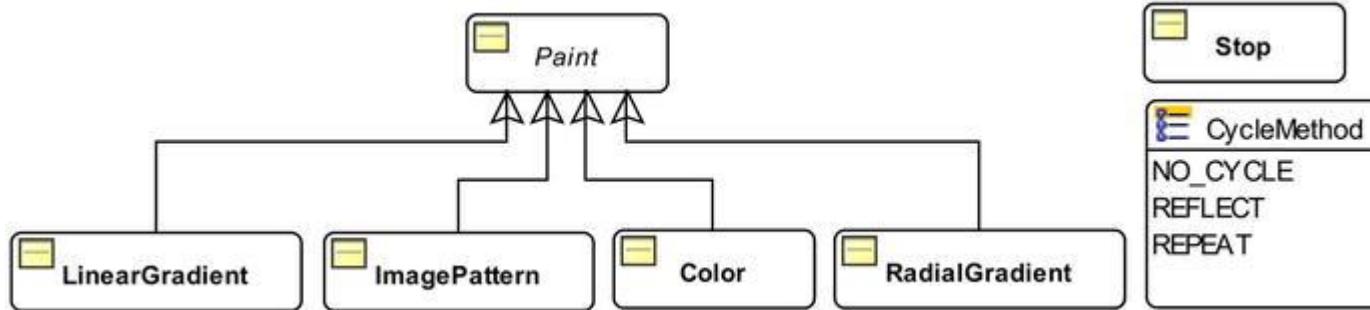
### Playing with Colors

In this chapter, you will learn:

- How colors are represented in JavaFX
- What different color patterns are
- How to use image pattern
- How to use linear color gradient
- How to use radial color gradient

### Understanding Colors

In JavaFX, you can specify color for text and background color for regions. You can specify a color as a uniform color, an image pattern, or a color gradient. A uniform color uses the same color to fill the entire region. An image pattern lets you fill a region with an image pattern. A color gradient defines a color pattern in which the color varies along a straight line from one color to another. The variation in a color gradient can be linear or radial. I will present examples using all color types in this chapter. Figure 7-1 shows the class diagram for color-related classes in JavaFX. All classes are included in the `javafx.scene.paint` package.



**Figure 7-1.** The class diagram of color-related classes in JavaFX

The `Paint` class is an abstract class and it is the base class for other color classes. It contains only one static method that takes a `String` argument and returns a `Paint` instance. The returned `Paint` instance would be of the `Color`, `LinearGradient`, or `RadialGradient` class, as shown in the following code:

```
public static Paint valueOf(String value)
```

You will not use the `valueOf()` method of the `Paint` class directly. It is used to convert the color value read in a `String` from the CSS files. The following snippet of code creates instances of the `Paint` class from `Strings`:

```
// redColor is an instance of the Color class
Paint redColor = Paint.valueOf("red");

// aLinearGradientColor is an instance of the LinearGradient class
Paint aLinearGradientColor = Paint.valueOf("linear-gradient(to bottom right,
```

```
red, black) " );
// aRadialGradientColor is an instance of the RadialGradient class
Paint aRadialGradientColor = Paint.valueOf("radial-gradient(radius 100%, red,
blue, black)");
```

A uniform color, an image pattern, a linear color gradient, and a radial color gradient are instances of the `Color`, `ImagePattern`, `LinearGradient`, and `RadialGradient` classes, respectively. The `Stop` class and the `CycleMethod` enum are used while working with color gradients.

**Tip** Typically, methods for setting the `color` attribute of a node take the `Paint` type as an argument, allowing you to use any of the four color patterns.

## Using the Color Class

The `Color` class represents a solid uniform color from the RGB color space. Every color has an alpha value defined between 0.0 to 1.0 or 0 to 255. An alpha value of 0.0 or 0 means the color is completely transparent, and an alpha value of 1.0 or 255 denotes a completely opaque color. By default, the alpha value is set to 1.0. You can have an instance of the `Color` class in three ways:

- Using the constructor
- Using one of the factory methods
- Using one of the color constants declared in the `Color` class

The `Color` class has only one constructor that lets you specify the RGB and opacity in the range of 0.0 and 1.0:

```
public Color(double red, double green, double blue, double opacity)
```

The following snippet of code creates a completely opaque blue color:

```
Color blue = new Color(0.0, 0.0, 1.0, 1.0);
```

You can use the following static methods in the `Color` class to create `Color` objects. The double values need to be between 0.0 and 1.0 and int values between 0 and 255:

- `Color color(double red, double green, double blue)`
- `Color color(double red, double green, double blue, double opacity)`
- `Color hsb(double hue, double saturation, double brightness)`
- `Color hsb(double hue, double saturation, double brightness, double opacity)`
- `Color rgb(int red, int green, int blue)`
- `Color rgb(int red, int green, int blue, double opacity)`

The `valueOf()` and `web()` factory methods let you create `Color` objects from strings in web color value formats. The following snippet of code creates blue `Color` objects using different string formats:

```
Color blue = Color.valueOf("blue");
Color blue = Color.web("blue");
Color blue = Color.web("#0000FF");
Color blue = Color.web("0X0000FF");
```

```
Color blue = Color.web("rgb(0, 0, 255)");
Color blue = Color.web("rgba(0, 0, 255, 0.5)"); // 50% transparent blue
```

The `Color` class defines about 140 color constants, for example, `RED`, `WHITE`, `TAN`, `BLUE`, among others. Colors defined by these constants are completely opaque.

### Using the `ImagePattern` Class

An image pattern lets you fill a shape with an image. The image may fill the entire shape or use a tiling pattern. Here are the steps you would use to get an image pattern:

1. Create an `Image` object using an image from a file.
2. Define a rectangle, known as the anchor rectangle, relative to the upper left corner of the shape to be filled.

The image is shown in the anchor rectangle and is then resized to fit the anchor rectangle. If the bounding box for the shape to be filled is bigger than that of the anchor rectangle, the anchor rectangle with the image is repeated within the shape in a tiling pattern.

You can create an object of the `ImagePattern` using one of its constructors:

- `ImagePattern(Image image)`
- `ImagePattern(Image image, double x, double y, double width, double height, boolean proportional)`

The first constructor fills the entire bounding box with the image without any pattern. The second constructor lets you specify the x and y coordinates, width, and height of the anchor rectangle. If the `proportional` argument is true, the anchor rectangle is specified relative to the bounding box of the shape to be filled in terms of a unit square. If the `proportional` argument is false, the anchor rectangle is specified in the local coordinate system of the shape. The following two calls to the two constructors would produce the same result:

```
ImagePattern ip1 = new ImagePattern(anImage);
ImagePattern ip2 = new ImagePattern(anImage, 0.0, 0.0, 1.0, 1.0, true);
```

For the example here, you will use the image shown in Figure 7-2. It is a 37px by 25px blue rounded rectangle. It can be found in

the `resources/picture/blue_rounded_rectangle.png` file under the source code folder.



**Figure 7-2.** A blue rounded rectangle

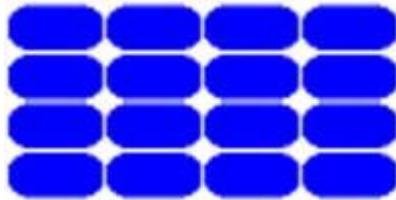
Using that file, let's create an image pattern, using the following code:

```
Image img = create the image object...
ImagePattern p1 = new ImagePattern(img, 0, 0, 0.25, 0.25, true);
```

The last argument in the `ImagePattern` constructor set to `true` makes the bounds of the anchor rectangle, `0, 0, 0.25`, and `0.25`, to be interpreted proportional to the size of the shape to be filled. The image pattern will create an anchor rectangle

at  $(0, 0)$  of the shape to be filled. Its width and height will be 25% of the shape to be filled. This will make the anchor rectangle repeat four times horizontally and four times vertically. If you use the following code with the above image pattern, it will produce a rectangle as shown in Figure 7-3.

```
Rectangle r1 = new Rectangle(100, 50);
r1.setFill(p1);
```



**Figure 7-3.** Filling a rectangle with an image pattern

If you use the same image pattern to fill a triangle with the following snippet of code, the resulting triangle will look like the one shown in Figure 7-4.

```
Polygon triangle = new Polygon(50, 0, 0, 50, 100, 50);
triangle.setFill(p1);
```



**Figure 7-4.** Filling a triangle with an image pattern

How would you fill a shape completely with an image without having a tiling pattern? You would need to use an `ImagePattern` with the proportional argument set to true. The center of the anchor rectangle should be at  $(0, 0)$  and its width and height should be set to 1 as follows:

```
// An image pattern to completely fill a shape with the image
ImagePattern ip = new ImagePattern(yourImage, 0.0, 0.0, 1.0, 1.0, true);
```

The program in Listing 7-1 shows how to use an image pattern. The resulting screen is shown in Figure 7-5. Its `init()` method loads an image in an `Image` object and stores it in an instance variable. If the image file is not found in the CLASSPATH, it prints an error message and quits.

### **Listing 7-1.** Using an Image Pattern to Fill Different Shapes

```
// ImagePatternApp.java
package com.jdojo.color;

import java.net.URL;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.layout.HBox;
import javafx.scene.paint.ImagePattern;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ImagePatternApp extends Application {
    private Image img;
```

```

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

@Override
public void init() {
    // Create an Image object
    final String IMAGE_PATH
= "resources/picture/blue_rounded_rectangle.png";
    URL url = this.getClass().getClassLoader().getResource(IMAGE_PATH);
    if (url == null) {
        System.out.println(IMAGE_PATH + " file not found in
CLASSPATH");
        Platform.exit();
        return;
    }
    img = new Image(url.toExternalForm());
}

@Override
public void start(Stage stage) {
    // An anchor rectangle at (0, 0) that is 25% wide and 25% tall
    // relative to the rectangle to be filled
    ImagePattern p1 = new ImagePattern(img, 0, 0, 0.25, 0.25, true);
    Rectangle r1 = new Rectangle(100, 50);
    r1.setFill(p1);

    // An anchor rectangle at (0, 0) that is 50% wide and 50% tall
    // relative to the rectangle to be filled
    ImagePattern p2 = new ImagePattern(img, 0, 0, 0.5, 0.5, true);
    Rectangle r2 = new Rectangle(100, 50);
    r2.setFill(p2);

    // Using absolute bounds for the anchor rectangle
    ImagePattern p3 = new ImagePattern(img, 40, 15, 20, 20, false);
    Rectangle r3 = new Rectangle(100, 50);
    r3.setFill(p3);

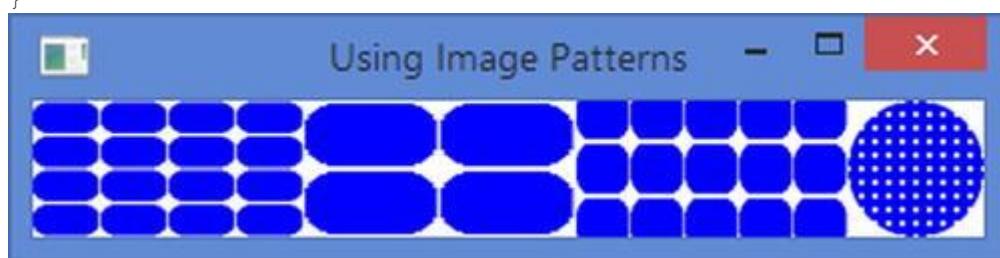
    // Fill a circle
    ImagePattern p4 = new ImagePattern(img, 0, 0, 0.1, 0.1, true);
    Circle c = new Circle(50, 50, 25);
    c.setFill(p4);

    HBox root = new HBox();
    root.getChildren().addAll(r1, r2, r3, c);

    Scene scene = new Scene(root);
    stage.setScene(scene);

    stage.setTitle("Using Image Patterns");
    stage.show();
}
}

```



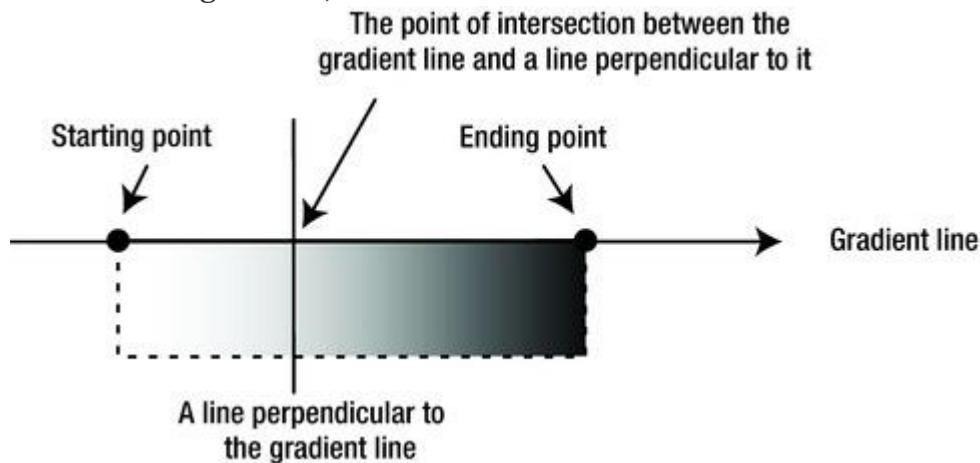
**Figure 7-5.** Filling different shapes with image patterns

## Understanding Linear Color Gradient

A linear color gradient is defined using an axis known as a *gradient line*. Each point on the gradient line is of a different color. All points on a line that is perpendicular to the gradient line have the same color, which is the color of the point of intersection between the two lines. The gradient line is defined by a starting point and an ending point. Colors along the gradient line are defined at some points on the gradient line, which are known as *stop-color points* (or stop points). Colors between two stop points are computed using interpolation.

The gradient line has a direction, which is from the starting point to the ending point. All points on a line perpendicular to the gradient line that pass through a stop point will have the color of the stop point. For example, suppose you have defined a stop point P<sub>1</sub> with a color C<sub>1</sub>. If you draw a line perpendicular to the gradient line passing through the point P<sub>1</sub>, all points on that line will have the color C<sub>1</sub>.

Figure 7-6 shows the details of the elements constituting a linear color gradient. It shows a rectangular region filled with a linear color gradient. The gradient line is defined from the left side to the right side. The starting point has a white color and the ending point has a black color. On the left side of the rectangle, all points have the white color, and on the right side, all points have the black color. In between the left and the right sides, the color varies between white and black.



**Figure 7-6.** The details of a linear color gradient

### Using the `LinearGradient` Class

In JavaFX, an instance of the `LinearGradient` class represents a linear color gradient. The class has the following two constructors. The types of their last arguments are different:

- `LinearGradient(double startX, double startY, double endX, double endY, boolean proportional, CycleMethod cycleMethod, List<Stop> stops)`
- `LinearGradient(double startX, double startY, double endX, double endY, boolean proportional, CycleMethod cycleMethod, Stop... stops)`

The `startX` and `startY` arguments define the x and y coordinates of the starting point of the gradient line. The `endX` and `endY` arguments define the x and y coordinates of the ending point of the gradient line.

The `proportional` argument affects the way the coordinates of the starting and ending points are treated. If it is true, the starting and ending points are treated relative to a unit square. Otherwise, they are treated as absolute values in the local coordinate system. The use of this argument needs a little more explanation.

Typically, a color gradient is used to fill a region, for example, a rectangle. Sometimes, you know the size of the region and sometimes you will not. The value of this argument lets you specify the gradient line in relative or absolute form. In relative form, the region is treated as a unit square. That is, the coordinates of the upper left and the lower right corners are (0.0, 0.0) and (1.0, 1.0), respectively. Other points in the regions will have x and y coordinates between 0.0 and 1.0. Suppose you specify the starting point as (0.0, 0.0) and the ending point as (1.0, 0.0). It defines a horizontal gradient line from the left to right. The starting and ending points of (0.0, 0.0) and (0.0, 1.0) define a vertical gradient line from top to bottom. The starting and ending points of (0.0, 0.0) and (0.5, 0.0) define a horizontal gradient line from left to middle of the region.

When the `proportional` argument is false, the coordinate values for the starting and ending points are treated as absolute values with respect to the local coordinate system. Suppose you have a rectangle of width 200 and height 100. The starting and ending points of (0.0, 0.0) and (200.0, 0.0) define a horizontal gradient line from left to right. The starting and ending points of (0.0, 0.0) and (200.0, 100.0) define a diagonal gradient line from the top left corner to the bottom right corner.

The `cycleMethod` argument defines how the regions outside the color gradient bounds, defined by the starting and ending points, should be filled. Suppose you define the starting and ending points with the proportional argument set to `true` as (0.0, 0.0) and (0.5, 0.0), respectively. This covers only the left half of the region. How should the right half of the region be filled? You specify this behavior using the `cycleMethod` argument. Its value is one of the enum constants defined in the `CycleMethod` enum:

- `CycleMethod.NO_CYCLE`
- `CycleMethod.REFLECT`
- `CycleMethod.REPEAT`

The cycle method of `NO_CYCLE` fills the remaining region with the terminal color. If you have defined color a stop point only from the left to the middle of a region, the right half will be filled with the color that is defined for the middle of the region. Suppose you define a color gradient for only the middle half of a region, leaving the 25% at the left side and 25% at the right side undefined. The `NO_CYCLE` method will fill the left 25% region with the color that is defined at the 25% distance from left and the right 25% region with the color defined at the 25% distance from right. The color for the middle 50% will be determined by the color-stop points.

The cycle method of `REFLECT` fills the remaining regions by reflecting the color gradient, as start-to-end and end-to-start, from the nearest filled region. The cycle method of `REPEAT` repeats the color gradient to fill the remaining region.

The `stops` argument defines the color-stop points along the gradient line. A color-stop point is represented by an instance of the `Stop` class, which has only one constructor:

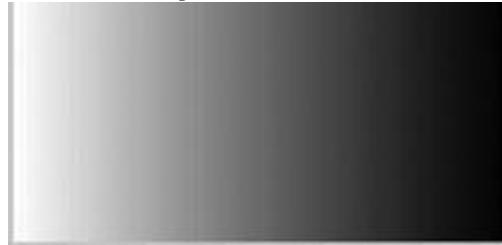
```
Stop(double offset, Color color)
```

The offset value is between 0.0 and 1.0. It defines the relative distance of the stop point along the gradient line from the starting point. For example, an offset of 0.0 is the starting point, an offset of 1.0 is the ending point, an offset of 0.5 is in the middle of the starting and ending points, and so forth. You define at least two stop points with two different colors to have a color gradient. There are no limits on the number of stop points you can define for a color gradient.

That covers the explanation for the arguments of the `LinearGradient` constructors. So let's look at some examples on how to use them.

The following snippet of code fills a rectangle with a linear color gradient, as shown in Figure 7-7:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 1, 0, true, NO_CYCLE, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```



**Figure 7-7.** A horizontal linear color gradient with two stop points: white at starting and black at ending point

You have two color-stop points. The stop point in the beginning is colored white and that of the end is colored black. The starting point (0, 0) and ending point (1, 0) define a horizontal gradient from left to right. The `proportional` argument is set to `true`, which means the coordinate values are interpreted as relative to a unit square. The cycle method argument, which is set to `NO_CYCLE`, has no effect in this case as your gradient bounds cover the entire region. In the above code, if you want to set the `proportional` argument value to `false`, to have the same effect, you would create the `LinearGradient` object as follows. Note the use of 200 as the `x` coordinate for the ending point to denote the end of the rectangle width:

```
LinearGradient lg = new LinearGradient(0, 0, 200, 0, false, NO_CYCLE, stops);
```

Let's look at another example. The resulting rectangle after running the following snippet of code is shown in Figure 7-8:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 0.5, 0, true, NO_CYCLE, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```

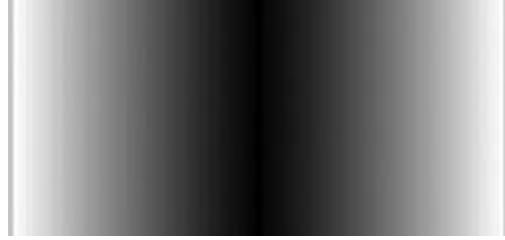


**Figure 7-8.** A horizontal linear color gradient with two stop points: white at starting and black at midpoint

In this code, you have made a slight change. You defined a horizontal gradient line, which starts at the left side of the rectangle and ends in the middle. Note the use of (0.5, 0) as the coordinates for the ending point. This leaves the right half of the rectangle with no color gradient. The cycle method is effective in this case as its job is to fill the unfilled regions. The color at the middle of the rectangle is black, which is defined by the second stop point. The NO\_CYCLE value uses the terminal black color to fill the right half of the rectangle.

Let's look at a slight variant of the previous example. You change the cycle method from NO\_CYCLE to REFLECT, as shown in the following snippet of code, which results in a rectangle as shown in Figure 7-9. Note that the right half region (the region with undefined gradient) is the reflection of the left half:

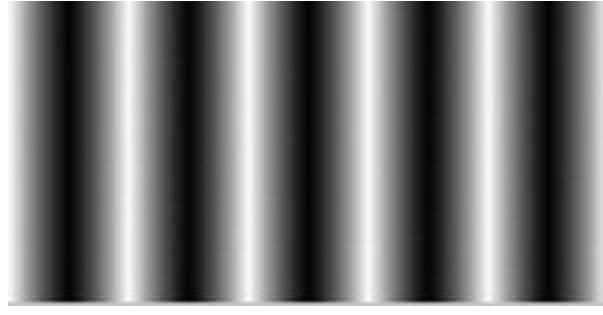
```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 0.5, 0, true, REFLECT, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```



**Figure 7-9.** A horizontal linear color gradient with two stop points: white at starting and black at midpoint and REFLECT as the cycle method

Let's make a slight change in the previous example so the ending point coordinate covers only one-tenth of the width of the rectangle. The code is as follows, and the resulting rectangle is shown in Figure 7-10. The right 90% of the rectangle is filled using the REFLECT cycle method by alternating end-to-start and start-to-end color patterns:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 0.1, 0, true, REFLECT, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```

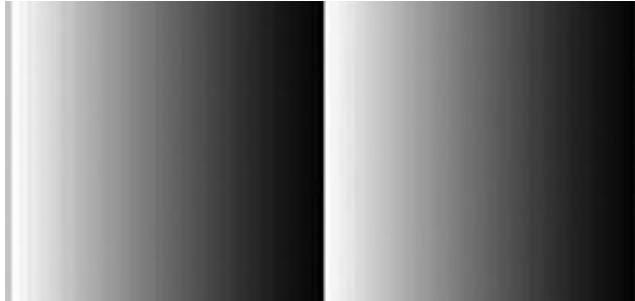


**Figure 7-10.** A horizontal linear color gradient with two stop points: white at starting and black at one-tenth point and REFLECT as the cycle method

Now let's look at the effect of using the REPEAT cycle method. The following snippet of code uses an ending point at the middle of the width of the rectangle and a cycle method of REPEAT. This results in a rectangle as shown in Figure 7-11. If you

set the ending point to one-tenth of the width in this example, it will result in a rectangle as shown in Figure 7-12.

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 0.5, 0, true, REPEAT, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```



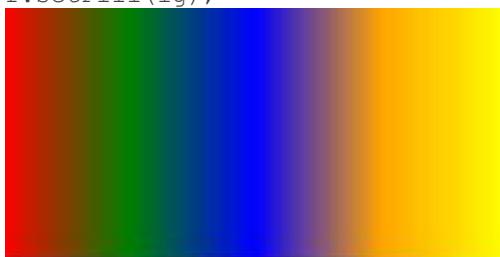
**Figure 7-11.** A horizontal linear color gradient with two stop points: white at starting and black at midpoint and REPEAT as the cycle method



**Figure 7-12.** A horizontal linear color gradient with two stop points: white at starting and black at one-tenth point and REPEAT as the cycle method

You could also define more than two stop points, as shown in the following snippet of code. It divides the distance between the starting and the ending points on the gradient line into four segments, each by 25% of the width. The first segment (from left) will have colors between red and green, the second between green and blue, the third between blue and orange, and the fourth between orange and yellow. The resulting rectangle is shown in Figure 7-13. If you are reading a printed copy of the book, you may not see the colors.

```
Stop[] stops = new Stop[]{new Stop(0, Color.RED),
                           new Stop(0.25, Color.GREEN),
                           new Stop(0.50, Color.BLUE),
                           new Stop(0.75, Color.ORANGE),
                           new Stop(1, Color.YELLOW)};
LinearGradient lg = new LinearGradient(0, 0, 1, 0, true, NO_CYCLE, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```



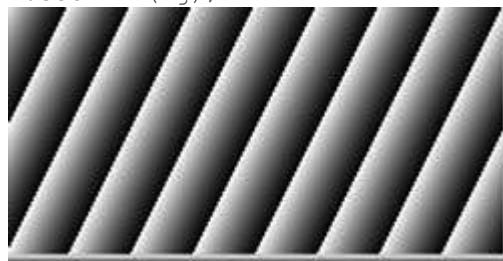
**Figure 7-13.** A horizontal linear color gradient with five stop points

You are not limited to defining only horizontal color gradients. You can define a color gradient with a gradient line with any angle. The following snippet of code creates a gradient from the top left corner to the bottom right corner. Note that when the proportional argument is true, (0, 0) and (1, 1) define the (x, y) coordinates of the top left and bottom right corners of the region:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 1, 1, true, NO_CYCLE, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```

The following snippet of code defines a gradient line between (0, 0) and (0.1, 0.1) points. It uses the REPEAT cycle method to fill the rest of the region. The resulting rectangle is shown in Figure 7-14.

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
LinearGradient lg = new LinearGradient(0, 0, 0.1, 0.1, true, REPEAT, stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(lg);
```



**Figure 7-14.** An angled linear color gradient with two stop points: white at the starting point (0, 0) and black at the ending point (0.1, 0.1) with REPEAT as the cycle method

## Defining Linear Color Gradients Using a String Format

You can also specify a linear color gradient in string format using the static method `valueOf(String colorString)` of the `LinearGradient` class. Typically, the string format is used to specify a linear color gradient in a CSS file. It has the following syntax:

```
linear-gradient([gradient-line], [cycle-method], color-stops-list)
```

The arguments within square brackets ([ and ]) are optional. If you do not specify an optional argument, the comma that follows also needs to be excluded. The default value for the gradient-line argument is “to bottom.” The default value for the cycle-method argument is `NO_CYCLE`. You can specify the gradient line in two ways:

- Using two points—the starting point and the ending point
- Using a side or a corner

The syntax for using two points for the gradient line is:

```
from point-1 to point-2
```

The coordinates of the points may be specified in percentage of the area or in actual measurement in pixels. For a 200px wide by 100px tall rectangle, a horizontal gradient line may be specified in the following two ways:

```
from 0% 0% to 100% 0%
```

or

```
from 0px 0px to 200px 0px
```

The syntax for using a side or a corner is:

```
to side-or-corner
```

The side-or-corner value may be top, left, bottom, right, top left, bottom left, bottom right, or top right. When you define the gradient line using a side or a corner, you specify only the ending point. The starting point is inferred. For example, the value “to top” infers the starting point as “from bottom”; the value “to bottom right” infers the starting point as “from top left,” and so forth. If the gradient-line value is missing, it defaults to “to bottom.”

The valid values for the `cycle-method` are `repeat` and `reflect`. If it is missing, it defaults to `NO_CYCLE`. It is a runtime error to specify the value of the `cycle-method` argument as `NO_CYCLE`. If you want it to be `NO_CYCLE`, simply omit the `cycle-method` argument from the syntax.

The `color-stops-list` argument is a list of color stops. A color stop consists of a web color name and, optionally, a position in pixels or percentage from the starting point. Examples of lists of color stops are:

- white, black
- white 0%, black 100%
- white 0%, yellow 50%, blue 100%
- white 0px, yellow 100px, red 200px

When you do not specify positions for the first and the last color stops, the positions for the first one defaults to 0% and the second one to 100%. So, the color stop lists "white, black" and "white 0%, black 100%" are fundamentally the same.

If you do not specify positions for any of the color stops in the list, they are assigned positions in such a way that they are evenly placed between the starting point and the ending point. The following two lists of color stops are the same:

- white, yellow, black, red, green
- white 0%, yellow 25%, black 50%, red 75%, green 100%

You can specify positions for some color stops in a list and not for others. In this case, the color stops without positions are evenly spaced between the preceding and following color stops with positions. The following two lists of color stops are the same:

- white, yellow, black 60%, red, green
- white 0%, yellow 30%, black 50%, red 80%, green 100%

If a color stop in a list has its position set less than the position specified for any previous color stops, its position is set equal to the maximum position set for the previous color stops. The following list of color stops sets 10% for the third color stop, which is less than the position of the second color stop (50%):

white, yellow 50%, black 10%, green

This will be changed at runtime to use 50% for the third color stop as follows:

white 0%, yellow 50%, black 50%, green 100%

Now let's look at some examples. The following string will create a linear gradient from top to bottom with `NO_CYCLE` as the cycle method. Colors are white and black at the top and bottom, respectively:

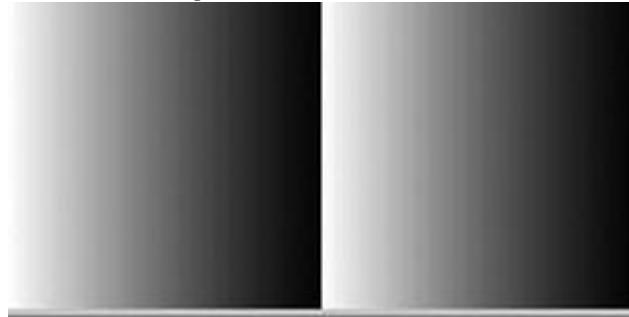
`linear-gradient(white, black)`

This value is the same as

```
linear-gradient(to bottom, white, black)
```

The following snippet of code will create a rectangle as shown in Figure 7-15. It defines a horizontal color gradient with the ending point midway through the width of the rectangle. It uses `repeat` as the cycle method:

```
String value = "from 0px 0px to 100px 0px, repeat, white 0%, black 100%";  
LinearGradient lg2 = LinearGradient.valueOf(value);  
Rectangle r2 = new Rectangle(200, 100);  
r2.setFill(lg2);
```



**Figure 7-15.** Creating a linear color gradient using the string format

The following string value for a linear color gradient will create a diagonal gradient from the top left corner to the bottom right corner filling the area with white and black colors:

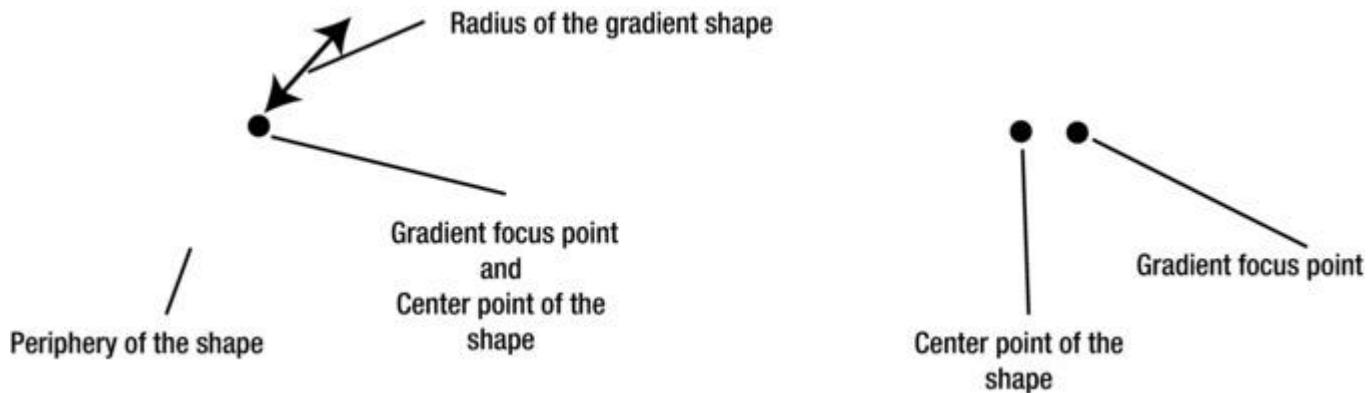
```
"to bottom right, white 0%, black 100%"
```

## Understanding Radial Color Gradient

In a radial color gradient, colors start at a single point, transitioning smoothly outward in a circular or elliptical shape. The shape, let's say a circle, is defined by a center point and a radius. The starting point of colors is known as the *focus point of the gradient*. The colors change along a line, starting at the focus point of the gradient, in all directions until the periphery of the shape is reached. A radial color gradient is defined using three components:

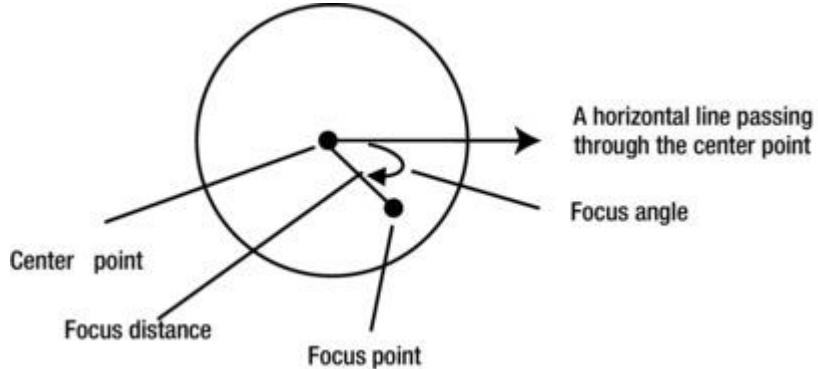
- A gradient shape (the center and radius of the gradient circle)
- A focus point that has the first color of the gradient
- Color stops

The focus point of the gradient and the center point of the gradient shape may be different. Figure 7-16 shows the components of a radial color gradient. The figure shows two radial gradients: In the left side, the focus point and the center point are located at the same place; in the right side, the focus point is located horizontally right to the center point of the shape.



**Figure 7-16.** Elements defining a radial color gradient

The focus point is defined in terms of a focus angle and a focus distance, as shown in Figure 7-17. The focus angle is the angle between a horizontal line passing through the center point of the shape and a line joining the center point and the focus point. The focus distance is the distance between the center point of the shape and the focus point of the gradient.



**Figure 7-17.** Defining a focus point in a radial color gradient

The list of color stops determines the value of the color at a point inside the gradient shape. The focus point defines the 0% position of the color stops. The points on the periphery of the circle define the 100% position for the color stops. How would you determine the color at a point inside the gradient circle? You would draw a line passing through the point and the focus point. The color at the point will be interpolated using the nearest color stops on each side of the point on the line.

### Using the *RadialGradient* Class

An instance of the *RadialGradient* class represents a radial color gradient. The class contains the following two constructors that differ in the types of their last argument:

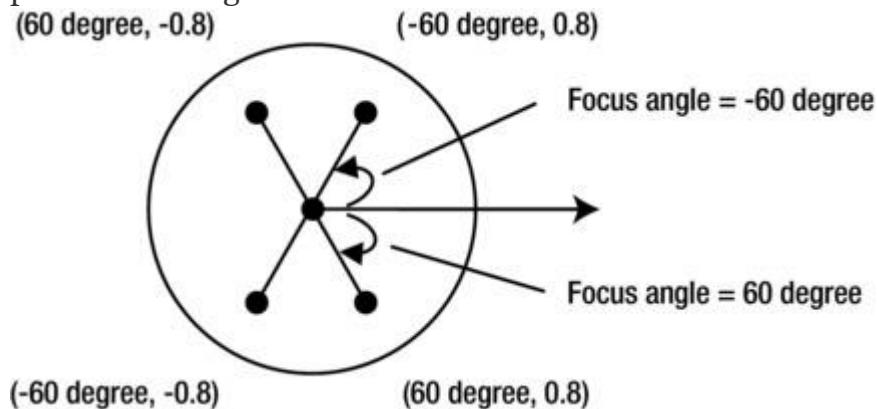
- `RadialGradient(double focusAngle, double focusDistance, double centerX, double centerY, double radius, boolean proportional, CycleMethod cycleMethod, List<Stop> stops)`
- `RadialGradient(double focusAngle, double focusDistance, double centerX, double centerY,`

```
double radius, boolean proportional, CycleMethod
cycleMethod, Stop... stops)
```

The `focusAngle` argument defines the focus angle for the focus point. A positive focus angle is measured clockwise from the horizontal line passing through the center point and the line connecting the center point and the focus point. A negative value is measured counterclockwise.

The `focusDistance` argument is specified in terms of the percentage of the radius of the circle. The value is clamped between -1 and 1. That is, the focus point is always inside the gradient circle. If the focus distance sets the focus point outside the periphery of the gradient circle, the focus point that is used is the point of intersection of the periphery of the circle and the line connecting the center point and the set focus point.

The focus angle and the focus distance can have positive and negative values. Figure 7-18 illustrates this: it shows four focus points located at 80% distance, positive and negative, from the center point and at a 60-degree angle, positive and negative.



*Figure 7-18. Locating a focus point with its focus angle and focus distance*

The `centerX` and `centerY` arguments define the x and y coordinates of the center point, respectively, and the `radius` argument is the radius of the gradient circle. These arguments can be specified relative to a unit square (between 0.0 and 1.0) or in pixels.

The `proportional` argument affects the way the values for the coordinates of the center point and radius are treated. If it is true, they are treated relative to a unit square. Otherwise, they are treated as absolute values in the local coordinate system. For more details on the use of the `proportional` argument, please refer to the section “Using the *LinearGradient Class*” earlier in this chapter.

**Tip** JavaFX lets you create a radial gradient of a circular shape. However, when the region to be filled by a radial color gradient has a nonsquare bounding box (e.g., a rectangle) and you specify the radius of the gradient circle relative to the size of the shape to be filled, JavaFX will use an elliptical radial color gradient. This is not documented in the API documentation of the `RadialGradient` class. I will present an example of this kind shortly.

The `cycleMethod` and `stops` arguments have the same meaning as described earlier in the section on using the `LinearGradient` class. In a radial color gradient, stops are defined along lines connecting the focus point and points on the periphery

of the gradient circle. The focus point defines the 0% stop point and the points on the circle periphery define 100% stop points.

Let's look at some examples of using the `RadialGradient` class. The following snippet of code produces a radial color gradient for a circle as shown in Figure 7-19:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 0.5, 0.5, 0.5, true, NO_CYCLE,
stops);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```



**Figure 7-19.** A radial color gradient with the same center point and focus point

The zero value for the focus angle and focus distance locates the focus point at the center of the gradient circle. A true proportional argument interprets the center point coordinates (0.5, 0.5) as (25px, 25px) for the 50 by 50 rectangular bounds of the circle. The radius value of 0.5 is interpreted as 25px, and that places the center of the gradient circle at the same location as the center of the circle to fill. The cycle method of `NO_CYCLE` has no effect in this case as the gradient circle fills the entire circular area. The color stop at the focus point is white and at the periphery of the gradient circle it is black.

The following snippet of code specifies the radius of the gradient circle as 0.2 of the circle to be filled. This means that it will use a gradient circle of 10px (0.2 multiplied by 50px, which is the radius of the circle to be filled). The resulting circle is shown in Figure 7-20. The region of the circle beyond the 0.2 of its radius has been filled with the color black, as the cycle method was specified as `NO_CYCLE`:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 0.5, 0.5, 0.2, true, NO_CYCLE,
stops);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```



**Figure 7-20.** A radial color gradient with the same center point and focus point having a gradient circle with a radius of 0.20

Now let's use the cycle method of REPEAT in the above snippet of code. The resulting circle is shown in Figure 7-21.

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 0.5, 0.5, 0.2, true, REPEAT,
stops);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```



**Figure 7-21.** A radial color gradient with the same center point and focus point, a gradient circle with a radius of 0.20, and the cycle method as REPEAT

So now let's use a different center point and focus point. Use a 60-degree focus angle and 0.2 times the radius as the focus distance as in the following code. The resulting circle is shown in Figure 7-22. Notice the 3D effect you get by moving the focus point away from the center point.

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(60, 0.2, 0.5, 0.5, 0.2, true, REPEAT,
stops);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```

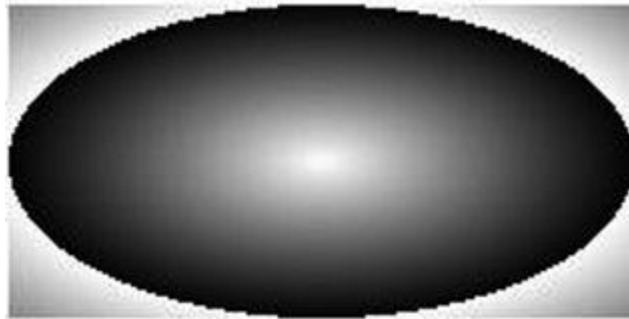


**Figure 7-22.** A radial color gradient using different center and focus points

Now let's fill a rectangular region (nonsquare) with a radial color gradient. The code for this effect follows and the resulting rectangle is shown in Figure 7-23. Notice the elliptical gradient shape used by JavaFX. You have specified the radius of the gradient as 0.5 and the proportional argument as true. Since your rectangle is 200px wide and 100px tall, it results in two radii: one along the x-axis and one along

the y-axis, giving rise to an ellipse. The radii along the x and y axes are 100px and 50px, respectively.

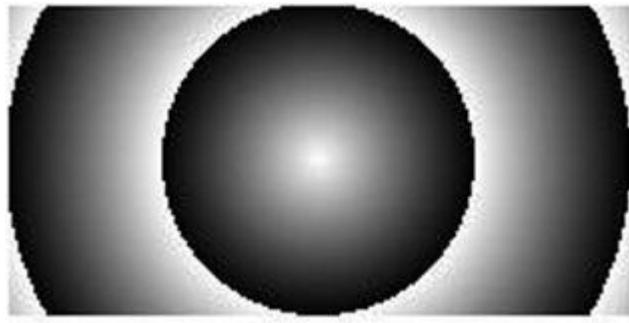
```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 0.5, 0.5, 0.5, true, REPEAT,
stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(rg);
```



**Figure 7-23.** A rectangle filled with a radial color gradient with a proportional argument value of true

If you want a rectangle to be filled with a color gradient of a circular shape rather than elliptical shape, you should specify the proportional argument as `false` and the radius value will be treated in pixels. The following snippet of code produces a rectangle, as shown in Figure 7-24:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 100, 50, 50, false, REPEAT,
stops);
Rectangle r = new Rectangle(200, 100);
r.setFill(rg);
```



**Figure 7-24.** A rectangle filled with a radial color gradient with a proportional argument value of false

How can you fill a triangle or any other shape with a radial color gradient? The shape of a radial gradient, circular or elliptical, depends on the several conditions. Table 7-1 shows the combinations of the criteria that will determine the shape of a radial color gradient.

**Table 7-1.** Criteria Used to Determine the Shape of a Radial Color Gradient

Proportional Argument	Bounding Box for the Filled Region	Gradient
true	Square	Circle

Proportional Argument	Bounding Box for the Filled Region	Gradient
true	Nonsquare	Ellipse
false	Square	Circle
false	Nonsquare	Circle

I should emphasize here that, in the above discussion, I am talking about the bounds of the regions to be filled, not the region. For example, suppose you want to fill a triangle with a radial color gradient. The bounds of the triangle will be determined by its width and height. If the triangle has the same width and height, its bounds take a square region. Otherwise, its bounds take a rectangular region.

The following snippet of code fills a triangle with vertices (0.0, 0.0), (0.0, 100.0), and (100.0, 100.0). Notice that the bounding box for this triangle is a 100px by 100px square. The resulting triangle is the left one shown in Figure 7-25.

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE), new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(0, 0, 0.5, 0.5, 0.2, true, REPEAT,
stops);
Polygon triangle = new Polygon(0.0, 0.0, 0.0, 100.0, 100.0, 100.0);
triangle.setFill(rg);
```



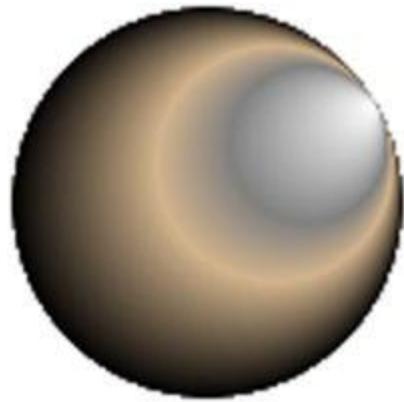
*Figure 7-25. Filling triangles with radial color gradients of circular and elliptical shapes*

The triangle in the right side of Figure 7-25 uses a rectangular bounding box of 200px by 100px, which is produced by the following snippet of code. Notice that the gradient uses an elliptical shape:

```
Polygon triangle = new Polygon(0.0, 0.0, 0.0, 100.0, 200.0, 100.0);
```

Finally, let's look at an example of using multiple color stops with the focus point on the periphery of the circle, as shown in Figure 7-26. The code to produce the effect is as follows:

```
Stop[] stops = new Stop[]{new Stop(0, Color.WHITE),
                        new Stop(0.40, Color.GRAY),
                        new Stop(0.60, Color.TAN),
                        new Stop(1, Color.BLACK)};
RadialGradient rg = new RadialGradient(-30, 1.0, 0.5, 0.5, 0.5, true, REPEAT,
stops);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```



**Figure 7-26.** Using multiple color stops in a radial color gradient

### Defining Radial Color Gradients in String Format

You can also specify a radial color gradient in string format by using the static method `valueOf(String colorString)` of the `RadialGradient` class. Typically, the string format is used to specify a radial color gradient in a CSS file. It has the following syntax:

```
radial-gradient([focus-angle], [focus-distance], [center], radius, [cycle-method], color-stops-list)
```

The arguments within square brackets are optional. If you do not specify an optional argument, the comma that follows needs to be excluded as well.

The default value for `focus-angle` and `focus-distance` is 0. You can specify the focus angle in degrees, radians, gradians, and turns. The focus distance is specified as a percentage of the radius. Examples are as follows:

- `focus-angle 45.0deg`
- `focus-angle 0.5rad`
- `focus-angle 30.0grad`
- `focus-angle 0.125turn`
- `focus-distance 50%`

The `center` and `radius` arguments are specified in a percentage relative to the region being filled or in absolute pixels. You cannot specify one argument in a percentage and the other in pixels. Both must be specified in the same unit. The default value for `center` is (0, 0) in the unit. Examples are as follows:

- `center 50px 50px, radius 50px`
- `center 50% 50%, radius 50%`

The valid values for the `cycle-method` argument are `repeat` and `reflect`. If this is not specified, it defaults to `NO_CYCLE`.

A list of color stops is specified using colors and their positions. Positions are specified as a percentage of distance on a line from the focus point to the periphery of the shape of the gradient. Please refer to the earlier discussion on specifying the color stops in a linear color gradient for more details. Examples are as follows:

- `white, black`
- `white 0%, black 100%`

- red, green, blue
- red 0%, green 80%, blue 100%

The following snippet of code will produce a circle, as shown in Figure 7-27:

```
String colorValue = "radial-gradient(focus-angle 45deg, focus-distance 50%, " +
    "center 50% 50%, radius 50%, white 0%, black 100%)";
RadialGradient rg = RadialGradient.valueOf(colorValue);
Circle c = new Circle(50, 50, 50);
c.setFill(rg);
```



**Figure 7-27.** Using string format for specifying a radial color gradient

## Summary

In JavaFX, you can specify text color and background color for regions. You can specify a color as a uniform color, an image pattern, or a color gradient. A uniform color uses the same color to fill the entire region. An image pattern lets you fill a region with an image pattern. A color gradient defines a color pattern in which the color varies along a straight line from one color to another. The variation in a color gradient can be linear or radial. All classes are included in the `javafx.scene.paint` package.

The `Paint` class is an abstract class and it is the base class for other color classes. A uniform color, an image pattern, a linear color gradient, and a radial color gradient are instances of the `Color`, `ImagePattern`, `LinearGradient`, and `RadialGradient` classes, respectively. The `Stop` class and the `CycleMethod` enum are used when working with color gradients. You can specify colors using instances of one of these classes or in string forms. When you use a CSS to style nodes, you specify colors using string forms.

An image pattern lets you fill a shape with an image. The image may fill the entire shape or use a tiling pattern.

A linear color gradient is defined using an axis known as a gradient line. Each point on the gradient line is of a different color. All points on a line that is perpendicular to the gradient line have the same color, which is the color of the point of intersection between the two lines. The gradient line is defined by a starting point and an ending point. Colors along the gradient line are defined at some points on the gradient line, which are known as stop-color points (or stop points). Colors between two stop points are computed using interpolation. The gradient line has a direction, which is from the starting point to the ending point. All points on a line perpendicular to the gradient line that passes through a stop point will have the color of the stop point. For example, suppose you have defined a stop point P1 with a color

C1. If you draw a line perpendicular to the gradient line passing through the point P1, all points on that line will have the color C1.

In a radial color gradient, colors start at a single point, transitioning smoothly outward in a circular or elliptical shape. The shape is defined by a center point and a radius. The starting point of colors is known as the focus point of the gradient. The colors change along a line, starting at the focus point of the gradient, in all directions until the periphery of the shape is reached.

The next chapter will show you how to style nodes in a scene graph using CSS.

## CHAPTER 8



### Styling Nodes

In this chapter, you will learn:

- What a cascading style sheets is
- The difference between styles, skins, and themes
- Naming conventions of cascading style sheets styles in JavaFX
- How to add style sheets to a scene
- How to use and override the default style sheet in a JavaFX application
- How to add inline styles for a node
- About the different types of cascading style sheet properties
- About cascading style sheets style selectors
- How to look up nodes in a scene graph using cascading style sheets selectors
- How to use compiled style sheets

#### What Is a Cascading Style Sheet?

A cascading style sheet (CSS) is a language used to describe the presentation (the look or the style) of UI elements in a GUI application. CSS was primarily developed for use in web pages for styling HTML elements. It allows for the separation of the presentation from the content and behavior. In a typical web page, the content and presentation are defined using HTML and CSS, respectively.

JavaFX allows you to define the look (or the style) of JavaFX applications using CSS. You can define UI elements using JavaFX class libraries or FXML and use CSS to define their look.

CSS provides the syntax to write rules to set the visual properties. A rule consists of a *selector* and a set of *property-valuepairs*. A selector is a string that identifies the UI elements to which the rules will be applied. A property-value pair consists of a property name and its corresponding value separated by a colon (:). Two property-value pairs are separated by a semicolon (;). The set of property-value pairs is enclosed within curly braces ({} ) preceded by the selector. An example of a rule in CSS is as follows:

```
.button {
    -fx-background-color: red;
    -fx-text-fill: white;
}
```

Here, .button is a selector, which specifies that the rule will apply to all buttons; `-fx-background-color` and `-fx-text-fill` are property names with their values set to `red` and `white`, respectively. When the above rule is applied, all buttons will have the red background color and white text color.

**Tip** Using CSS in JavaFX is similar to using CSS with HTML. If you have worked with CSS and HTML before, the information in this chapter will sound familiar. Prior experience with CSS is not necessary to understand how to use CSS in JavaFX. This chapter covers all of the necessary material to enable you to use CSS in JavaFX.

## What are Styles, Skins, and Themes?

A CSS rule is also known as a *style*. A collection of CSS rules is known as a *style sheet*. *Styles*, *skins*, and *themes* are three related, and highly confused, concepts.

*Styles* provide a mechanism to separate the presentation and content of UI elements. They also facilitate grouping of visual properties and their values, so they can be shared by multiple UI elements. JavaFX lets you create styles using JavaFX CSS.

*Skins* are collections of application-specific styles, which define the appearance of an application. *Skinning* is the process of changing the appearance of an application (or the skin) on the fly. JavaFX does not provide a specific mechanism for skinning. However, using the JavaFX CSS and JavaFX API, available for the `Scene` class and other UI-related classes, you can provide skinning for your JavaFX application easily.

*Themes* are visual characteristics of an operating system that are reflected in the appearance of UI elements of all applications. For example, changing the theme on the Windows operating system changes the appearance of UI elements in all applications that are running. To contrast skins and themes, skins are application specific, whereas themes are operating system specific. It is typical to base skins on themes. That is, when the current theme is changed, you would change the skin of an application to match the theme. JavaFX has no direct support for themes.

## A Quick Example

Let's look at a simple, though complete, example of using style sheets in JavaFX. You will set the background color and text color of all buttons to red and white, respectively. The code for the styles is shown in Listing 8-1.

### **Listing 8-1.** The Content of the File buttonstyles.css

```
.button {
    -fx-background-color: red;
    -fx-text-fill: white;
}
```

Save the content of Listing 8-1 in a `buttonstyles.css` file under the `resources\css` directory. You can place the file in any other directory; however, make sure that you change the file path accordingly. Finally, place the `resources\css` directory in the application CLASSPATH.

A scene contains an `ObservableList` of string URLs of styles sheets. You can get the reference of the `ObservableList` using the `getStylesheets()` method of the `Scene` class. The following snippet of code adds the URL for the `buttonstyles.css` style sheet to the scene:

```
Scene scene;
...
scene.getStylesheets().add("resources/css/buttonstyles.css");
```

Listing 8-2 contains the complete program, which shows three buttons with a red background and white text. If you get the following warning message and do not see the buttons in red background with white text, it indicates that you have not placed the `resources\css` directory in the CLASSPATH.

WARNING: com.sun.javafx.css.StyleManager loadStylesheetUnPrivileged Resource "resources/css/buttonstyles.css" not found.

If one of the CLASSPATH entries is C:\abc\xyz, you need to place the buttonstyles.css file under the C:\abc\xyz\resources\css directory.

### **Listing 8-2.** Using a Style Sheet to Change the Background and Text Colors for Buttons

```
// ButtonStyleTest.java
package com.jdojo.style;

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

public class ButtonStyleTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button yesBtn = new Button("Yes");
        Button noBtn = new Button("No");
        Button cancelBtn = new Button("Cancel");

        HBox root = new HBox();
        root.getChildren().addAll(yesBtn, noBtn, cancelBtn);

        Scene scene = new Scene(root);

        // Add a style sheet to the scene
        scene.getStylesheets().add("resources/css/buttonstyles.css");

        stage.setScene(scene);
        stage.setTitle("Styling Buttons");
        stage.show();
    }
}
```

### **Naming Conventions in JavaFX CSS**

JavaFX uses slightly different naming conventions for the CSS style classes and properties. CSS style class names are based on the simple names of the JavaFX classes representing the node in a scene graph. All style class names are lowercased. For example, the style class name is `button` for the `Button` class. If the class name for the JavaFX node consists of multiple words, for example, `TextField`, a hyphen is inserted between two words to get the style class name. For example, the style classes for the `TextField` and `CheckBox` classes are `text-field` and `checkbox`, respectively.

**Tip** It is important to understand the difference between a JavaFX class and a CSS style class. A JavaFX class is a Java class, for example, `javafx.scene.control.Button`. A CSS style class is used as a selector in a style sheet, for example, `button` in Listing 8-1.

Property names in JavaFX styles start with `-fx-`. For example, the property name `font-size` in normal CSS styles becomes `-fx-font-size` in JavaFX CSS style. JavaFX uses a convention to map the style property names to the instance variables. It takes an instance variable; it inserts a hyphen between two words; if the

instance variable consists of multiple words, it converts the name to the lowercase and prefixes it with `-fx-`. For example, for an instance variable named `textAlignment`, the style property name would be `-fx-text-alignment`.

## Adding Style Sheets

You can add multiple style sheets to a JavaFX application. Style sheets are added to a scene or parents. Scene and Parent classes maintain an observable list of string URLs linking to style sheets. Use the `getStylesheets()` method in the Scene and Parent classes to get the reference of the observable list and add additional URLs to the list. The following code would accomplish this:

```
// Add two style sheets, ss1.css and ss2.css to a scene
Scene scene = ...
scene.getStylesheets().addAll("resources/css/ss1.css",
"resources/css/ss2.css");

// Add a style sheet, vbox.css, to a VBox (a Parent)
VBox root = new VBox();
root.getStylesheets().add("vbox.css");
```

How are the string URLs for a style sheet resolved? You can specify a style sheet URL in three forms:

- A relative URL, for example, "resources/css/ss1.css"
- An absolute URL with no scheme or authority, for example, "/resources/css/ss1.css"
- An absolute URL, for example, "http://jdojo.com/resources/css/ss1.css" and "file:///C:/css/ss2.css"

The first two types of URLs are resolved the same way. They are resolved relative to the base URL of the `ClassLoader` of the concrete class that extends the `Application` class. This needs a little explanation. Suppose you have a `com.jdojo.style.FXApp` class, which extends the `Application` class and it is the main application class for your JavaFX application. To resolve the style sheets URLs correctly, you need to place your style sheet files in the same directory that contains the `com/jdojo/style/FXApp.class` file.

If you have problems accessing your style sheets using the above technique, you can use the absolute URLs. You can also use class's `getResource()` method or the `ClassLoader` of a class to get the URL of your style sheet. The following snippet of code uses the base URL of the `ClassLoader` of the `Test` class to resolve the relative URL of the style sheet:

```
Scene scene;
...
String urlString = Test.class.getClassLoader()
    .getResource("resources/css/hjfx.css")
    .toExternalForm();
scene.getStylesheets().add(urlString);
```

## Default Style Sheet

In previous chapters you developed JavaFX applications with UI elements without the use of any style sheets. However, JavaFX runtime was always using a style sheet behind the scenes. The style sheet is named `Modena.css`, which is known as

the *default style sheet* or the *user-agent style sheet*. The default look that you get for a JavaFX application is defined in the default style sheet.

The `modena.css` file is packaged in the JavaFX runtime `jfxrt.jar` file. If you want to know the details of how styles are set for specific nodes, you need to look at the `modena.css` file. You can extract this file using the following command:

```
jar -xf jfxrt.jar com/sun/javafx/scene/control/skin/modena/modena.css
```

This command places the `modena.css` file in the `com\sun\javafx\scene\control\skin\modena` directory under the current directory. Note that the `jar` command is in the `JAVA_HOME\bin` directory.

Prior to JavaFX 8, Caspian was the default style sheet. Caspian is defined in the `jfxrt.jar` file in the file named `com/sun/javafx/scene/control/skin/caspian/caspian.css`. In JavaFX 8, Modena is the default style sheet. The `Application` class defines two String constants named `STYLESTHEET_CASPION` and `STYLESTHEET_MODENA` to represent the two themes. Use the following static methods of the `Application` class to set and get the application-wide default style sheet:

- `public static void setUserAgentStylesheet(String url)`
- `public static String getUserAgentStylesheet()`

Use the `setUserAgentStylesheet(String url)` method to set an application-wide default. A value of `null` will restore the platform default style sheet, for example, Modena on JavaFX 8 and Caspian on the prior versions. The following statement sets Caspian as the default style sheet:

```
Application.setUserAgentStylesheet(Application.STYLESTHEET_CASPION);
```

Use the `getUserAgentStylesheet()` method to return the current default style sheet for the application. If one of the built-in style sheet is the default, it returns `null`.

## Adding Inline Styles

CSS styles for a node in a scene graph may come from style sheets or an inline style. In the previous section, you learned how to add style sheets to the `Scene` and `Parent` objects. In this section, you will learn how to specify an inline style for a node.

The `Node` class has a `style` property that is of `StringProperty` type. The `style` property holds the inline style for a node. You can use the `setStyle(String inlineStyle)` and `getStyle()` methods to set and get the inline style of a node.

There is a difference between a style in a style sheet and an inline style. A style in a style sheet consists of a selector and a set of property-value pairs, and it may affect zero or more nodes in a scene graph. The number of nodes affected by a style in a style sheet depends on the number of nodes that match the selector of the style. An inline style does not contain a selector. It consists of only set property-value pairs. An inline style affects the node on which it is set. The following snippet of code uses an inline style for a button to display its text in red and bold:

```
Button yesBtn = new Button("Yes");
yesBtn.setStyle("-fx-text-fill: red; -fx-font-weight: bold;");
```

**Listing 8-3** displays six buttons. It uses two `VBox` instances to hold three buttons. It places both `VBox` instances into an `HBox`. Inline styles are used to set a 4.0px blue border for both `VBox` instances. The inline style for the `HBox` sets a 10.0px navy border. The resulting screen is shown in Figure 8-1.

### ***Listing 8-3.*** Using Inline Styles

```
// InlineStyles.java
package com.jdojo.style;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class InlineStyles extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button yesBtn = new Button("Yes");
        Button noBtn = new Button("No");
        Button cancelBtn = new Button("Cancel");

        // Add an inline style to the Yes button
        yesBtn.setStyle("-fx-text-fill: red; -fx-font-weight: bold;");

        Button openBtn = new Button("Open");
        Button saveBtn = new Button("Save");
        Button closeBtn = new Button("Close");

        VBox vb1 = new VBox();
        vb1.setPadding(new Insets(10, 10, 10, 10));
        vb1.getChildren().addAll(yesBtn, noBtn, cancelBtn);

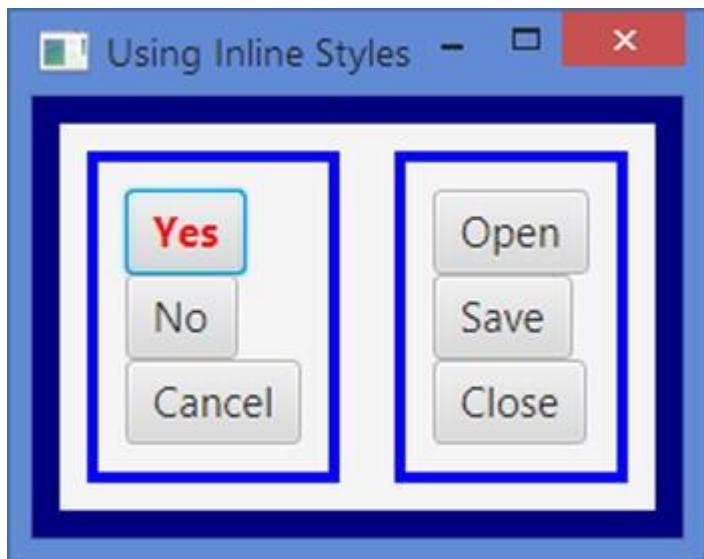
        VBox vb2 = new VBox();
        vb2.setPadding(new Insets(10, 10, 10, 10));
        vb2.getChildren().addAll(openBtn, saveBtn, closeBtn);

        // Add a border to VBoxes using an inline style
        vb1.setStyle("-fx-border-width: 4.0; -fx-border-color: blue;");
        vb2.setStyle("-fx-border-width: 4.0; -fx-border-color: blue;");

        HBox root = new HBox();
        root.setSpacing(20);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.getChildren().addAll(vb1, vb2);

        // Add a border to the HBox using an inline style
        root.setStyle("-fx-border-width: 10.0; -fx-border-color: navy;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Inline Styles");
        stage.show();
    }
}
```



**Figure 8-1.** A button, two `VBox` instances, and an `HBox` using inline styles

### Priorities of Styles for a Node

In a JavaFX application, it is possible, and very common, for the visual properties of nodes to come from multiple sources. For example, the font size of a button can be set by the JavaFX runtime, style sheets can be added to the parent and the scene of the button, an inline style can be set for the button, and programmatically can be added using the `setFont(Font f)` method. If the value for the font size of a button is available from multiple sources, JavaFX uses a rule to decide the source whose value is to be used.

Consider the following snippet of code along with the `stylespriorities.css` style sheet whose content is shown in Listing 8-4:

```
Button yesBtn = new Button("Yes");
yesBtn.setStyle("-fx-font-size: 16px");
yesBtn.setFont(new Font(10));

Scene scene = new Scene(yesBtn);
scene.getStylesheets().addAll("resources/css/stylespriorities.css");
...
```

### **Listing 8-4.** The Content of `stylespriorities.css` File

```
.button {
    -fx-font-size: 24px;
    -fx-font-weight: bold;
}
```

What will be the font size of the button? Will it be the default font size set by the JavaFX runtime, 24px, declared in the `stylespriorities.css`, 16px set by the inline style, or 10px set by the program using the `setFont()` method? The correct answer is 16px, which is set by the inline style.

The JavaFX runtime uses the following priority rules to set the visual properties of a node. The source with a higher priority that has a value for a property is used:

- Inline style (the highest priority)
- Parent style sheets

- Scene style sheets
- Values set in the code using JavaFX API
- User agent style sheets (the lowest priority)

The style sheet added to the parent of a node is given higher priority than the style sheets added to the scene. This enables developers to have custom styles for different branches of the scene graph. For example, you can use two style sheets that set properties of buttons differently: one for buttons in the scene and one for buttons in any `HBox`. Buttons in an `HBox` will use styles from its parent, whereas all other buttons will use styles from the scene.

The values set using the JavaFX API, for example, `setFont()` method, have the second lowest priority.

**Note** It is a common mistake to set the same properties of a node in a style sheet and code using the Java API. In that case, the styles in the style sheet win and developers spend countless hours trying to find the reasons why the properties set in the code are not taking effect.

The lowest priority is given to style sheets used by the user agent. What is a user agent? A user agent, in general, is a program that interprets a document and applies style sheets to the document to format, print, or read. For example, a web browser is a user agent that applies default formatting to HTML documents. In our case, the user agent is the JavaFX runtime, which uses the `caspian.css` style sheet for providing the default look for all UI nodes.

**Tip** The default font size that is inherited by nodes is determined by the system font size. Not all nodes use fonts. Fonts are used by only those nodes that display text, for example, a `Button` or a `CheckBox`. To experiment with the default font, you can change the system font and check it in code using the `getFont()` method of those nodes.

**Listing 8-5** demonstrates the priority rules for choosing a style from multiple sources. It adds the style sheet, as shown in Listing 8-4, to the scene. The resulting screen is shown in Figure 8-2.

### **Listing 8-5.** Testing Priorities of Styles for a Node

```
// StylesPriorities.java
package com.jdojo.style;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class StylesPriorities extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button yesBtn = new Button("Yes");
        Button noBtn = new Button("No");
        Button cancelBtn = new Button("Cancel");
    }
}
```

```

        // Change the font size for the Yes button
        // using two methods: inline style and JavaFX API
        yesBtn.setStyle("-fx-font-size: 16px");
        yesBtn.setFont(new Font(10));

        // Change the font size for the No button using the JavaFX API
        noBtn.setFont(new Font(8));

        HBox root = new HBox();
        root.setSpacing(10);
        root.getChildren().addAll(yesBtn, noBtn, cancelBtn);

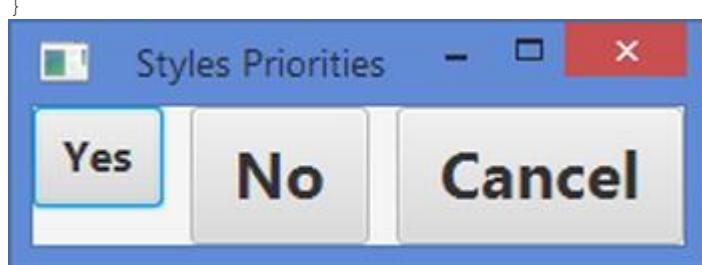
        Scene scene = new Scene(root);

        // Add a style sheet to the scene
        scene.getStylesheets().addAll("resources/css/stylespriorities.css");
    }

}

}


```



**Figure 8-2.** Nodes using styles from different sources

The font size value for the Yes button comes from four sources:

- Inline style (16px)
- Style sheet added to the scene (24px)
- JavaFX API (10px)
- Default font size set by the user agent (the JavaFX runtime)

The Yes button gets a 16px font size from its inline style, because that has the highest priority. The font size value for the No button comes from three sources:

- Style sheet added to the scene (24px)
- JavaFX API (10px)
- Default font size set by the user agent (the JavaFX runtime)

The No button gets a 24px font size from the style sheet added to the scene because that has the highest priority among the three available sources.

The font size value for the Cancel button comes from two sources:

- Style sheet added to the scene (24px)
- Default font size set by the user agent (the JavaFX runtime)

The Cancel button gets a 24px font size from the style sheet added to the scene, because that has the highest priority between the two available sources. The text for

all buttons are shown in bold, because you have used the "-fx-font-weight: bold;" style in the style sheet and this property value is not overridden by any other sources.

At this point, several questions may arise in your mind:

- How do you let the `Cancel` button use the default font size that is set by the JavaFX runtime?
- How do you use one font size (or any other properties) for buttons if they are inside an `HBox` and use another font size if they are inside a `VBox`?

You can achieve all these and several other effects using appropriate selectors for a style declared in a style sheet. I will discuss different types of selectors supported by JavaFX CSS shortly.

## Inheriting CSS Properties

JavaFX offers two types of inheritance for CSS properties:

- Inheritance of CSS property types
- Inheritance of CSS property values

In the first type of inheritance, all CSS properties declared in a JavaFX class are inherited by all its subclasses. For example, the `Node` class declares a `cursor` property and its corresponding CSS property is `-fx-cursor`. Because the `Node` class is the superclass of all JavaFX nodes, the `-fx-cursor` CSS property is available for all node types.

In the second type of inheritance, a CSS property for a node may inherit its value from its parent. The parent of a node is the container of the node in the scene graph, not its JavaFX superclass. The values of some properties of a node are inherited from its parent by default, and for some, the node needs to specify explicitly that it wants to inherit the values of the properties from its parent.

You can specify `inherit` as the value for a CSS property of a node if you want the value to be inherited from its parent. If a node inherits a CSS property from its parent by default, you do not need to do anything, that is, you do not even need to specify the property value as `inherit`. If you want to override the inherited value, you need to specify the value explicitly (overriding the parent's value).

Listing 8-6 demonstrates how a node inherits the CSS properties of its parent. It adds two buttons, OK and Cancel, to `HBox`. The following CSS properties are set on the parent and the OK button. No CSS properties are set on the Cancel button:

```
/* Parent Node (HBox)*/
-fx-cursor: hand;
-fx-border-color: blue;
-fx-border-width: 5px;

/* Child Node (OK Button)*/
-fx-border-color: red;
-fx-border-width: inherit;
```

The `-fx-cursor` CSS property is declared in the `Node` class and is inherited by all nodes by default. The `HBox` overrides the default value and overrides it to the `HAND` cursor. Both the OK and Cancel buttons inherit the `HAND` cursor value for

their `-fx-cursor` from their parent, `HBox`. When you point your mouse to the area occupied by the `HBox` and these buttons, your mouse pointer will change to a `HAND` cursor. You can use the "`-fx-cursor: inherit`" style on the `OK` and `Cancel` buttons to achieve the same functionality you get by default.

Border-related CSS properties are not inherited by nodes by default. The `HBox` sets its `-fx-border-color` to blue and `-fx-border-width` to `5px`. The `OK` button sets its `-fx-border-color` to red and `-fx-border-width` to `inherit`. The `inherit` value will make the `-fx-border-width` of the `OK` button to inherit from its parent (the `HBox`), which is `5px`. Figure 8-3 shows the changes after adding this coding.

### *****Listing 8-6.***** Inheriting CSS Properties from the Parent Node

```
// CSSInheritance.java
package com.jdojo.style;

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

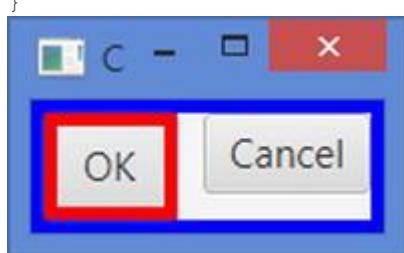
public class CSSInheritance extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        HBox root = new HBox(10); // 10px spacing
        root.getChildren().addAll(okBtn, cancelBtn);

        // Set styles for the OK button and its parent HBox
        root.setStyle("-fx-cursor: hand;-fx-border-color: blue;-fx-
border-width: 5px;");
        okBtn.setStyle("-fx-border-color: red;-fx-border-width:
inherit");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("CSS Inheritance");
        stage.show();
    }
}
```



**Figure 8-3.** A button inheriting its border width and cursor CSS properties from its parent

**Tip** A node inherits `-fx-cursor`, `-fx-text-alignment`, and `-fx-font` CSS properties from its parent by default.

## Types of CSS Properties

All values in Java (and in JavaFX as well) have a type. The values of CSS properties set in styles also have types. Each type of value has a different syntax. JavaFX CSS supports the following types:

- `inherit`
- `boolean`
- `string`
- `number`
- `angle`
- `point`
- `color-stop`
- `URI`
- `effect`
- `font`
- `paint`

Note that the CSS types have nothing to do with Java types. They can only be used in specifying the values in CSS style sheets or inline styles. The JavaFX runtime takes care of parsing and converting these types to appropriate JavaFX types before assigning them to nodes.

### The *inherit* Type

You have seen an example of the use of the `inherit` type in the previous section. It is used to inherit the value of a CSS property for a node from its parent.

### The *boolean* Type

You can specify the `boolean` type values as `true` or `false`. They can also be specified as strings: `"true"` or `"false"`. The following style sets the `-fx-display-caret` CSS property of a `TextField` node to `false`:

```
.text-field {
    -fx-display-caret: false;
}
```

### The *string* Type

String values can be enclosed in single quotes or double quotes. If the string value is enclosed in double quotes, a double quote as part of the value should be escaped, such as `\"` or `\22`. Similarly, a single quote as part of the string value enclosed in single quotes must be escaped, such as `\'` or `\27`. The following style uses strings to set the `skin` and `font` properties. It encloses the `string` value for the `skin` property in double quotes and the `font` family for the `font` property in single quotes:

```
.my-control {
    -fx-skin: "com.jdojo.MySkin";
    -fx-font: normal bold 20px 'serif';
}
```

**Tip** A string value cannot contain a newline directly. To embed a newline in a string value, use the escape sequence \Aor \00000a.

### The *number* Type

Number values may be represented as integers or real numbers. They are specified using the decimal number format. The following style sets the opacity to 0.60:

```
.my-style {
    -fx-opacity: 0.60;
}
```

The value of a CSS property denoting a size can be specified using a number following by a unit of length. The unit of length can be px (pixels), mm (millimeters), cm (centimeters), in (inches), pt (points), pc (picas ), em, or ex. A size can also be specified using the percentage of a length, for example, the width or height of a node. If a unit of a percentage is specified, it must immediately follow the number, for example, 12px, 2em, 80%:

```
.my-style {
    -fx-font-size: 12px;
    -fx-background-radius: 0.5em;
    -fx-border-width: 5%;
}
```

### The *angle* Type

An angle is specified using a number and a unit. The unit of an angle can be deg (degrees), rad (radians), grad (gradients), or turn (turns). The following style sets the -fx-rotate CSS property to 45 degrees:

```
.my-style {
    -fx-rotate: 45deg;
}
```

### The *point* Type

A point is specified using x and y coordinates. It can be specified using two numbers separated by whitespaces, for example, 0 0, 100, 0, 90 67, or in percentage form, for example, 2% 2%. The following style specifies a linear gradient color from the point (0, 0) to (100, 0):

```
.my-style {
    -fx-background-color: linear-gradient(from 0 0 to 100 0, repeat, red,
blue);
```

### The *color-stop* Type

A color-stop is used to specify color at a specific distance in linear or radial color gradients. A color-stop consists of a color and a stop distance. The color and the distance are separated by whitespaces. The stop distance may be specified as a percentage, for example 10%, or as a length, for example, 65px. Some examples of color-stops are white 0%, yellow 50%, yellow 100px. Please refer to Chapter 7 for more details on how to use color-stops in colors.

### The *URI* Type

A URI can be specified using the `url(<address>)` function. A relative `<address>` is resolved relative to the location of the CSS file:

```
.image-view {  
    -fx-image: url("http://jdojo.com/myimage.png");  
}
```

### The *effect* Type

Drop shadow and inner shadow effects can be specified for nodes using CSS styles using the `dropshadow()` and `innershadow()` CSS functions, respectively. Their signature are:

- `dropshadow(<blur-type>, <color>, <radius>, <spread>, <x-offset>, <y-offset>)`
- `innershadow(<blur-type>, <color>, <radius>, <choke>, <x-offset>, <y-offset>)`

The `<blur-type>` value can be Gaussian, one-pass-box, three-pass-box, or two-pass-box. The color of the shadow is specified in `<color>`. The `<radius>` value specifies the radius of the shadow blur kernel between 0.0 and 127.0. The spread/choke of the shadow is specified between 0.0 and 1.0. The last two parameters specify the shadow offsets in pixels in x and y directions. The following styles show how to specify the values for the `-fx-effect` CSS property:

```
.drop-shadow-1 {  
    -fx-effect: dropshadow(gaussian, gray, 10, 0.6, 10, 10);  
}  
  
.drop-shadow-2 {  
    -fx-effect: dropshadow(one-pass-box, gray, 10, 0.6, 10, 10);  
}  
  
.inner-shadow-1 {  
    -fx-effect: innershadow(gaussian, gray, 10, 0.6, 10, 10);  
}
```

### The *font* Type

A font consists of four attributes: family, size, style, and weight. There are two ways to specify the font CSS property:

- Specify the four attributes of a font separately using the four CSS properties: `-fx-font-family`, `-fx-font-size`, `-fx-font-style`, and `-fx-font-weight`.
- Use a shorthand CSS property `-fx-font` to specify all four attributes as one value.

The font family is a string value that can be the actual font family available on the system, for example, "Arial", "Times", or generic family names, for example, "serif", "sans-serif", "monospace".

The font size can be specified in units such as px, em, pt, in, cm. If the unit for the font size is omitted, px (pixels) is assumed.

The font style can be normal, italic, or oblique.

The font weight can be specified as `normal`, `bold`, `bolder`, `lighter`, `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, or `900`.

The following style sets the font attributes separately:

```
.my-font-style {  
    -fx-font-family: "serif";  
    -fx-font-size: 20px;  
    -fx-font-style: normal;  
    -fx-font-weight: bolder;  
}
```

Another way to specify the font property is to combine all four attributes of the font into one value and use the `-fx-font` CSS property. The syntax for using the `-fx-font` property is:

```
-fx-font: <font-style> <font-weight> <font-size> <font-family>;
```

The following style uses the `-fx-font` CSS property to set the font attributes:

```
.my-font-style {  
    -fx-font: italic bolder 20px "serif";  
}
```

### The *paint* Type

A paint type value specifies a color, for example, the fill color of a rectangle or the background color of a button. You can specify a color value in the following ways:

- Using the `linear-gradient()` function
- Using the `radial-gradient()` function
- Using various color values and color functions

Please refer to Chapter 7 for a complete discussion on how to specify gradient colors in string format using the `linear-gradient()` and `radial-gradient()` functions. These functions are used to specify color gradients. The following style shows how to use these functions:

```
.my-style {  
    -fx-fill: linear-gradient(from 0% 0% to 100% 0%, black 0%, red 100%);  
    -fx-background-color: radial-gradient(radius 100%, black, red);  
}
```

You can specify a solid color in several ways:

- Using named colors
- Using looked-up colors
- Using the `rgb()` and `rgba()` functions
- Using red, green, blue (RGB) hexadecimal notation
- Using the `hsb()` or `hsba()` function
- Using color functions: `derive()` and `ladder()`

You can use predefined color names to specify the color values, for example, `red`, `blue`, `green`, or `aqua`:

```
.my-style {  
    -fx-background-color: red;  
}
```

You can define a color as a CSS property on a node or any of its parents and, later, look it up by name, when you want to use its value. The following styles define a color named `my-color` and refer to it later:

```
.root {  
    my-color: black;  
}  
  
.my-style {  
    -fx-fill: my-color;  
}
```

You can use the `rgb(red, green, blue)` and the `rgba(red, green, blue, alpha)` functions to define colors in terms of RGB components:

```
.my-style-1 {  
    -fx-fill: rgb(0, 0, 255);  
}  
  
.my-style-2 {  
    -fx-fill: rgba(0, 0, 255, 0.5);  
}
```

You can specify a color value in the `#rrggbb` or `#rgb` format, where `rr`, `gg`, and `bb` are the values for red, green, and blue components, respectively, in hexadecimal format. Note that you need to specify the three components using two digits or one hexadecimal digit. You cannot specify some components in one hexadecimal digit and others in two:

```
.my-style-1 {  
    -fx-fill: #0000ff;  
}  
  
.my-style-2 {  
    -fx-fill: #0bc;  
}
```

You can specify a color value in hue, saturation, brightness (HSB) color components using the `hsb(hue, saturation, brightness)` or `hsba(hue, saturation, brightness, alpha)` function:

```
.my-style-1 {  
    -fx-fill: hsb(200, 70%, 40%);  
}  
  
.my-style-2 {  
    -fx-fill: hsba(200, 70%, 40%, 0.30);  
}
```

You can compute colors from other colors using the `derive()` and `ladder()` functions. The JavaFX default CSS, `caspian.css`, uses this technique. It defines some base colors and derives other colors from the base colors.

The `derive` function takes two parameters:

```
derive(color, brightness)
```

The `derive()` function derives a brighter or darker version of the specified color. The brightness value ranges from -100% to 100%. A brightness of -100% means completely black, 0% means no change in brightness, and 100% means completely white. The following style will use a version of red that is 20% darker:

```
.my-style {  
    -fx-fill: derive(red, -20%);  
}
```

The `ladder()` function takes a color and one or more color-stops as parameters:

```
ladder(color, color-stop-1, color-stop-2, ...)
```

Think of the `ladder()` function as creating a gradient using the color-stops and then using the brightness of the specified color to return the color value. If the brightness of the specified color is  $x\%$ , the color at the  $x\%$  distance from the

beginning of the gradient will be returned. For example, for 0% brightness, the color at the 0.0 end of the gradient is returned; for 40% brightness, the color at the 0.4 end of the gradient is returned.

Consider the following two styles:

```
.root {
    my-base-text-color: red;
}

.my-style {
    -fx-text-fill: ladder(my-base-text-color, white 29%, black 30%);
}
```

The `ladder()` function will return the color `white` or `black` depending on the brightness of the `my-base-text-color`. If its brightness is 29% or lower, `white` is returned; otherwise, `black` is returned. You can specify as many color-stops as you want in the `ladder()` function to choose from a variety of colors depending on the brightness of the specified color.

You can use this technique to change the color of a JavaFX application on the fly. The default style sheet, `caspian.css`, defines some base colors and uses the `derive()` and `ladder()` functions to derive other colors of different brightnesses. You need to redefine the base colors in your style sheet for the `root` class to make an application-wide color change.

## Specifying Background Colors

A node (a `Region` and a `Control`) can have multiple background fills, which are specified using three properties:

- `-fx-background-color`
- `-fx-background-radius`
- `-fx-background-insets`

The `-fx-background-color` property is a list of comma-separated color values. The number of colors in the list determines the number of rectangles that will be painted. You need to specify the radius values for four corners and insets for four sides, for each rectangle, using the other two properties. The number of color values must match the number of radius values and inset values.

The `-fx-background-radius` property is a list of a comma-separated set of four radius values for the rectangles to be filled. A set of radius values in the list may specify only one value, for example, `10`, or four values separated by whitespaces, for example, `10 5 15 20`. The radius values are specified for the top-left, top-right, bottom-right, and bottom-left corners in order. If only one radius value is specified, the same radius value is used for all corners.

The `-fx-background-insets` property is a list of a comma-separated set of four inset values for the rectangles to be filled. A set of inset values in the list may specify only one value, for example, `10`, or four values separated by whitespaces, for example, `10 5 15 20`. The inset values are specified for the top, right, bottom, and left sides in order. If only one inset value is specified, the same inset value is used for all sides.

Let's look at an example. The following snippet of code creates a `Pane`, which is a subclass of the `Region` class:

```
Pane pane = new Pane();
pane.setPrefSize(100, 100);
```

Figure 8-4 shows how the `Pane` looks when the following three styles are supplied:

```
.my-style-1 {
    -fx-background-color: gray;
    -fx-background-insets: 5;
    -fx-background-radius: 10;
}

.my-style-2 {
    -fx-background-color: gray;
    -fx-background-insets: 0;
    -fx-background-radius: 0;
}

.my-style-3 {
    -fx-background-color: gray;
    -fx-background-insets: 5 10 15 20;
    -fx-background-radius: 10 0 0 5;
}
```



`my-style-1`



`my-style-2`



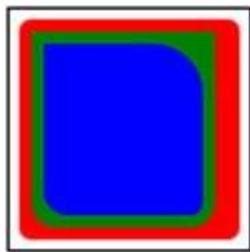
`my-style-3`

**Figure 8-4.** A `Pane` with three different background fills

All three styles use a gray fill color, which means that only one rectangle will be drawn. The first style uses a 5px inset on all four sides, and a radius of 10px for all corners. The second style uses a 0px inset and a 0px radius, which makes the fill rectangle occupy the entire area of the pane. The third style uses a different inset on each side: 5px on the top, 10px on the right, 15px on the bottom, and 20px on the left. Notice the different unfilled background on each side for the third style. The third style also sets different values for the radius of four corners: 10px for the top-left, 0px for the top-right, 0px for the bottom-right, and 5px for the bottom-left. Notice that if the radius of a corner is 0px, the two sides at the corner meet at 90 degrees.

If you apply the following style to the same pane, the background will be filled as shown in Figure 8-5:

```
.my-style-4 {
    -fx-background-color: red, green, blue;
    -fx-background-insets: 5 5 5 5, 10 15 10 10, 15 20 15 15;
    -fx-background-radius: 5 5 5 5, 0 0 10 10, 0 20 5 10;
}
```



**Figure 8-5.** A pane with three background fills with different radius and inset values

The style uses three colors and, therefore, three background rectangles will be painted. The background rectangles are painted in the order they are specified in the style: red, green, and blue. The inset and radius values are specified in the same order as the colors. The style uses the same value for insets and radii for the red color. You can replace the set of four similar values with one value; that is, 5 5 5 5 in the above style can be replaced with 5.

## Specifying Borders

A node (a Region and a Control) can have multiple borders through CSS. A border is specified using five properties:

- `-fx-border-color`
- `-fx-border-width`
- `-fx-border-radius`
- `-fx-border-insets`
- `-fx-border-style`

Each property consists of a comma-separated list of items. Each item may consist of a set of values, which are separated by whitespaces.

### Border Colors

The number of items in the list for the `-fx-border-color` property determines the number of borders that are painted. The following style will paint one border with the red color:

```
-fx-border-color: red;
```

The following style specifies a set of red, green, blue, and aqua colors to paint the borders on top, right, bottom, and left sides, respectively. Note that it still results in only one border, not four borders, with different colors on four sides:

```
-fx-border-color: red green blue aqua;
```

The following style specifies two sets of border colors:

```
-fx-border-color: red green blue aqua, tan;
```

The first set consists of four colors, red green blue aqua, and the second set consists of only one color, tan. It will result in two borders. The first border will be painted with different colors on four sides; the second border will use the same color on all four sides.

**Tip** A node may not be rectangular in shape. In that case, only the first border color (and other properties) in the set will be used to paint the entire border.

### Border Widths

You can specify the width for borders using the `-fx-border-width` property. You have an option to specify different widths for all four sides of a border. Different border widths are specified for top, right, bottom, and left sides in order. If the unit for the width value is not specified, pixel is used.

The following style specifies one border with all sides painted in red in 2px width:

```
-fx-border-color: red;  
-fx-border-width: 2;
```

The following style specifies three borders, as determined by the three sets of colors specified in the `-fx-border-color` property. The first two borders use different border widths of four sides. The third border uses the border width of 3px on all sides:

```
-fx-border-color: red green blue black, tan, aqua;  
-fx-border-width: 2 1 2 2, 2 2 2 1, 3;
```

## Border Radii

You can specify the radius values for four corners of a border using the `-fx-border-radius` property. You can specify the same radius value for all corners. Different radius values are specified for top-left, top-right, bottom-right, and bottom-left corners in order. If the unit for the radius value is not specified, pixel is used.

The following style specifies one border in red, 2px width, and 5px radii on all four corners:

```
-fx-border-color: red;  
-fx-border-width: 2;  
-fx-border-radius: 5;
```

The following style specifies three borders. The first two borders use different radius values for four corners. The third border uses the radius value of 0px for all corners:

```
-fx-border-color: red green blue black, tan, aqua;  
-fx-border-width: 2 1 2 2, 2 2 2 1, 3;  
-fx-border-radius: 5 2 0 2, 0 2 0 1, 0;
```

## Border Insets

You can specify the inset values for four sides of a border using the `-fx-border-insets` property. You can specify the same inset value for all sides. Different inset values are specified for top, right, bottom, and left sides in order. If the unit for the inset value is not specified, pixel is used.

The following style specifies one border in red, 2px width, 5px radius, and 20px inset on all four sides:

```
-fx-border-color: red;  
-fx-border-width: 2;  
-fx-border-radius: 5;  
-fx-border-insets: 20;
```

The following style specifies three borders with insets 10px, 20px, and 30px on all sides:

```
-fx-border-color: red green blue black, tan, aqua;  
-fx-border-width: 2 1 2 2, 2 2 2 1, 3;  
-fx-border-radius: 5 2 0 2, 0 2 0 1, 0;  
-fx-border-insets: 10, 20, 30;
```

**Tip** An inset is the distance from the side of the node at which the border will be painted. The final location of the border also depends of other properties, for example, `-fx-border-width` and `-fx-border-style`.

## Border Styles

The `-fx-border-style` property defines the style of a border. Its value may contain several parts as follows:

```
-fx-border-style: <dash-style> [<phase <number>>] [<stroke-type>] [<line-join>
<line-join-value>] [<line-cap <line-cap-value>>]
```

The value for `<dash-style>` can be none, solid, dotted, dashed, or segments (`<number>`, `<number>...`). The value for `<stroke-type>` can be centered, inside, or outside. The value for `<line-join-value>` can be miter `<number>`, bevel, or round. The value for `<line-cap-value>` can be square, butt, or round.

The simplest border style would be to specify just the value for the `<dash-style>`:

```
-fx-border-style: solid;
```

The `segments()` function is used to have a border with a pattern using alternate dashes and gaps:

```
-fx-border-style: segments(dash-length, gap-length, dash-length, ...);
```

The first argument to the function is the length of the dash; the second argument is the length of the gap, and so on. After the last argument, the pattern repeats itself from the beginning. The following style will paint a border with a pattern of a 10px dash, a 5px gap, a 10px dash, and so on:

```
-fx-border-style: segments(10px, 5px);
```

You can pass as many dashes and gap segments to the function as you want. The function expects you to pass an even number of values. If you pass an odd number of values, this will result in values that are concatenated to make them even in number. For example, if you use `segments(20px, 10px, 5px)`, it is the same as if you passed `segments(20px, 10px, 5px, 20px, 10px, 5px)`.

The `phase` parameter is applicable only when you use the `segments()` function. The number following the `phase` parameter specifies the offset into the dashed pattern that corresponds to the beginning of the stroke. Consider the following style:

```
-fx-border-style: segments(20px, 5px) phase 10.0;
```

It specifies the `phase` parameter as 10.0. The length of the dashing pattern is 25px. The first segment will start at 10px from the beginning of the pattern. That is, the first dash will only be 10px in length. The second segment will be a 5px gap followed by a 20px dash, and so on. The default value for `phase` is 0.0.

The `<stroke-type>` has three valid values: centered, inside, and outside. Its value determines where the border is drawn relative to the inset. Assume that you have a 200px by 200px region. Assume that you have specified the top inset as 10px and a top border width of 4px. If `<stroke-type>` is specified as centered, the border thickness at the top will occupy the area from the eighth pixel to the 12th pixel from the top boundary of the region. For `<stroke-type>` as inside, the border thickness will occupy the area from the 10th pixel to 14th pixel. For `<stroke-type>` as outside, the border thickness at the top will occupy the area from the sixth pixel to the tenth pixel.

You can specify how the two segments of the borders are joined using the `line-join` parameter. Its value can be miter, bevel, or round. If you specify the value of `line-join` as miter, you need to pass a miter limit value. If the specified miter limit is less than the miter length, a bevel join is used instead. Miter length is the distance between the inner point and the outer point of a miter join. Miter length is

measured in terms of the border width. The miter limit parameter specifies how far the outside edges of two meeting border segments can extend to form a miter join. For example, suppose the miter length is 5 and you specify the miter limit as 4, a bevel join is used; however, if you specify a miter limit greater than 5, a miter join is used. The following style uses a miter limit of 30:

```
-fx-border-style: solid line-join miter 30;
```

The value for the `line-cap` parameter specifies how the start and end of a border segment are drawn. The valid values are `square`, `butt`, and `round`. The following style specified a `line-cap` of `round`:

```
-fx-border-style: solid line-join bevel 30 line-cap round;
```

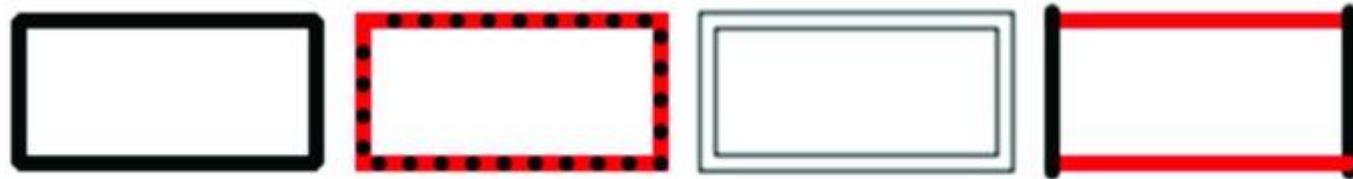
Let's look at some examples. Figure 8-6 shows four instances of the `Pane` class of 100px by 50px, when the following styles are applied to them:

```
.my-style-1 {
    -fx-border-color: black;
    -fx-border-width: 5;
    -fx-border-radius: 0;
    -fx-border-insets: 0;
    -fx-border-style: solid line-join bevel line-cap square;
}

.my-style-2 {
    -fx-border-color: red, black;
    -fx-border-width: 5, 5;
    -fx-border-radius: 0, 0;
    -fx-border-insets: 0, 5;
    -fx-border-style: solid inside, dotted outside;
}

.my-style-3 {
    -fx-border-color: black, black;
    -fx-border-width: 1, 1;
    -fx-border-radius: 0, 0;
    -fx-border-insets: 0, 5;
    -fx-border-style: solid centered, solid centered;
}

.my-style-4 {
    -fx-border-color: red black red black;
    -fx-border-width: 5;
    -fx-border-radius: 0;
    -fx-border-insets: 0;
    -fx-border-style: solid line-join bevel line-cap round;
}
```



*Figure 8-6. Using border styles*

Notice that the second style achieves overlapping of two borders, one in solid red and one in dotted black, by specifying the appropriate insets and stroke type (`inside` and `outside`). Borders are drawn in the order they are specified. It is important that you draw the solid border first in this case; otherwise, you would not

see the dotted border. The third one draws two borders, giving it the look of a double border type.

**Tip** A Region can also have a background image and a border image specified through CSS. Please refer to the *JavaFX CSS Reference Guide*, which is available online, for more details. Many other CSS styles are supported by nodes in JavaFX. The styles for those nodes will be discussed later in this book.

## Understanding Style Selectors

Each style in a style sheet has an associated *selector* that identifies the nodes in the scene graph to which the associated JavaFX CSS property values are applied. JavaFX CSS supports several types of selectors: class selectors, pseudo-class selectors, ID selectors, among others. Let's look at some of these selector types briefly.

### Using Class Selectors

The `Node` class defines a `styleClass` variable that is an `ObservableList<String>`. Its purpose is to maintain a list of JavaFX style class names for a node. Note that the JavaFX class name and the style class name of a node are two different things. A JavaFX class name of a node is a Java class name, for example, `javafx.scene.layout.VBox`, or simply `VBox`, which is used to create objects of that class. A style class name of a node is a string name that is used in CSS styling.

You can assign multiple CSS class names to a node. The following snippet of code assigns two style class names, "hbox" and "myhbox", to an `HBox`:

```
HBox hb = new HBox();
hb.getStyleClass().addAll("hbox", "myhbox");
```

A style class selector applies the associated style to all nodes, which have the same style class name as the name of the selector. A style class selector starts with a period followed by the style class name. Note that the style class names of nodes do not start with a period.

**Listing 8-7** shows the content of a style sheet. It has two styles. Both styles use style class selectors because both of them start with a period. The first style class selector is ".hbox", which means it will match all nodes with a style class named `hbox`. The second style uses the style class name as `button`. Save the style sheet in a file named `resources\css\styleclass.css` in the CLASSPATH.

### **Listing 8-7.** A Style Sheet with Two Style Class Selectors Named `hbox` and `button`

```
.hbox {
    -fx-border-color: blue;
    -fx-border-width: 2px;
    -fx-border-radius: 5px;
    -fx-border-insets: 5px;
    -fx-padding: 10px;
    -fx-spacing: 5px;
    -fx-background-color: lightgray;
    -fx-background-insets: 5px;
}

.button {
    -fx-text-fill: blue;
}
```

**Listing 8-8** has the complete program to demonstrate the use of the style class selectors `hbox` and `button`. The resulting screen is shown in Figure 8-7.

### **Listing 8-8.** Using Style Class Selectors in Code

```
// StyleClassTest.java
package com.jdojo.style;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class StyleClassTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

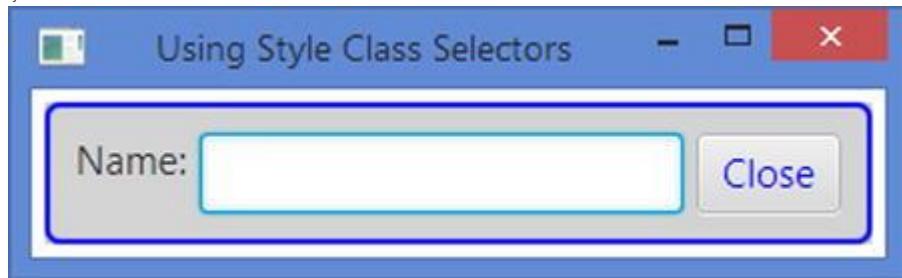
    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Name:");
        TextField nameTf = new TextField("");
        Button closeBtn = new Button("Close");
        closeBtn.setOnAction(e -> Platform.exit());

        HBox root = new HBox();
        root.getChildren().addAll(nameLbl, nameTf, closeBtn);

        // Set the styleClass for the HBox to "hbox"
        root.getStyleClass().add("hbox");

        Scene scene = new Scene(root);
        scene.getStylesheets().add("resources/css/styleclass.css");

        stage.setScene(scene);
        stage.setTitle("Using Style Class Selectors");
        stage.show();
    }
}
```



**Figure 8-7.** An `HBox` using border, padding, spacing, and background color from a style sheet

Notice that you have set the style class name for the `HBox` (named `root` in the code) to "`hbox`", which will apply CSS properties to the `HBox` from the style with the class selector `hbox`. The text color of the `Close` button is blue because of the second style with the style class selector `button`. You did not set the style class name for the `Close` button to "`button`". The `Button` class adds a style class, which is

named "button", to all its instances. This is the reason that the `Closebutton` was selected by the `button` style class selector.

Most of the commonly used controls in JavaFX have a default style class name. You can add more style class names if needed. The default style class names are constructed from the JavaFX class names. The JavaFX class name is converted to lowercase and a hyphen is inserted in the middle of two words. If the JavaFX class name consists of only one word, the corresponding default style class name is created by just converting it to lowercase. For example, the default style class name is `button` for `Button`, `label` for `Label`, `hyperlink` for `Hyperlink`, `text-field` for `TextField`, `text-area` for `TextArea`, `check-box` for `CheckBox`.

JavaFX container classes, for example, `Region`, `Pane`, `HBox`, `VBox`, do not have a default style class name. If you want to style them using style class selectors, you need to add a style class name to them. This is the reason that you had to add a style class name to the `HBox` that you used in Listing 8-8 to use the style class selector.

**Tip** Style class names in JavaFX are case-sensitive.

Sometimes you might need to know the default style class name of a node to use it in a style sheet. There are three ways to determine the default style class name of a JavaFX node:

- Guess it using the described rules to form the default style class name from the JavaFX class name;
- Use the online *JavaFX CSS Reference Guide* to look up the name;
- Write a small piece of code.

The following snippet of code shows how to print the default style class name for the `Button` class. Change the name of the JavaFX node class, for example, from `Button` to `TextField`, to print the default style class name for other types of nodes:

```
Button btn = new Button();
ObservableList<String> list = btn.getStyleClass();

if (list.isEmpty()) {
    System.out.println("No default style class name");
} else {
    for(String styleClassName : list) {
        System.out.println(styleClassName);
    }
}
button
```

### Class Selector for the root Node

The `root` node of a scene is assigned a style class named "root". You can use the `root` style class selector for CSS properties that are inherited by other nodes. The `root` node is the parent of all nodes in a scene graph. Storing CSS properties in the `root` node is preferred because they can be looked up from any node in the scene graph.

Listing 8-9 shows the content of a style sheet saved in a file `resources\css\rootclass.css`. The style with the `rootclass` selector declares two properties: `-fx-cursor` and `-my-button-color`. The `-fx-cursor` property is inherited by all nodes. If this style sheet is attached to a scene, all

nodes will have a HAND cursor unless they override it. The `-my-button-color` property is a look-up property, which is looked up in the second style to set the text color of buttons.

### ***Listing 8-9.*** The Content of the Style Sheet with Root as a Style Class Selector

```
.root {
    -fx-cursor: hand;
    -my-button-color: blue;
}

.button {
    -fx-text-fill: -my-button-color;
}
```

Run the program in Listing 8-10 to see the effects of these changes. Notice that you get a HAND cursor when you move the mouse anywhere in the scene, except over the name text field. This is because the `TextField` class overrides the `-fx-cursor` CSS property to set it to the TEXT cursor.

### ***Listing 8-10.*** Using the Root Style Class Selector

```
// RootClassTest.java
package com.jdojo.style;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class RootClassTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Name:");
        TextField nameTf = new TextField("");
        Button closeBtn = new Button("Close");

        HBox root = new HBox();
        root.getChildren().addAll(nameLbl, nameTf, closeBtn);

        Scene scene = new Scene(root);
        /* The root variable is assigned a default style class name
        "root" */

        scene.getStylesheets().add("resources/css/rootclass.css");

        stage.setScene(scene);
        stage.setTitle("Using the root Style Class Selector");
        stage.show();
    }
}
```

## Using ID Selectors

The `Node` class has an `id` property of the `StringProperty` type, which can be used to assign a unique `id` to each node in a scene graph. Maintaining the uniqueness of an `id` in a scene graph is the responsibility of the developer. It is not an error to set a duplicate `id` for a node.

You do not use the `id` property of a node directly in your code, except when you are setting it. It is mainly used for styling nodes using ID selectors. The following snippet of code sets the `id` property of a `Button` to "closeBtn":

```
Button b1 = new Button("Close");
b1.setId("closeBtn");
```

An ID selector in a style sheet is preceded by the pound (#) sign. Note that the ID value set for a node does not include the # sign. Listing 8-11 shows the content of a style sheet, which contains two styles, one with a class selector ".button" and one with an ID selector "#closeButton". Save the content of Listing 8-11 in a file called `resources\css\idselector.css` in the CLASSPATH. Figure 8-8 shows the results after the program is run.

### ***Listing 8-11.*** A Style Sheet that Uses a Class Selector and an ID Selector

```
.button {
    -fx-text-fill: blue;
}

#closeButton {
    -fx-text-fill: red;
}
```

Listing 8-12 presents the program that uses the style sheet in Listing 8-11. The program creates three buttons. It sets the ID for a button to "closeButton". The other two buttons do not have an ID. When the program is run, the Closebutton's text is in red, whereas the other two have blue text.

### ***Listing 8-12.*** Using ID Selector in a Style Sheet

```
// IDSelectorTest.java
package com.jdojo.style;

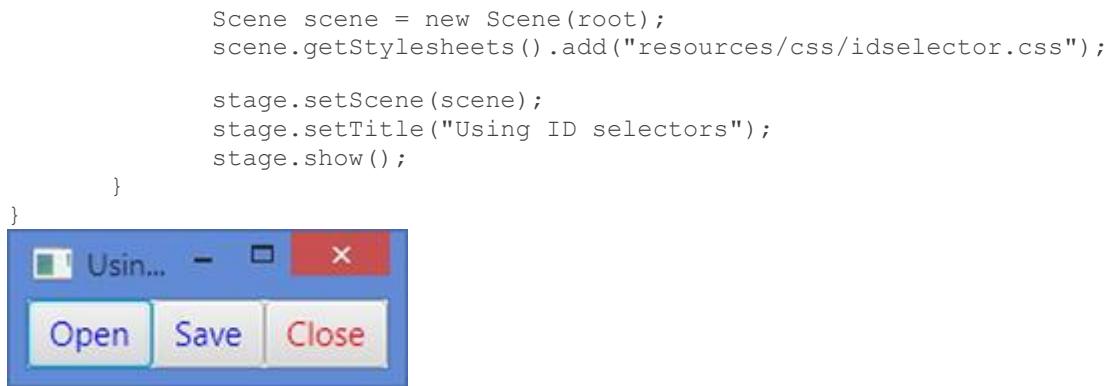
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class IDSelectorTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button openBtn = new Button("Open");
        Button saveBtn = new Button("Save");

        Button closeBtn = new Button("Close");
        closeBtn.setId("closeButton");

        HBox root = new HBox();
        root.getChildren().addAll(openBtn, saveBtn, closeBtn);
    }
}
```



**Figure 8-8.** Buttons using class and ID selectors

Did you notice a conflict in the styles for the `Close` button? All buttons in JavaFX are assigned a default style class named `button`, so does the `Close` button. The `Close` button also has an ID that matches with the ID style selector. Therefore, both selectors in the style sheet match the `Close` button. In cases where there are multiple selectors matching a node, JavaFX uses the *specificity of selectors* to determine which selector will be used. In cases where a class selector and an ID selector are used, the ID selector has higher specificity. This is the reason that the ID selector matched the `Close` button, not the class selector.

**Tip** CSS uses complex rules to calculate the specificity of selectors. Please refer to <http://www.w3.org/TR/CSS21/cascade.html#specificity> for more details.

### Combining ID and Class Selectors

A selector can use the combination of a style class and an ID. In this case, the selector matches all nodes with the specified style class and ID. Consider the following style:

```
#closeButton.button {
    -fx-text-fill: red;
}
```

The selector `#closeButton.button` matches all nodes with a `closeButton` ID and a `button` style class. You can also reverse the order:

```
.button#closeButton {
    -fx-text-fill: red;
}
```

Now it matches all nodes with a `button` style class and a `closeButton` ID.

### The Universal Selector

An asterisk (\*) is used as a universal selector, which matches any node. The universal selector has the lowest specificity. The following style uses the universal selector to set the text fill property of all nodes to blue:

```
* {
    -fx-text-fill: blue;
}
```

When the universal selector does not appear by itself, it can be ignored. For example, the selectors `*.button` and `.button` are the same.

### Grouping Multiple Selectors

If the same CSS properties apply to multiple selectors, you have two choices:

- You can use multiple styles by duplicating the property declarations.
- You can group all selectors into one style, separating the selectors by a comma.

Suppose you want to set the `button` and `label` classes text fill color to blue. The following code uses two styles with the duplicate property declarations:

```
.button {
    -fx-text-fill: blue;
}

.label {
    -fx-text-fill: blue;
}
```

The two styles can be combined into one style as follows:

```
.button, .label {
    -fx-text-fill: blue;
}
```

## Descendant Selectors

A descendant selector is used to match nodes that are descendants of another node in the scene graph. A descendant selector consists of two or more selectors separated by whitespaces. The following style uses a descendant selector:

```
.hbox .button {
    -fx-text-fill: blue;
}
```

It will select all nodes that have a `button` style class and are descendants of a node with an `hbox` style class. The term *descendant* in this context means a child at any level (immediate or nonimmediate).

A descendant selector comes in handy when you want to style parts of JavaFX controls. Many controls in JavaFX consist of subnodes, which are JavaFX nodes. In the *JavaFX CSS Reference Guide*, those subnodes are listed as substructures. For example, a `CheckBox` consists of a `LabeledText` (not part of the public API) with a style class name of `text` and a `StackPane` with a style class name of `box`.

The `box` contains another `StackPane` with the style class name of `mark`. You can use these pieces of information for the substructure of the `CheckBox` class to style the subparts. The following styles use descendant selectors to set the text color of all `CheckBox` instances to blue and the box to a dotted border:

```
.check-box .text {
    -fx-fill: blue;
}

.check-box .box {
    -fx-border-color: black;
    -fx-border-width: 1px;
    -fx-border-style: dotted;
}
```

## Child Selectors

A child selector matches a child node. It consists of two or more selectors separated by the greater than sign (`>`). The following style matches all nodes with a `button` style class, which are the children of a node with an `hbox` style class:

```
.hbox > .button {
    -fx-text-fill: blue;
}
```

**Tip** CSS supports other types of selectors, for example, sibling selectors and attribute selectors. JavaFX CSS does not support them yet.

## State-Based Selectors

State-based selectors are also known as *pseudo-class* selectors. A pseudo-class selector matches nodes based on their current states, for example, matching a node that has focus or matching text input controls that are read-only. A pseudo-class is preceded by a colon and is appended to an existing selector. For example, `.button:focused` is a pseudo-class selector that matches a node with the `button` style class name that also has the focus; `#openBtn:hover` is another pseudo-class selector that matches a node with the ID `#openBtn`, when the mouse hovers over the node. Listing 8-13 presents the content of a style sheet that has a pseudo-class selector. It changes the text color to red when the mouse hovers over the node. When you add this style sheet to a scene, all buttons will change their text color to red when the mouse hovers over them.

### **Listing 8-13.** A Style Sheet with a Pseudo-class Selector

```
.button:hover {
    -fx-text-fill: red;
}
```

JavaFX CSS does not support the `:first-child` and `:lang` pseudo-classes that are supported by CSS. JavaFX does not support *pseudo-elements* that allow you to style the content of nodes (e.g., the first line in a `TextArea`). Table 8-1 contains a partial list of the pseudo-classes supported by JavaFX CSS. Please refer to the online *JavaFX CSS Reference Guide* for the complete list of pseudo-classes supported by JavaFX CSS.

**Table 8-1.** Some Pseudo-classes Supported by JavaFX CSS

Pseudo-class	Applies to	Description
disabled	Node	It applies when the node is disabled.
focused	Node	It applies when the node has the focus.
hover	Node	It applies when the mouse hovers over the node.
pressed	Node	It applies when the mouse button is clicked over the node.
show-mnemonic	Node	It applies when the mnemonic should be shown.

Pseudo-class	Applies to	Description
cancel	Button	It applies when the Button would receive VK_ESC if the event was consumed.
default	Button	It applies when the Button would receive VK_ENTER if the event was consumed.
empty	Cell	It applies when the Cell is empty.
filled	Cell	It applies when the Cell is not empty.
selected	Cell, CheckBox	It applies when the node is selected.
determinate	CheckBox	It applies when the CheckBox is in a determinate state.
indeterminate	CheckBox	It applies when the CheckBox is in an indeterminate state.
visited	Hyperlink	It applies when the Hyperlink has been visited.
horizontal	ListView	It applies when the node is horizontal.
vertical	ListView	It applies when the node is vertical.

### Using JavaFX Class Names as Selectors

It is allowed, but not recommended, to use the JavaFX class name as a type selector in a style. Consider the following content of a style sheet:

```
HBox {
    -fx-border-color: blue;
    -fx-border-width: 2px;
    -fx-border-insets: 10px;
    -fx-padding: 10px;
}

Button {
    -fx-text-fill: blue;
}
```

Notice that a type selector differs from a class selector in that the former does not start with a period. A class selector is the JavaFX class name of the node without any modification (HBOX and HBox are not the same). If you attach a style sheet with the above content to a scene, all HBox instances will have a border and all Button instances will have blue text.

It is not recommended to use the JavaFX class names as type selectors because the class name may be different when you subclass a JavaFX class. If you depend on the class name in your style sheet, the new classes will not pick up your styles.

## Looking Up Nodes in a Scene Graph

You can look up a node in a scene graph by using a selector. `Scene` and `Node` classes have a `lookup(String selector)` method, which returns the reference of the first node found with the specified `selector`. If no node is found, it returns `null`. The methods in two classes work a little differently. The method in the `Scene` class searches the entire scene graph. The method in the `Node` class searches the node on which it is called and its subnodes. The `Node` class also has a `lookupAll(String selector)` method that returns a `Set` of all `Nodes` that are matched by the specified `selector`, including the node on which this method is called and its subnode.

The following snippet of code shows how to use the look-up methods using ID selectors. However, you are not limited to using only ID selectors in these methods. You can use all selectors that are valid in JavaFX:

```
Button b1 = new Button("Close");
b1.setId("closeBtn");
VBox root = new VBox();
root.setId("myvbox");
root.getChildren().addAll(b1);
Scene scene = new Scene(root, 200, 300);
...
Node n1 = scene.lookup("#closeBtn");           // n1 is the reference of b1
Node n2 = root.lookup("#closeBtn");           // n2 is the reference of b1
Node n3 = b1.lookup("#closeBtn");           // n3 is the reference of b1
Node n4 = root.lookup("#myvbox");           // n4 is the reference of root
Node n5 = b1.lookup("#myvbox");           // n5 is null
Set<Node> s = root.lookupAll("#closeBtn"); // s contains the reference of b1
```

## Using Compiled Style Sheets

When packaging JavaFX projects, you can convert the CSS files into binary form to improve the runtime performance of your application. The `.css` files are converted to `.bss` files. You can convert CSS files into binary form using the `javafxpackager` tool with `-createbss` command:

```
javafxpackager -createbss -srcfiles mystyles.css -outdir compiledcss
```

If you are using the NetBeans IDE, you can select the Project Properties ▶ Build ▶ Packaging ▶ Binary Encode JavaFX CSS Files property, which will convert all your CSS files into BSS files while packaging your project.

## Summary

CSS is a language used to describe the presentation of UI elements in a GUI application. It was primarily used in web pages for styling HTML elements and separating presentation from contents and behavior. In a typical web page, the content and presentation are defined using HTML and CSS, respectively.

JavaFX allows you to define the look of JavaFX applications using CSS. You can define UI elements using JavaFX class libraries or FXML, and use CSS to define their look.

A CSS rule is also known as a style. A collection of CSS rules is known as a style sheet. Skins are collections of application-specific styles, which define the appearance of an application. Skinning is the process of changing the appearance of an application (or the skin) on the fly. JavaFX does not provide a specific mechanism for skinning. Themes are visual characteristics of an operating system that are reflected in the appearance of UI elements of all applications. JavaFX has no direct support for themes.

You can add multiple style sheets to a JavaFX application. Style sheets are added to a scene or parents. Scene and Parent classes maintain an observable list of string URLs linking to style sheets.

JavaFX 8 use a default style sheet called Modena. Prior to JavaFX 8, the default style sheet was called Caspian. You can still use the Caspian style sheet as the default in JavaFX 8 using the static method `setUserAgentStylesheet(String url)` of the Application class. You can refer to the Caspian and Modena stylesheets' URLs using the constants named `STYLESHEET_CASPION` and `STYLESHEET_MODENA` defined in the Application class.

It is common for the visual properties of nodes to come from multiple sources. The JavaFX runtime uses the following priority rules to set the visual properties of a node: inline style (the highest priority), parent style sheets, scene style sheets, values set in the code using JavaFX API, and user agent style sheets (the lowest priority).

JavaFX offers two types of inheritance for CSS properties: CSS property types and CSS property values. In the first type of inheritance, all CSS properties declared in a JavaFX class are inherited by all its subclasses. In the second type of inheritance, a CSS property for a node may inherit its value from its parent. The parent of a node is the container of the node in the scene graph, not its JavaFX superclass.

Each style in a style sheet has a selector that identifies the nodes in the scene graph to which the style is applied. JavaFX CSS supports several types of selectors: class selectors and most of them work the same way they do in web browsers. You can look up a node in a scene graph by using a selector and the `lookup(String selector)` method of the Scene and Node classes.

The next chapter will discuss how to handle events in a JavaFX application.

## CHAPTER 9



### Event Handling

---

In this chapter, you will learn:

- What an event is
- What an event source, an event target, and event type are
- About the event processing mechanism
- How to handle events using event filters and event handlers
- How to handle mouse events, key events, and window events

#### What Is an Event?

In general, the term *event* is used to describe an occurrence of interest. In a GUI application, an event is an occurrence of a user interaction with the application. Clicking the mouse and pressing a key on the keyboard are examples of events in a JavaFX application.

An event in JavaFX is represented by an object of the `javafx.event.Event` class or any of its subclasses. Every event in JavaFX has three properties:

- An event source
- An event target
- An event type

When an event occurs in an application, you typically perform some processing by executing a piece of code. The piece of code that is executed in response to an event is known as an *event handler* or an *event filter*. I will clarify the difference between these shortly. For now, think of both as a piece of code and I will refer to both of them as event handlers. When you want to handle an event for a UI element, you need to add event handlers to the UI element, for example, a `Window`, a `Scene`, or a `Node`. When the UI element detects the event, it executes your event handlers.

The UI element that calls event handlers is the source of the event for those event handlers. When an event occurs, it passes through a chain of event dispatchers. The source of an event is the current element in the event dispatcher chain. The event source changes as the event passes through one dispatcher to another in the event dispatcher chain.

The event target is the destination of an event. The event target determines the route through which the event travels during its processing. Suppose a mouse click occurs over a `Circle` node. In this case, the `Circle` node is the event target of the mouse-clicked event.

The event type describes the type of the event that occurs. Event types are defined in a hierarchical fashion. Each event type has a name and a supertype.

The three properties that are common to all events in JavaFX are represented by objects of three different classes. Specific events define additional event properties;

for example, the event class to represent a mouse event adds properties to describe the location of the mouse cursor, state of the mouse buttons, among others. Table 9-1 lists the classes and interfaces involved in event processing. JavaFX has an event delivery mechanism that defines the details of the occurrence and processing of events. I will discuss all of these in detail in subsequent sections.

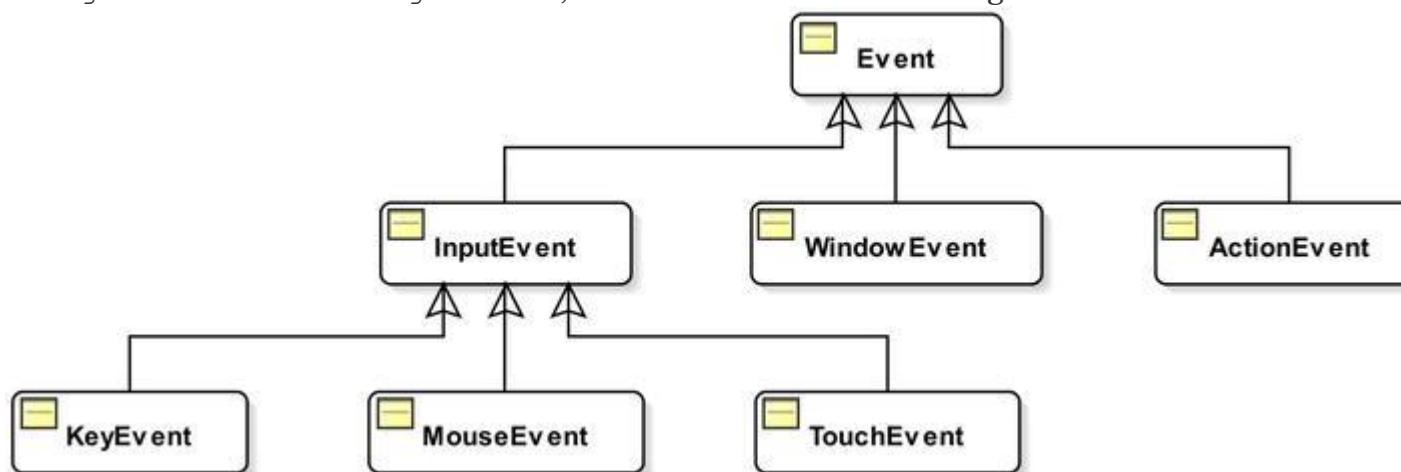
**Table 9-1.** Classes Involved in Event Processing

Name	Class/Interface	Description
Event	Class	An instance of this class represents an event. Several subclasses of the Event class exist to represent specific types of events.
EventTarget	Interface	An instance of this interface represents an event target.
EventType	Class	An instance of this class represents an event type, for example, mouse released, mouse moved.
EventHandler	Interface	An instance of this interface represents an event handler or an event listener. Its handle() method is called when the event for which it has been registered occurs.

## Event Class Hierarchy

Classes representing events in JavaFX are arranged in hierarchical fashion through class inheritance. Figure 9-1 shows a partial class diagram for the Event class.

The Event class is at the top of the class hierarchy and it inherits from java.util.EventObject class, which is not shown in the diagram.



**Figure 9-1.** A partial class hierarchy for the `javafx.event.Event` class

Subclasses of the Event class represent specific types of events. Sometimes a subclass of the Event class is used to represent a generic event of some kind. For

example, the `InputEvent` class represents a generic event to indicate a user input event, whereas the `KeyEvent` and `MouseEvent` classes represent specific input events such as the user input from the keyboard and mouse, respectively. An object of the `WindowEvent` class represents an event of a window, for example, showing and hiding of the window. An object of the `ActionEvent` is used to represent several kinds of events denoting some type of action, for example, firing a button or a menu item. Firing of a button may happen if the user clicks it with the mouse, presses some keys, or touches it on the touch screen.

The `Event` class provides properties and methods that are common to all events. The `getSource()` method returns an `Object`, which is the source of the event. The `Event` class inherits this method from the `EventObject` class. The `getTarget()` method returns an instance of the `EventTarget` interface, which is the target of the event. The `getEventType()` method returns an object of the `EventType` class, which indicates the type of the event.

The `Event` class contains `consume()` and `isConsumed()` methods. As noted before, an event travels from one element to another in an event-dispatching chain. Calling the `consume()` method on an `Event` object indicates that the event has been consumed and no further processing is required. After the `consume()` method is called, the event does not travel to the next element in the event-processing chain. The `isConsumed()` method returns `true` if the `consume()` method has been called, otherwise, it returns `false`.

Specific `Event` subclasses define more properties and methods. For example, the `MouseEvent` class defines `getX()` and `getY()` methods that return the x and y coordinates of the mouse cursor relative to the source of the event. I'll explain the details of the methods in event-specific classes when I discuss them later in this chapter or subsequent chapters.

## Event Targets

An *event target* is a UI element (not necessarily just `Nodes`) that can respond to events. Technically, a UI element that wants to respond to events must implement the `EventTarget` interface. That is, in JavaFX, implementing the `EventTarget` interface makes a UI element eligible to be an event target.

The `Window`, `Scene`, and `Node` classes implement the `EventTarget` interface. This means that all nodes, including windows and scenes, can respond to events. The classes for some UI elements, for example, `Tab`, `TreeItem`, and `MenuItem`, do not inherit from the `Node` class. They can still respond to events because they implement the `EventTarget` interface. If you develop a custom UI element, you will need to implement this interface if you want your UI element to respond to events.

The responsibility of an event target is to build a chain of event dispatchers, which is also called the *event route*. An *event dispatcher* is an instance of the `EventDispatcher` interface. Each dispatcher in the chain can affect the event by handling and consuming. An event dispatcher in the chain can also modify the event properties, substitute the event with a new event, or chain the event route. Typically, an event target route consists of dispatchers associated with all UI elements in the container-child hierarchy. Suppose you have a `Circle` node placed in an `HBox`, which is placed in a `Scene`. The `Scene` is added to a `Stage`. If the mouse is clicked on the `Circle`, the `Circle` becomes the event target.

The `Circle` builds an event dispatcher chain whose route will be, from head to tail, the `Stage`, `Scene`, `HBox`, and `Circle`.

## Event Types

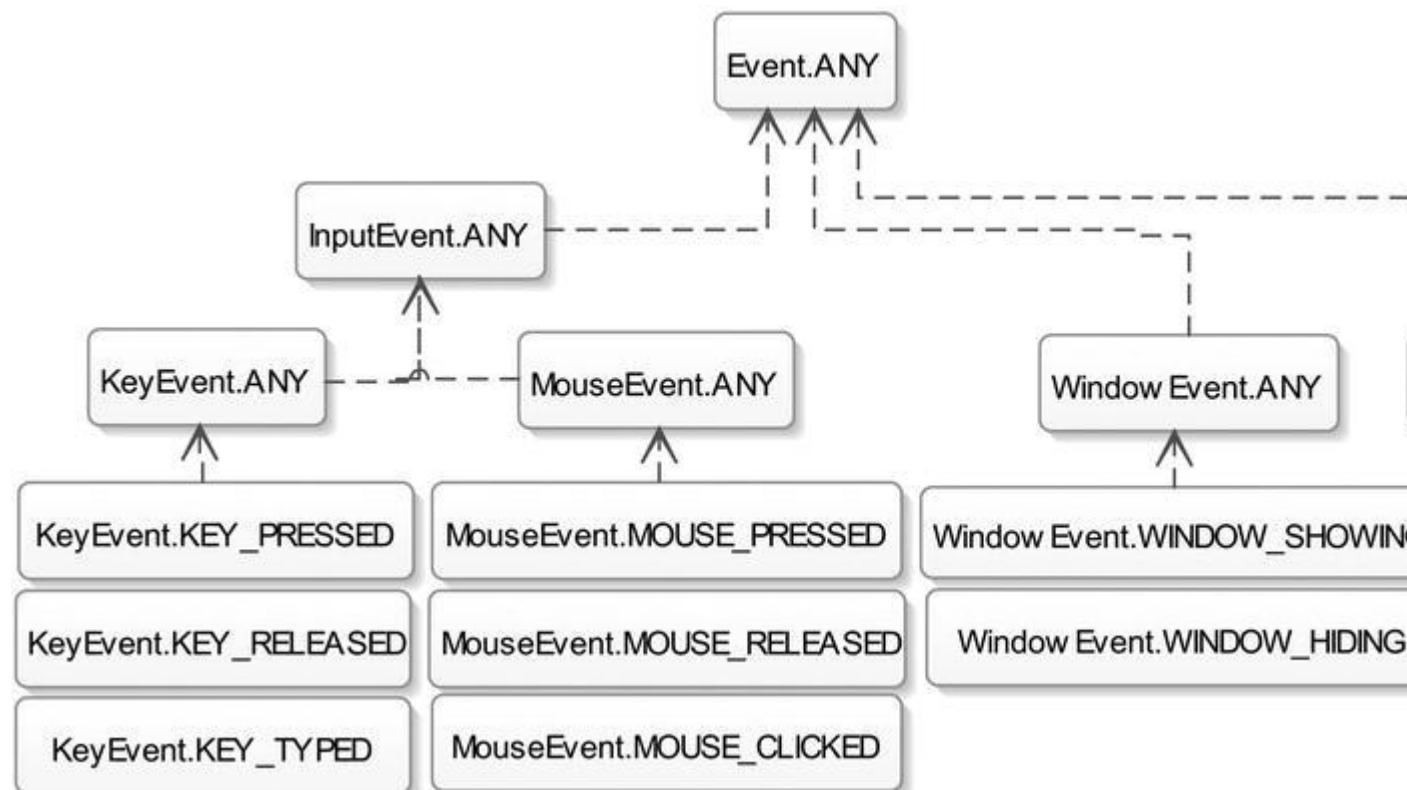
An instance of the `EventType` class defines an event type. Why do you need a separate class to define event types? Aren't separate event classes, for example, `KeyEvent`, `MouseEvent`, for each event sufficient to define event types? Can't you distinguish one event from another based on the event class?

The `EventType` class is used to further classify the events within an event class. For example, the `MouseEvent` class only tells us that the user has used the mouse. It does not tell us the details of the mouse use, for example, whether the mouse was pressed, released, dragged, or clicked. The `EventType` class is used to classify these subevent types of an event. The `EventType` class is a generic class whose type parameter is defined as follows:

```
EventType<T extends Event>
```

Event types are hierarchical. They are hierarchical by implementation, not by class inheritance. Each event type has a name and a supertype.

The `getName()` and `getSuperType()` methods in the `EventType` class return the name and supertype of an event type. The constant `Event.ANY`, which is the same as the constant `EventType.ROOT`, is the supertype of all events in JavaFX. Figure 9-2 shows a partial list of some event types that have been predefined in some event classes.



**Figure 9-2.** A partial list of predefined event types for some event classes

Note that the arrows in the diagram do not denote class inheritance. They denote dependencies. For example, the `InputEvent.ANY` event type depends on the `Event.ANY` event type, as the latter is the supertype of the former.

An event class, which has subevent types, defines an ANY event type. For example, the `MouseEvent` class defines an ANY event type that represents a mouse event of any type, for example, mouse released, mouse clicked, mouse moved. `MOUSE_PRESSED` and `MOUSE_RELEASED` are other event types defined in the `MouseEvent` class. The ANY event type in an event class is the supertype of all other event types in the same event class. For example, the `MouseEvent.ANY` event type is the supertype of `MOUSE_RELEASED` and `MOUSE_PRESSED` mouse events.

## Event Processing Mechanism

When an event occurs, several steps are performed as part of the event processing:

- Event target selection
- Event route construction
- Event route traversal

### Event Target Selection

The first step in the event processing is the selection of the event target. Recall that an event target is the destination node of an event. The event target is selected based on the event type.

For mouse events, the event target is the node at the mouse cursor. Multiple nodes can be available at the mouse cursor. For example, you can have a circle placed over a rectangle. The topmost node at the mouse cursor is selected as the event target.

The event target for key events is the node that has focus. How a node gets the focus depends on the type of the node. For example, a `TextField` may gain focus by clicking the mouse inside it or using the focus traversal keys such as Tab or Shift + Tab on the Windows format. Shapes such as `Circles` or `Rectangles` do not get focus by default. If you want them to receive key events, you can give them focus by calling the `requestFocus()` method of the `Node` class.

JavaFX supports touch and gesture events on touch-enabled devices. A touch event is generated by touching a touch screen. Each touch action has a point of contact called a *touch point*. It is possible to touch a touch screen with multiple fingers, resulting in multiple touch points. Each state of a touch point, for example, pressed, released, and so forth, generates a touch event. The location of the touch point determines the target of the touch event. For example, if the location of the touch event is a point within a circle, the circle becomes the target of the touch event. In case of multiple nodes at the touch point, the topmost node is selected as the target.

Users can interact with a JavaFX application using gestures. Typically, a gesture on a touch screen and a track pad consists of multiple touch points with touch actions. Examples of gesture events are rotating, scrolling, swiping, and zooming. A rotating gesture is performed by rotating two fingers around each other. A scrolling gesture is performed by dragging a finger on touch screen. A swiping gesture is performed by dragging a finger (or multiple fingers) on the touch screen in one

direction. A zooming gesture is performed to scale a node by dragging two fingers apart or closer.

The target for gesture events are selected depending on the type of gesture. For direct gestures, for example, gestures performed on touch screens, the topmost node at the center point of all touch points at the start of the gesture is selected as the event target. For indirect gestures, for example, gestures performed on a track pad, the topmost node at the mouse cursor is selected as the event target.

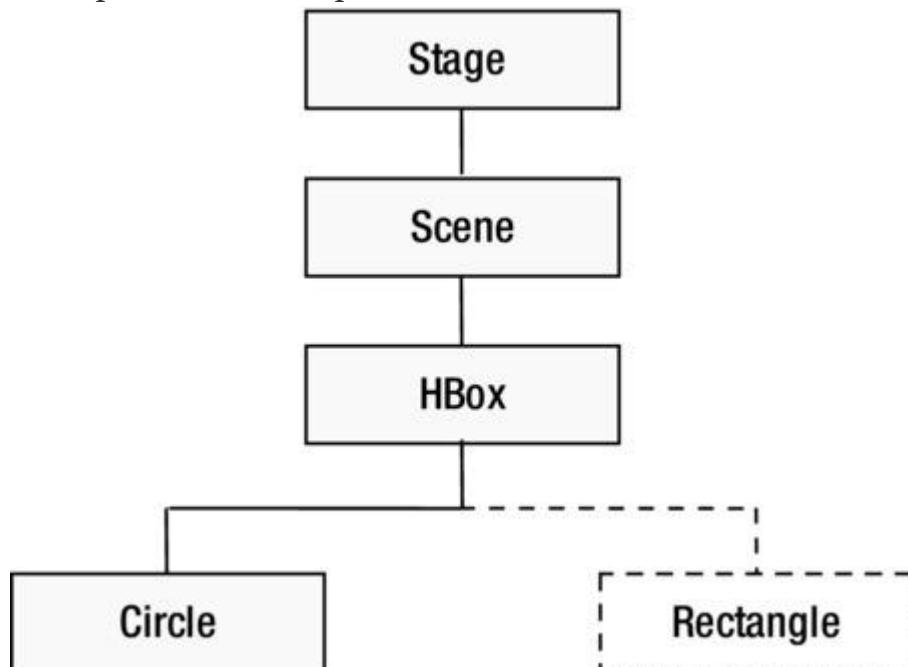
### Event Route Construction

An event travels through event dispatchers in an event dispatch chain. The event dispatch chain is the *event route*. The initial and default routes for an event are determined by the event target. The default event route consists of the container-children path starting at the stage to the event target node.

Suppose you have placed a `Circle` and a `Rectangle` in an `HBox` and the `HBox` is the root node of the `Scene` of a `Stage`. When you click the `Circle`, the `Circle` becomes the event target. The `Circle` constructs the default event route, which is the path starting at the stage to the event target (the `Circle`).

In fact, an event route consists of event dispatchers that are associated with nodes. However, for all practical and understanding purposes, you can think of the event route as the path comprising the nodes. Typically, you do not deal with event dispatchers directly.

Figure 9-3 shows the event route for the mouse-clicked event. The nodes on the event route have been shown in gray background fills. The nodes on the event route are connected by solid lines. Note that the `Rectangle` that is part of the scene graph is not part of the event path when the `Circle` is clicked.



**Figure 9-3.** Construction of the default event route for an event

An event dispatch chain (or event route) has a *head* and a *tail*. In Figure 9-3, the `Stage` and the `Circle` are the head and the tail of the event dispatch chain, respectively. The initial event route may be modified as the event processing

progresses. Typically, but not necessarily, the event passes through all nodes in its route twice during the event traversal step, as described in the next section.

### Event Route Traversal

An event route traversal consists of two phases:

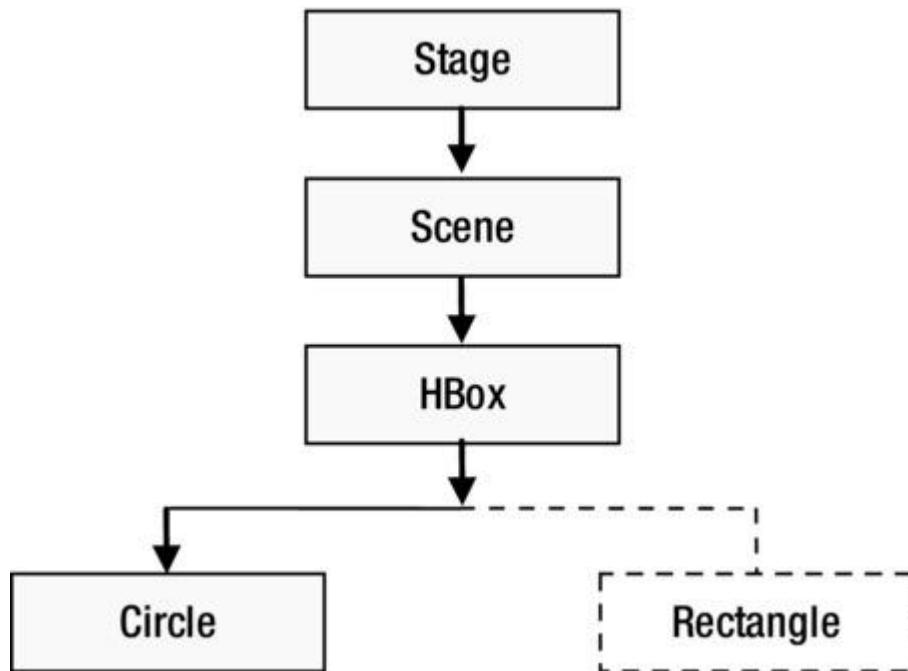
- Capture phase
- Bubbling phase

An event travels through each node in its route twice: once during the capture phase and once during the bubbling phase. You can register event filters and event handlers to a node for specific events types. The event filters and event handlers registered to a node are executed as the event passes through the node during the capture phase and the bubbling phase, respectively. The event filters and handlers are passed in the reference of the current node as the source of the event. As the event travels from one node to another, the event source keeps changing. However, the event target remains the same from the start to the finish of the event route traversal.

During the route traversal, a node can consume the event in event filters or handlers, thus completing the processing of the event. Consuming an event is simply calling the `consume()` method on the event object. When an event is consumed, the event processing is stopped, even though some of the nodes in the route were not traversed at all.

### Event Capture Phase

During the capture phase, an event travels from the head to the tail of its event dispatch chain. Figure 9-4 shows the traveling of a mouse-clicked event for the `Circle` in our example in the capture phase. The down arrows in the figure denote the direction of the event travel. As the event passes through a node, the registered event filters for the node are executed. Note that the event capture phase executes only event filters, not event handlers, for the current node.



*Figure 9-4. The event capture phase*

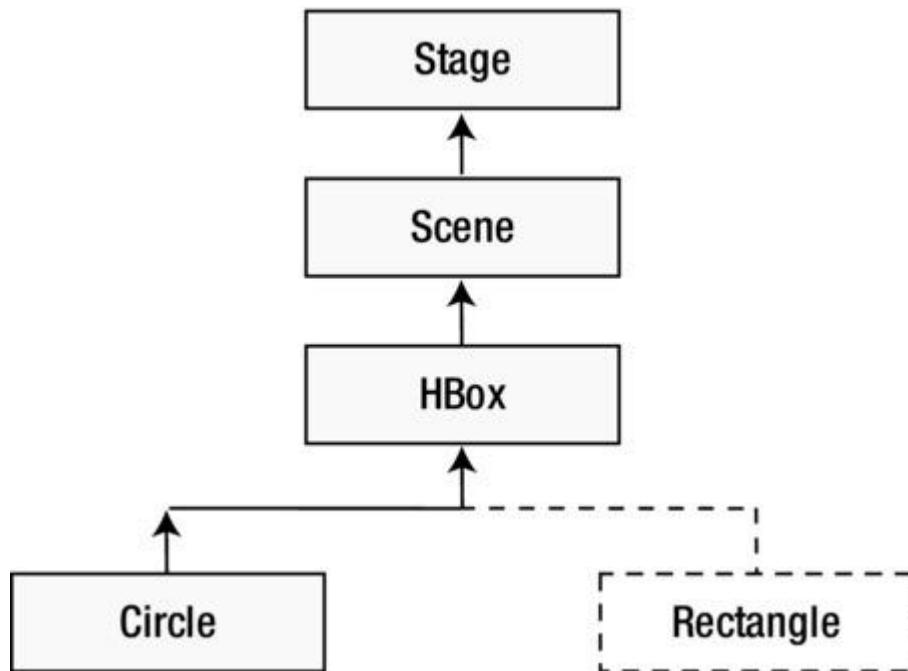
In Figure 9-4, the event filters for the `Stage`, `Scene`, `HBox`, and `Circle` are executed in order, assuming none of the event filters consumes the event.

You can register multiple event filters for a node. If the node consumes the event in one of its event filters, its other event filters, which have not been executed yet, are executed before the event processing stops. Suppose you have registered five event filters for the `Scene` in our example, and the first event filter that is executed consumes the event. In this case, the other four event filters for the `Scene` will still be executed. After executing the fifth event filter for the `Scene`, the event processing will stop, without the event traveling to the remaining nodes (`HBox` and `Circle`).

In the event capture phase, you can intercept events (and provide a generic response) that are targeted at the children of a node. For example, you can add event filters for the mouse-clicked event to the `Stage` in our example to intercept all mouse-clicked events for all its children. You can block events from reaching their targets by consuming the event in event filters for a parent node. For example, if you consume the mouse-clicked event in a filter for the `Stage`, the event will not reach its target, in our example, the `Circle`.

### Event Bubbling Phase

During the bubbling phase, an event travels from the tail to the head of its event dispatch chain. Figure 9-5 shows the traveling of a mouse-clicked event for the `Circle` in the bubbling phase.



*Figure 9-5. The event bubbling phase*

The up arrows in Figure 9-5 denote the direction of the event travel. As the event passes through a node, the registered event handlers for the node are executed. Note that the event bubbling phase executes event handlers for the current node, whereas the event capture phase executes the event filters.

In our example, the event handlers for the `Circle`, `HBox`, `Scene`, and `Stage` are executed in order, assuming none of the event filters consumes the event. Note that the event bubbling phase starts at the target of the event and travels up to the topmost parent in the parent-children hierarchy.

You can register multiple event handlers for a node. If the node consumes the event in one of its event handlers, its other event handlers, which have not been executed yet, are executed before the event processing stops. Suppose you have registered five event handlers for the `Circle` in our example, and the first event handler that is executed consumes the event. In this case, the other four event handlers for the `Circle` will still be executed. After executing the fifth event handler for the `Circle`, the event processing will stop, without the event traveling to the remaining nodes (`HBox`, `Scene`, and `Stage`).

Typically, event handlers are registered to target nodes to provide a specific response to events. Sometimes event handlers are installed on parent nodes to provide a default event response for all its children. If an event target decides to provide a specific response to the event, it can do so by adding event handlers and consuming the event, thus blocking the event from reaching the parent nodes in the event bubbling phase.

Let's look at a trivial example. Suppose you want to display a message box to the user when he clicks anywhere inside a window. You can register an event handler to the window to display the message box. When the user clicks inside a circle in the window, you want to display a specific message. You can register an event handler to the circle to provide the specific message and consume the event. This will provide a specific event response when the circle is clicked, whereas for other nodes, the window provides a default event response.

## Handling Events

Handling an event means executing the application logic in response to the occurrence of the event. The application logic is contained in the event filters and handlers, which are objects of the `EventHandler` interface, as shown in the following code:

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

The `EventHandler` class is a generic class in the `javafx.event` package. It extends the `EventListener` marker interface, which is in the `java.util` package. The `handle()` method receives the reference of the event object, for example, the reference of the `KeyEvent`, `MouseEvent`, among others.

Both event filters and handlers are objects of the same `EventHandler` interface. You cannot tell whether an `EventHandler` object is an event filter or an event handler by just looking at it. In fact, you can register the same `EventHandler` object as event filters as well as handlers at the same time. The distinction between the two is made when they are registered to a node. Nodes provide different methods to register them. Internally, nodes know whether an `EventHandler` object was registered as an event filter or a handler. Another distinction between them is made based on the event traversal phase in which they are called. During the event capture phase, the `handle()` method of registered filters is called, whereas the `handle()` method of registered handlers is called in the event bubbling phase.

**Tip** In essence, handling an event means writing the application logic for `EventHandler` objects and registering them to nodes as event filters, handlers, or both.

### Creating Event Filters and Handlers

Creating event filters and handlers is as simple as creating objects of the class that implement the `EventHandler` interface. Before Java 8, you would use inner classes to create event filters and handlers, as in the following code:

```
EventHandler<MouseEvent> aHandler = new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent e) {
        /* Event handling code goes here */
    }
};
```

Starting in Java 8, using lambda expressions is the best choice for creating the event filters and handlers, as in the following code:

```
EventHandler<MouseEvent> aHandler = e -> /* Event handling code goes here */;
```

I use lambda expressions in this book to create event filters and handlers. If you are not familiar with lambda expressions in Java 8, I suggest you learn at least the basics so you can understand the event handling code.

The following snippet of code creates a `MouseEvent` handler. It prints the type of the mouse event that occurs:

```
EventHandler<MouseEvent> mouseEventHandler =
    e -> System.out.println("Mouse event type: " + e.getEventType());
```

### Registering Event Filters and Handlers

If you want a node to process events of specific types, you need to register event filters and handlers for those event types to the node. When the event occurs,

the `handle()` method of the registered event filters and handlers for the node are called following the rules discussed in the previous sections. If the node is no longer interested in processing the events, you need to unregister the event filters and handlers from the node. Registering and unregistering event filters and handlers are also known as adding and removing event filters and handlers, respectively.

JavaFX provides two ways to register and unregister event filters and handlers to nodes:

- Using  
the `addEventFilter()`, `addEventHandler()`, `removeEventFilter()`, and `removeEventHandler()` methods
- Using the `onXXX` convenience properties

## Using `addXXX()` and `removeXXX()` Methods

You can use the `addEventFilter()` and `addEventHandler()` methods to register event filters and handlers to nodes, respectively. These methods are defined in the `Node` class, `Scene` class, and `Window` class. Some classes (e.g., `MenuItem` and `TreeItem`) can be event targets; however, they are not inherited from the `Node` class. The classes provide only the `addEventHandler()` method for event handlers registration, such as:

- `<T extends Event> void addEventFilter(EventType<T> eventType, EventHandler<? super T> eventFilter)`
- `<T extends Event> void addEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)`

These methods have two parameters. The first parameter is the event type and the second is an object of the `EventHandler` interface.

You can handle mouse-clicked events for a `Circle` using the following snippet of code:

```
import javafx.scene.shape.Circle;
import javafx.event.EventHandler;
import javafx.scene.input.MouseEvent;
...
Circle circle = new Circle (100, 100, 50);

// Create a MouseEvent filter
EventHandler<MouseEvent> mouseEventFilter =
    e -> System.out.println("Mouse event filter has been
called.");

// Create a MouseEvent handler
EventHandler<MouseEvent> mouseEventHandler =
    e -> System.out.println("Mouse event handler has been called.");

// Register the MouseEvent filter and handler to the Circle
// for mouse-clicked events
circle.addEventFilter(MouseEvent.MOUSE_CLICKED, mouseEventFilter);
circle.addEventHandler(MouseEvent.MOUSE_CLICKED, mouseEventHandler);
```

This code creates two `EventHandler` objects, which prints a message on the console. At this stage, they are not event filters or handlers. They are just two `EventHandler` objects. Note that giving the reference variables names and printing messages that use the words filter and handler does not make any difference

in their status as filters and handlers. The last two statements register one of the `EventHandler` objects as an event filter and another as an event handler; both are registered for the mouse-clicked event.

Registering the same `EventHandler` object as event filters as well as handlers is allowed. The following snippet of code uses one `EventHandler` object as the filter and handler for the `Circle` to handle the mouse-clicked event:

```
// Create a MouseEvent EventHandler object
EventHandler<MouseEvent> handler =
    e -> System.out.println("Mouse event filter or handler has been
called.");

// Register the same EventHandler object as the MouseEvent filter and handler
// to the Circle for mouse-clicked events
circle.addEventFilter(MouseEvent.MOUSE_CLICKED, handler);
circle.addEventHandler(MouseEvent.MOUSE_CLICKED, handler);
```

**Tip** You can add multiple event filters and events for a node using the `addEventFilter()` and `addEventHandler()` methods. You need to call these methods once for every instance of the event filters and handlers that you want to add.

**Listing 9-1** has the complete program to demonstrate the handling of the mouse-clicked events of a `Circle` object. It uses an event filter and an event handler. Run the program and click inside the circle. When the circle is clicked, the event filter is called first, followed by the event handler. This is evident from the output. The mouse-clicked event occurs every time you click any point inside the circle. If you click outside the circle, the mouse-clicked event still occurs; however, you do not see any output because you have not registered event filters or handlers on the `HBox`, `Scene`, and `Stage`.

### **Listing 9-1.** Registering Event Filters and Handlers

```
// EventRegistration.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class EventRegistration extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle (100, 100, 50);
        circle.setFill(Color.CORAL);

        // Create a MouseEvent filter
        EventHandler<MouseEvent> mouseEventFilter =
            e -> System.out.println("Mouse event filter has been
called.");

        // Create a MouseEvent handler
        EventHandler<MouseEvent> mouseEventHandler =
```

```

        e -> System.out.println("Mouse event handler has been
called.");
        // Register the MouseEvent filter and handler to the Circle
        // for mouse-clicked events
        circle.addEventFilter(MouseEvent.MOUSE_CLICKED,
mouseEventFilter);
        circle.addEventHandler(MouseEvent.MOUSE_CLICKED,
mouseEventHandler);

        HBox root = new HBox();
        root.getChildren().add(circle);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Registering Event Filters and Handlers");
        stage.show();
        stage.sizeToScene();
    }
}
Mouse event filter has been called.
Mouse event handler has been called.
...

```

To unregister an event filter and an event handler, you need to call the `removeEventFilter()` and `removeEventHandler()` methods, respectively:

- <T extends Event> void `removeEventFilter(EventType<T> eventType,`  
`EventHandler<? super T> eventFilter)`
- <T extends Event> void `removeEventHandler(EventType<T> eventType,`  
`EventHandler<? super T> eventHandler)`

The following snippet of code adds and removes an event filter to a `Circle`, and later, it removes them. Note that once an `EventHandler` is removed from a node, its `handle()` method is not called when the event occurs:

```

// Create a MouseEvent EventHandler object
EventHandler<MouseEvent> handler =
    e -> System.out.println("Mouse event filter or handler has been
called.");

// Register the same EventHandler object as the MouseEvent filter and handler
// to the Circle
// for mouse-clicked events
circle.addEventFilter(MouseEvent.MOUSE_CLICKED, handler);
circle.addEventHandler(MouseEvent.MOUSE_CLICKED, handler);

...
// At a later stage, when you are no longer interested in handling the mouse
// clicked event for the Circle, unregister the event filter and handler
circle.removeEventFilter(MouseEvent.MOUSE_CLICKED, handler);
circle.removeEventHandler(MouseEvent.MOUSE_CLICKED, handler);

```

## Using `onXXX` Convenience Properties

The `Node`, `Scene`, and `Window` classes contain event properties to store event handlers of some selected event types. The property names use the event type pattern. They are named as `onXXX`. For example, the `onMouseClicked` property

stores the event handler for the mouse-clicked event type; the `onKeyTyped` property stores the event handler for the key-typed event, and so on. You can use the `setOnXXX()` methods of these properties to register event handlers for a node. For example, use the `setOnMouseClicked()` method to register an event handler for the mouse-clicked event and use the `setOnKeyTyped()` method to register an event handler for the key-typed event, and so on. The `setOnXXX()` methods in various classes are known as convenience methods for registering event handlers.

You need to remember some points about the `onXXX` convenience properties:

- They only support the registration of event handlers, not event filters. If you need to register event filters, use the `addEventFilter()` method.
- They only support the registration of *one event handler* for a node. Multiple event handlers for a node may be registered using the `addEventHandler()` method.
- These properties exist only for the commonly used events for a node type. For example, the `onMouseClicked` property exists in the `Node` and `Scene` classes, but not the `Window` class; the `onShowing` property exists in the `Window` class, but not in the `Node` and `Scene` classes.

The program in Listing 9-2 works the same as the program in Listing 9-1. This time, you have used the `onMouseClicked` property of the `Node` class to register the mouse-clicked event handler for the circle. Notice that to register the event filter, you have to use the `addEventFilter()` method as before. Run the program and click inside the circle. You will get the same output you got when running the code in Listing 9-1.

### *****Listing 9-2.***** Using the Convenience Event Handler Properties

```
// EventHandlerProperties.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class EventHandlerProperties extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle(100, 100, 50);
        circle.setFill(Color.CORAL);

        HBox root = new HBox();
        root.getChildren().add(circle);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using convenience event handler properties");
    }
}
```

```

        stage.show();
        stage.sizeToScene();

        // Create a MouseEvent filter
        EventHandler<MouseEvent> eventFilter =
            e -> System.out.println("Mouse event filter has been
called.");

        // Create a MouseEvent handler
        EventHandler<MouseEvent> eventHandler =
            e -> System.out.println("Mouse event handler has been
called.");

        // Register the filter using the addEventFilter() method
        circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventFilter);

        // Register the handler using the setter method for
        // the onMouseClicked convenience event property
        circle.setOnMouseClicked(eventHandler);
    }
}

```

The convenience event properties do not provide a separate method to unregister the event handler. Setting the property to `null` unregisters the event handler that has already been registered:

```

// Register an event handler for the mouse-clicked event
circle.setOnMouseClicked(eventHandler);

...

// Later, when you are no longer interested in processing the mouse-clicked
event, // unregister it.
circle.setOnMouseClicked(null);

```

Classes that define the `onXXX` event properties also define `getOnXXX()` getter methods that return the reference of the registered event handler. If no event handler is set, the getter method returns `null`.

## Execution Order of Event Filters and Handlers

There are some execution order rules for event filters and handlers for both similar and different nodes:

- Event filters are called before event handlers. Event filters are executed from the topmost parent to the event target in the parent-child order. Event handlers are executed in the reverse order of the event filters. That is, the execution of the event handlers starts at the event target and moves up in the child-parent order.
- For the same node, event filters and handlers for a specific event type are called before the event filters and handlers for generic types. Suppose you have registered event handlers to a node for `MouseEvent.ANY` and `MouseEvent.MOUSE_CLICKED`. Event handlers for both event types are capable of handling mouse-clicked events. When the mouse is clicked on the node, the event handler for the `MouseEvent.MOUSE_CLICKED` event type is called before the event handler for the `MouseEvent.ANY` event type. Note that a mouse-pressed event and a mouse-released event occur before a mouse-clicked event occurs. In our example, these events will be handled by the event handler for the `MouseEvent.ANY` event type.

- The order in which the event filters and handlers for the same event type for a node are executed is not specified. There is one exception to this rule. Event handlers registered to a node using the `addEventHandler()` method are executed before the event handlers registered using the `setOnXXX()` convenience methods.

**Listing 9-3** demonstrates the execution order of the event filters and handlers for different nodes. The program adds a `Circle` and a `Rectangle` to an `HBox`. The `HBox` is added to the `Scene`. An event filter and an event handler are added to the `Stage`, `Scene`, `HBox`, and `Circle` for the mouse-clicked event. Run the program and click anywhere inside the circle. The output shows the order in which filters and handlers are called. The output contains the event phase, type, target, source, and location. Notice that the source of the event changes as the event travels from one node to another. The location is relative to the event source. Because every node uses its own local coordinate system, the same point, where the mouse is clicked, has different values for (x, y) coordinates relative to different nodes.

### **Listing 9-3.** Execution Order for Event Filters and Handlers

```
// CaptureBubblingOrder.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import static javafx.scene.input.MouseEvent.MOUSE_CLICKED;
```

If you click the rectangle, you will notice that the output shows the same path for the event through its parents as it did for the circle. The event still passes through the rectangle, which is the event target. However, you do not see any output, because you have not registered any event filters or handlers for the rectangle to output any message. You can click at any point outside the circle and rectangle to see the event target and the event path.

```
public class CaptureBubblingOrder extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle(50, 50, 50);
        circle.setFill(Color.CORAL);

        Rectangle rect = new Rectangle(100, 100);
        rect.setFill(Color.TAN);

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(circle, rect);

        Scene scene = new Scene(root);
```

```

        // Create two EventHandlers
        EventHandler<MouseEvent> filter = e -> handleEvent("Capture", e);
        EventHandler<MouseEvent> handler = e -> handleEvent("Bubbling",
e);

        // Register filters
        stage.addEventFilter(MOUSE_CLICKED, filter);
        scene.addEventFilter(MOUSE_CLICKED, filter);
        root.addEventFilter(MOUSE_CLICKED, filter);
        circle.addEventFilter(MOUSE_CLICKED, filter);

        // Register handlers
        stage.addHandler(MOUSE_CLICKED, handler);
        scene.addHandler(MOUSE_CLICKED, handler);
        root.addHandler(MOUSE_CLICKED, handler);
        circle.addHandler(MOUSE_CLICKED, handler);

        stage.setScene(scene);
        stage.setTitle("Event Capture and Bubbling Execution Order");
        stage.show();
    }

    public void handleEvent(String phase, MouseEvent e) {
        String type = e.getEventType().getName();
        String source = e.getSource().getClass().getSimpleName();
        String target = e.getTarget().getClass().getSimpleName();

        // Get coordinates of the mouse cursor relative to the event
        source
        double x = e.getX();
        double y = e.getY();

        System.out.println(phase + ": Type=" + type +
                           ", Target=" + target + ", Source=" + source +
                           ", location(" + x + ", " + y + ")");
    }
}
}

```

**Listing 9-4** demonstrates the execution order of event handlers for a node. It displays a circle. It registers three event handlers for the circle:

- One for the `MouseEvent.ANY` event type
- One for the `MouseEvent.MOUSE_CLICKED` event type using the `addHandler()` method
- One for the `MouseEvent.MOUSE_CLICKED` event type using the `setOnMouseClicked()` method

Run the program and click inside the circle. The output shows the order in which three event handlers are called. The order will be similar to that presented in the discussion at the beginning of the section.

#### ***Listing 9-4.*** Order of Execution of Event Handlers for a Node

```

// HandlersOrder.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;

```

```

import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class HandlersOrder extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle(50, 50, 50);
        circle.setFill(Color.CORAL);

        HBox root = new HBox();
        root.getChildren().addAll(circle);
        Scene scene = new Scene(root);

        /* Register three handlers for the circle that can handle mouse-
        clicked events */
        // This will be called last
        circle.addEventHandler(MouseEvent.ANY, e ->
handleAnyMouseEvent(e));

        // This will be called first
        circle.addEventHandler(MouseEvent.MOUSE_CLICKED,
            e -> handleMouseClicked("addEventHandler()", e));

        // This will be called second
        circle.setOnMouseClicked(e ->
handleMouseClicked("setOnMouseClicked()", e));
    }

    public void handleMouseClicked(String registrationMethod, MouseEvent e)
{
    System.out.println(registrationMethod
        + ": MOUSE_CLICKED handler detected a mouse
click.");
}

    public void handleAnyMouseEvent(MouseEvent e) {
        // Print a message only for mouse-clicked events, ignoring
        // other mouse events such as mouse-pressed, mouse-released, etc.
        if (e.getEventType() == MouseEvent.MOUSE_CLICKED) {
            System.out.println("MouseEvent.ANY handler detected
a mouse click.");
        }
    }
}
addEventHandler(): MOUSE_CLICKED handler detected a mouse click.
setOnMouseClicked(): MOUSE_CLICKED handler detected a mouse click.
MouseEvent.ANY handler detected a mouse click.

```

## Consuming Events

An event is consumed by calling its `consume()` method. The event class contains the method and it is inherited by all event classes. Typically, the `consume()` method is called inside the `handle()` method of the event filters and handlers.

Consuming an event indicates to the event dispatcher that the event processing is complete and that the event should not travel any farther in the event dispatch chain.

If an event is consumed in an event filter of a node, the event does not travel to any child node. If an event is consumed in an event handler of a node, the event does not travel to any parent node.

All event filters or handlers for the consuming node are called, irrespective of which filter or handler consumes the event. Suppose you have registered three event handlers for a node and the event handler, which is called first, consumes the event. In this case, the other two event handlers for the node are still called.

If a parent node does not want its child nodes to respond to an event, it can consume the event in its event filter. If a parent node provides a default response to an event in an event handler, a child node can provide a specific response and consume the event, thus suppressing the default response of the parent.

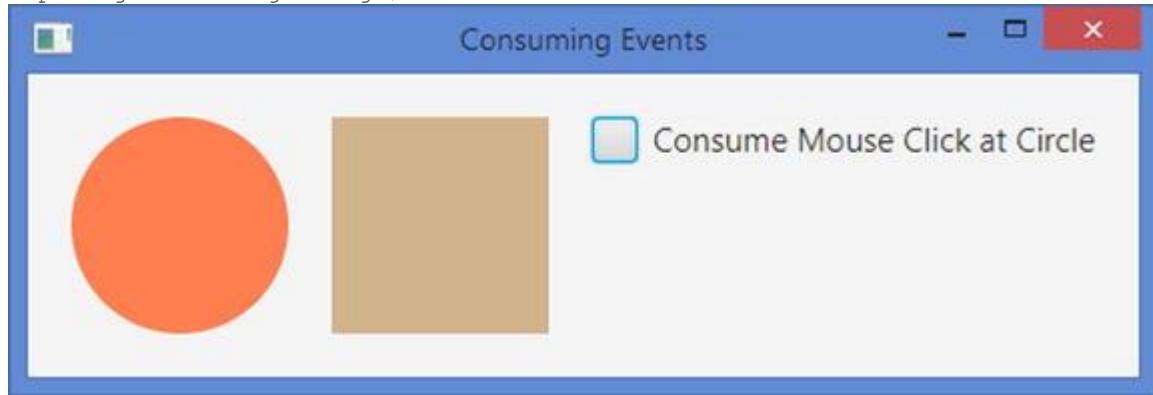
Typically, nodes consume most input events after providing a default response. The rule is that all event filters and handlers of a node are called, even if one of them consumes the event. This makes it possible for developers to execute their event filters and handlers for a node even if the node consumes the event.

The code in Listing 9-5 shows how to consume an event. Figure 9-6 shows the screen when you run the program.

### ***Listing 9-5.*** Consuming Events

```
// ConsumingEvents.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.input.MouseEvent;
import static javafx.scene.input.MouseEvent.MOUSE_CLICKED;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
```



**Figure 9-6.** Consuming events

The program adds a Circle, a Rectangle, and a CheckBox to an HBox. The HBox is added to the scene as the root node. An event handler is added to the Stage, Scene, HBox, and Circle. Notice that you have a different event handler for the Circle, just to keep the program logic simple. When the check box

is selected, the event handler for the circle consumes the mouse-clicked event, thus preventing the event from traveling up to the HBox, Scene, and Stage. If the check box is not selected, the mouse-clicked event on the circle travels from the Circle to the HBox, Scene, and Stage. Run the program and, using the mouse, click the different areas of the scene to see the effect. Notice that the mouse-clicked event handler for the HBox, Scene, and Stage are executed, even if you click a point outside the circle, because they are in the event dispatch chain of the clicked nodes.

```
public class ConsumingEvents extends Application {
    private CheckBox consumeEventCbx = new CheckBox("Consume Mouse Click at Circle");

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

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle(50, 50, 50);
        circle.setFill(Color.CORAL);

        Rectangle rect = new Rectangle(100, 100);
        rect.setFill(Color.TAN);

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(circle, rect, consumeEventCbx);

        Scene scene = new Scene(root);

        // Register mouse-clicked event handlers to all nodes,
        // except the rectangle and checkbox
        EventHandler<MouseEvent> handler = e -> handleEvent(e);
        EventHandler<MouseEvent> circleMeHandler = e ->
        handleEventforCircle(e);

        stage.addEventHandler(MOUSE_CLICKED, handler);
        scene.addEventHandler(MOUSE_CLICKED, handler);
        root.addEventHandler(MOUSE_CLICKED, handler);
        circle.addEventHandler(MOUSE_CLICKED, circleMeHandler);

        stage.setScene(scene);
        stage.setTitle("Consuming Events");
        stage.show();
    }

    public void handleEvent(MouseEvent e) {
        print(e);
    }

    public void handleEventforCircle(MouseEvent e) {
        print(e);
        if (consumeEventCbx.isSelected()) {
            e.consume();
        }
    }

    public void print(MouseEvent e) {
        String type = e.getEventType().getName();
        String source = e.getSource().getClass().getSimpleName();
        String target = e.getTarget().getClass().getSimpleName();

        // Get coordinates of the mouse cursor relative to the event
        source
    }
}
```

```

        double x = e.getX();
        double y = e.getY();

        System.out.println("Type=" + type + ", Target=" + target +
                           ", Source=" + source +
                           ", location(" + x + ", " + y + ")");
    }
}

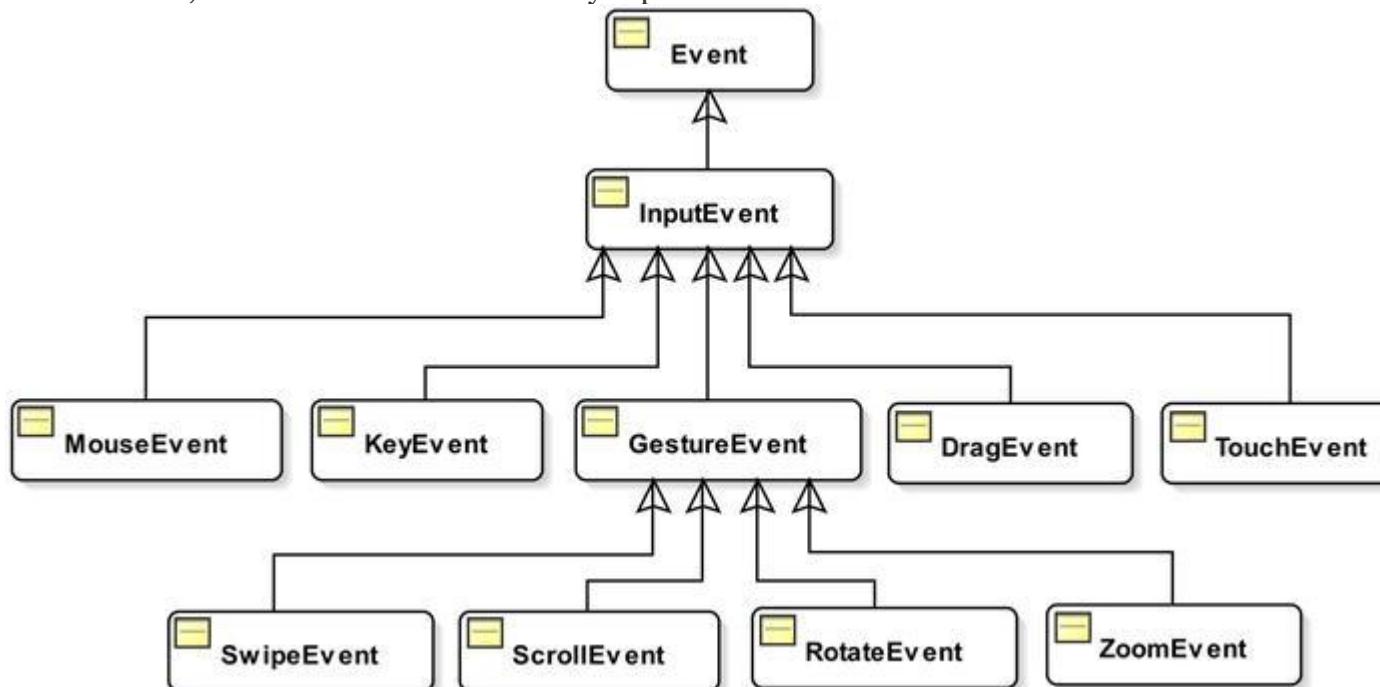
```

Clicking the check box does not execute the mouse-clicked event handlers for the `HBox`, `Scene`, and `Stage`, whereas clicking the rectangle does. Can you think of a reason for this behavior? The reason is simple. The check box has a default event handler that takes a default action and consumes the event, preventing it from traveling up the event dispatch chain. The rectangle does not consume the event, allowing it to travel up the event dispatch chain.

**Tip** Consuming an event by the event target in an event filter has no effect on the execution of any other event filters. However, it prevents the event bubbling phase from happening. Consuming an event in the event handlers of the topmost node, which is the head of the event dispatch chain, has no effect on the event processing at all.

## Handling Input Events

An input event indicates a user input (or a user action), for example, clicking the mouse, pressing a key, touching a touch screen, and so forth. JavaFX supports many types of input events. Figure 9-7 shows the class diagram for some of the classes that represent input event. All input event-related classes are in the `javafx.scene.input` package. The `InputEvent` class is the superclass of all input event classes. Typically, nodes execute the user-registered input event handlers before taking the default action. If the user event handlers consume the event, nodes do not take the default action. Suppose you register key-typed event handlers for a `TextField`, which consume the event. When you type a character, the `TextField` will not add and display it as its content. Therefore, consuming input events for nodes gives you a chance to disable the default behavior of the node. In next sections, I will discuss mouse and key input events.



**Figure 9-7.** Class hierarchy for some input events

## Handling Mouse Events

An object of the `MouseEvent` class represents a mouse event.

The `MouseEvent` class defines the following mouse-related event types constants.

All constants are of the type `EventType<MouseEvent>`. The `Node` class contains the convenience `onXXX` properties for most of the mouse event types that can be used to add one event handler of a specific mouse event type for a node:

- **ANY:** It is the supertype of all mouse event types. If a node wants to receive all types of mouse events, you would register handlers for this type. The `InputEvent.ANY` is the supertype of this event type.
- **MOUSE\_PRESSED:** Pressing a mouse button generates this event. The `getButton()` method of the `MouseEvent` class returns the mouse button that is responsible for the event. A mouse button is represented by the `NONE`, `PRIMARY`, `MIDDLE`, and `SECONDARY` constants defined in the `MouseButton` enum.
- **MOUSE\_RELEASED:** Releasing a mouse button generates this event. This event is delivered to the same node on which the mouse was pressed. For example, you can press a mouse button on a circle, drag the mouse outside the circle, and release the mouse button. The `MOUSE_RELEASED` event will be delivered to the circle, not the node on which the mouse button was released.
- **MOUSE\_CLICKED:** This event is generated when a mouse button is clicked on a node. The button should be pressed and released on the same node for this event to occur.
- **MOUSE\_MOVED:** Moving the mouse without pressing any mouse buttons generates this event.
- **MOUSE\_ENTERED:** This event is generated when the mouse enters a node. The event capture and bubbling phases do not take place for this event. That is, event filters and handlers of the parent nodes of the event target of this event are not called.
- **MOUSE\_ENTERED\_TARGET:** This event is generated when the mouse enters a node. It is a variant of the `MOUSE_ENTERED` event type. Unlike the `MOUSE_ENTER` event, the event capture and bubbling phases take place for this event.
- **MOUSE\_EXITED:** This event is generated when the mouse leaves a node. The event capture and bubbling phases do not take place for this event, that is, it is delivered only to the target node.
- **MOUSE\_EXITED\_TARGET:** This event is generated when the mouse leaves a node. It is a variant of the `MOUSE_EXITED` event type. Unlike the `MOUSE_EXIT` event, the event capture and bubbling phases take place for this event.
- **DRAG\_DETECTED:** This event is generated when the mouse is pressed and dragged over a node over a platform-specific distance threshold.
- **MOUSE\_DRAGGED:** Moving the mouse with a pressed mouse button generates this event. This event is delivered to the same node on which

the mouse button was pressed, irrespective of the location of the mouse pointer during the drag.

## Getting Mouse Location

The `MouseEvent` class contains methods to give you the location of the mouse when a mouse event occurs. You can obtain the mouse location relative to the coordinate systems of the event source node, the scene, and the screen.

The `getX()` and `getY()` methods give the (x, y) coordinates of the mouse relative to the event source node. The `getSceneX()` and `getSceneY()` methods give the (x, y) coordinates of the mouse relative to the scene to which the node is added.

The `getScreenX()` and `getScreenY()` methods give the (x, y) coordinates of the mouse relative to the screen to which the node is added.

**Listing 9-6** contains the program to show how to use the methods in the `MouseEvent` class to know the mouse location. It adds a `MOUSE_CLICKED` event handler to the stage, and the stage can receive the notification when the mouse is clicked anywhere in its area. Run the program and click anywhere in the stage, excluding its title bar if you are running it on the desktop. Each mouse click prints a message describing the source, target, and location of the mouse relative to the source, scene, and screen.

### **Listing 9-6.** Determining the Mouse Location During Mouse Events

```
// MouseLocation.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class MouseLocation extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle (50, 50, 50);
        circle.setFill(Color.CORAL);

        Rectangle rect = new Rectangle(100, 100);
        rect.setFill(Color.TAN);

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(circle, rect);

        // Add a MOUSE_CLICKED event handler to the stage
        stage.addEventHandler(MouseEvent.MOUSE_CLICKED, e ->
            handleMouseMove(e));

        Scene scene = new Scene(root);
    }

    void handleMouseMove(MouseEvent e) {
        System.out.println("Source: " + e.getSource());
        System.out.println("Target: " + e.getTarget());
        System.out.println("X: " + e.getX());
        System.out.println("Y: " + e.getY());
        System.out.println("Scene X: " + e.getSceneX());
        System.out.println("Scene Y: " + e.getSceneY());
        System.out.println("Screen X: " + e.getScreenX());
        System.out.println("Screen Y: " + e.getScreenY());
    }
}
```

```

        stage.setScene(scene);
        stage.setTitle("Mouse Location");
        stage.show();
    }

    public void handleMouseMove(MouseEvent e) {
        String source = e.getSource().getClass().getSimpleName();
        String target = e.getTarget().getClass().getSimpleName();

        // Mouse location relative to the event source
        double sourceX = e.getX();
        double sourceY = e.getY();

        // Mouse location relative to the scene
        double sceneX = e.getSceneX();
        double sceneY = e.getSceneY();

        // Mouse location relative to the screen
        double screenX = e.getScreenX();
        double screenY = e.getScreenY();

        System.out.println("Source=" + source + ", Target=" + target +
            ", Location:" +
            " source(" + sourceX + ", " + sourceY + ")" +
            ", scene(" + sceneX + ", " + sceneY + ")" +
            ", screen(" + screenX + ", " + screenY + ")");
    }
}

```

## Representing Mouse Buttons

Typically, a mouse has three buttons. You will also find some that have only one or two buttons. Some platforms provide ways to simulate the missing mouse buttons. The `MouseButton` enum in the `javafx.scene.input` package contains constants to represent mouse button. Table 9-2 contains the list of constants defined in the `MouseButton` enum.

**Table 9-2.** Constants for the `MouseButton` Enum

<b>MouseButton</b> Enum Constant	Description
NONE	It represents no button.
PRIMARY	It represents the primary button. Usually it is the left button in the mouse.
MIDDLE	It represents the middle button.
SECONDARY	It represents the secondary button. Usually it is the right button in the mouse.

The location of the primary and second mouse buttons depends on the mouse configuration. Typically, for right-handed users, the left and right buttons are configured as the primary and secondary buttons, respectively. For the left-handed users, the buttons are configured in the reverse order. If you have a two-button mouse, you do not have a middle button.

## State of Mouse Buttons

The `MouseEvent` object that represents a mouse event contains the state of the mouse buttons at the time the event occurs. The `MouseEvent` class contains many methods to report the state of mouse buttons. Table 9-3 contains a list of such methods with their descriptions.

**Table 9-3.** Methods Related to the State of Mouse Buttons in the `MouseEvent` Class

Method	Description
<code>MouseButton getButton()</code>	It returns the mouse button responsible for the mouse event.
<code>int getClickCount()</code>	It returns the number of mouse clicks associated with the mouse event.
<code>boolean isPrimaryButtonDown()</code>	It returns <code>true</code> if the primary button is currently pressed. Otherwise, it returns <code>false</code> .
<code>boolean isMiddleButtonDown()</code>	It returns <code>true</code> if the middle button is currently pressed. Otherwise, it returns <code>false</code> .
<code>boolean isSecondaryButtonDown()</code>	It returns <code>true</code> if the secondary button is currently pressed. Otherwise, it returns <code>false</code> .
<code>boolean isPopupTrigger()</code>	It returns <code>true</code> if the mouse event is the pop-up menu trigger event for the current platform. Otherwise, it returns <code>false</code> .
<code>boolean isStillSincePress()</code>	It returns <code>true</code> if the mouse cursor stays within a small area, which is called the system-provided hysteresis area, between the last mouse-pressed event and the current mouse event.

In many circumstances, the `getButton()` method may return `MouseButton.NONE`, for example, when a mouse event is triggered on a touch screen by using the fingers instead of a mouse or when a mouse event, such as a mouse-moved event, is not triggered by a mouse button.

It is important to understand the difference between the `getButton()` method and other methods, for example, `isPrimaryButtonDown()`, which returns the pressed state of buttons. The `getButton()` method returns the button that triggers the event. Not all mouse events are triggered by buttons. For example, a mouse-move event is triggered when the mouse moves, not by pressing or releasing a button. If a button is not responsible for a mouse event, the `getButton()` method returns `MouseButton.NONE`. The `isPrimaryButtonDown()` method returns `true` if the primary button is currently pressed, whether or not it triggered the event. For example, when you press the primary button, the mouse-pressed event

occurs. The `getButton()` method will return `MouseButton.PRIMARY` because this is the button that triggered the mouse-pressed event.

The `isPrimaryButtonDown()` method returns `true` because this button is pressed when the mouse-pressed event occurs. Suppose you keep the primary button pressed and you press the secondary button. Another mouse-pressed event occurs. However, this time, the `getButton()` returns `MouseButton.SECONDARY` and both `isPrimaryButtonDown()` and `isSecondaryButtonDown()` methods return `true`, because both of these buttons are in the pressed state at the time of the second mouse-pressed event.

A *pop-up* menu, also known as a *context*, *contextual*, or *shortcut* menu, is a menu that gives a user a set of choices that are available in a specific context in an application. For example, when you click the right mouse button in a browser on the Windows platform, a pop-up menu is displayed. Different platforms trigger pop-up menu events differently upon use of a mouse or keyboard. On the Windows platform, typically it is a right-mouse click or Shift + F10 key press.

The `isPopupTrigger()` method returns `true` if the mouse event is the pop-up menu trigger event for the platform. Otherwise, it returns `false`. If you perform an action based on the returned value of this method, you need to use it in both mouse-pressed and mouse-released events. Typically, when this method returns `true`, you let the system display the default pop-up menu.

**Tip** JavaFX provides a *context menu event* that is a specific type of input event. It is represented by the `ContextMenuEvent` class in the `javafx.scene.input` package. If you want to handle context menu events, use `ContextMenuEvent`.

## Hysteresis in GUI Applications

Hysteresis is a feature that allows user inputs to be within a range of time or location. The time range within which user inputs are accepted is known as the *hysteresis time*. The area in which user inputs are accepted is known as the *hysteresis area*. Hysteresis time and area are system dependent. For example, modern GUI applications provide features that are invoked by double-clicking a mouse button. A time gap exists between two clicks. If the time gap is within the hysteresis time of the system, two clicks are considered a double-click. Otherwise, they are considered two separate single clicks.

Typically, during a mouse-click event, the mouse is moved by a very tiny distance between the mouse-pressed and mouse-released events. Sometimes it is important to take into account the distance the mouse is moved during a mouse click.

The `isStillSincePress()` method returns `true` if the mouse stays in the system-provided hysteresis area since the last mouse-pressed event and the current event. This method is important when you want to consider a mouse-drag action. If this method returns `true`, you may ignore mouse drags as the mouse movement is still within the hysteresis distance from the point where the mouse was last pressed.

## State of Modifier Keys

A modifier key is used to change the normal behavior of other keys. Some examples of modifier keys are Alt, Shift, Ctrl, Meta, Caps Lock, and Num Lock. Not all platforms support all modifier keys. The Meta key is present on Mac, not on Windows. Some systems let you simulate the functionality of a modifier key even if the modifier key is physically not present, for example, you can use the Windows key

on Windows to work as the Meta key. The `MouseEvent` method contains methods to report the pressed state of some of the modifier keys when the mouse event occurs. Table 9-4 lists the methods related to the modifier keys in the `MouseEvent` class.

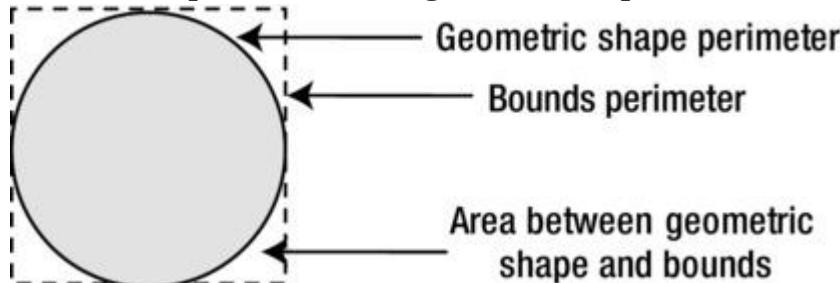
**Table 9-4.** Methods, Related to the State of Modifier Keys, in the `MouseEvent` Class

Method	Description
<code>boolean isAltDown()</code>	It returns <code>true</code> if the Alt key is down for this mouse event. Otherwise, it returns <code>false</code> .
<code>boolean isControlDown()</code>	It returns <code>true</code> if the Ctrl key is down for this mouse event. Otherwise, it returns <code>false</code> .
<code>boolean isMetaDown()</code>	It returns <code>true</code> if the Meta key is down for this mouse event. Otherwise, it returns <code>false</code> .
<code>boolean isShiftDown()</code>	It returns <code>true</code> if the Shift key is down for this mouse event. Otherwise, it returns <code>false</code> .
<code>boolean isShortcutDown()</code>	It returns <code>true</code> if the platform-specific shortcut key is down for this mouse event. Otherwise, it returns <code>false</code> . The shortcut modifier key is the Ctrl key on Windows and the Meta key on Mac.

### Picking Mouse Events on Bounds

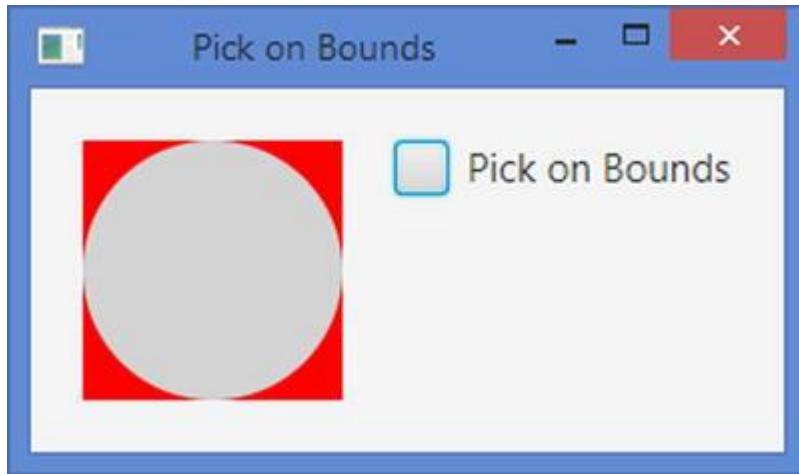
The `Node` class has a `pickOnBounds` property to control the way mouse events are picked (or generated) for a node. A node can have any geometric shape, whereas its bounds always define a rectangular area. If the property is set to true, the mouse events are generated for the node if the mouse is on the perimeter or inside of its bounds. If the property is set to false, which is the default value, mouse events are generated for the node if the mouse is on the perimeter or inside of its geometric shape. Some nodes, such as the `Text` node, have the default value for the `pickOnBounds` property set to true.

Figure 9-8 shows the perimeter for the geometric shape and bounds of a circle. If the `pickOnBounds` property for the circle is false, the mouse event will not be generated for the circle if the mouse is one of the four areas in the corners that lie between the perimeter of the geometric shape and bounds.



**Figure 9-8.** Difference between the geometric shape and bounds of a circle

**Listing 9-7** contains the program to show the effects of the `pickOnBounds` property of a `Circle` node. It displays a window as shown in Figure 9-9. The program adds a `Rectangle` and a `Circle` to a `Group`. Note that the `Rectangle` is added to the `Group` before the `Circle` to keep the former below the latter in Z-order.



**Figure 9-9.** Demonstrating the effects of the `pickOnBounds` property of a `Circle` node

The `Rectangle` uses red as the fill color, whereas light gray is used as the fill color for the `Circle`. The area in red is the area between the perimeters of the geometric shape and bounds of the `Circle`.

You have a check box that controls the `pickOnBounds` property of the circle. If it is selected, the property is set to true. Otherwise, it is set to false.

When you click the gray area, `Circle` always picks up the mouse-clicked event. When you click the red area with the check box unselected, the `Rectangle` picks up the event. When you click the red area with the check box selected, the `Circle` picks up the event. The output shows who picks up the mouse-clicked event.

### **Listing 9-7.** Testing the Effects of the `pickOnBounds` Property for a `Circle` Node

```
// PickOnBounds.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class PickOnBounds extends Application {
    private CheckBox pickonBoundsCbx = new CheckBox("Pick on Bounds");
    Circle circle = new Circle(50, 50, 50, Color.LIGHTGRAY);
```

```

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

@Override
public void start(Stage stage) {
    Rectangle rect = new Rectangle(100, 100);
    rect.setFill(Color.RED);

    Group group = new Group();
    group.getChildren().addAll(rect, circle);

    HBox root = new HBox();
    root.setPadding(new Insets(20));
    root.setSpacing(20);
    root.getChildren().addAll(group, pickonBoundsCbx);

    // Add MOUSE_CLICKED event handlers to the circle and rectangle
    circle.setOnMouseClicked(e -> handleMouseClicked(e));
    rect.setOnMouseClicked(e -> handleMouseClicked(e));

    // Add an Action handler to the checkbox
    pickonBoundsCbx.setOnAction(e -> handleActionEvent(e));

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Pick on Bounds");
    stage.show();
}

public void handleMouseClicked(MouseEvent e) {
    String target = e.getTarget().getClass().getSimpleName();
    String type = e.getEventType().getName();
    System.out.println(type + " on " + target);
}

public void handleActionEvent(ActionEvent e) {
    if (pickonBoundsCbx.isSelected()) {
        circle.setPickOnBounds(true);
    } else {
        circle.setPickOnBounds(false);
    }
}
}

```

## Mouse Transparency

The `Node` class has a `mouseTransparent` property to control whether or not a node and its children receive mouse events. Contrast

the `pickOnBounds` and `mouseTransparent` properties: The former determines the area of a node that generates mouse events, and the latter determines whether or not a node and its children generate mouse events, irrespective of the value of the former. The former affects only the node on which it is set; the latter affects the node on which it is set and all its children.

The code in Listing 9-8 shows the effects of the `mouseTransparent` property of a `Circle`. This is a variant of the program in Listing 9-7. It displays a window that is very similar to the one shown in Figure 9-9. When the check box `MouseTransparency` is selected, it sets the `mouseTransparent` property of the circle to `true`. When the check box is unselected, it sets the `mouseTransparent` property of the circle to `false`.

Click the circle, in the gray area, when the check box is selected and all mouse-clicked events will be delivered to the rectangle. This is because the circle is mouse-transparent and it lets the mouse events pass through. Unselect the check box, and all mouse-clicks in the gray area are delivered to the circle. Note that clicking the red area always delivers the event to the rectangle, because the `pickOnBounds` property for the circle is set to false by default. The output shows the node that receives the mouse-clicked events.

***Listing 9-8.*** Testing the Effects of the `mouseTransparent` Property for a Circle Node

```
// MouseTransparency.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class MouseTransparency extends Application {
    private CheckBox mouseTransparentCbx = new CheckBox("Mouse Transparent");
    Circle circle = new Circle(50, 50, 50, Color.LIGHTGRAY);

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

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(100, 100);
        rect.setFill(Color.RED);

        Group group = new Group();
        group.getChildren().addAll(rect, circle);

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(group, mouseTransparentCbx);

        // Add MOUSE_CLICKED event handlers to the circle and rectangle
        circle.setOnMouseClicked(e -> handleMouseClicked(e));
        rect.setOnMouseClicked(e -> handleMouseClicked(e));

        // Add an Action handler to the checkbox
        mouseTransparentCbx.setOnAction(e -> handleActionEvent(e));

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Mouse Transparency");
        stage.show();
    }
}
```

```

public void handleMouseClicked(MouseEvent e) {
    String target = e.getTarget().getClass().getSimpleName();
    String type = e.getEventType().getName();
    System.out.println(type + " on " + target);
}

public void handleActionEvent(ActionEvent e) {
    if (mouseTransparentCbx.isSelected()) {
        circle.setMouseTransparent(true);
    } else {
        circle.setMouseTransparent(false);
    }
}
}

```

## Synthesized Mouse Events

A mouse event can be generated using several types of devices, such as a mouse, track pad, or touch screen. Some actions on a touch screen generate mouse events, which are considered *synthesized mouse events*. The `isSynthesized()` method of the `MouseEvent` class returns `true` if the event is synthesized from using a touch screen. Otherwise, it returns `false`.

When a finger is dragged on a touch screen, it generates both a scrolling gesture event and a mouse-drag event. The return value of the `isSynthesized()` method can be used inside the mouse-drag event handlers to detect if the event is generated by dragging a finger on a touch screen or by dragging a mouse.

## Handling Mouse Entered and Exited Events

Four mouse event types deal with events when the mouse enters or exits a node:

- `MOUSE_ENTERED`
- `MOUSE_EXITED`
- `MOUSE_ENTERED_TARGET`
- `MOUSE_EXITED_TARGET`

You have two sets of event types for mouse-entered and mouse-exited events. One set contains two types called `MOUSE_ENTERED` and `MOUSE_EXITED` and another set contains `MOUSE_ENTERED_TARGET` and `MOUSE_EXITED_TARGET`. They both have something in common, such as when they are triggered. They differ in their delivery mechanisms. I will discuss all of them this section.

When the mouse enters a node, a `MOUSE_ENTERED` event is generated. When the mouse leaves a node, a `MOUSE_EXITED` event is generated. These events do not go through the capture and bubbling phases. That is, they are delivered directly to the target node, not to any of its parent nodes.

**Tip** The `MOUSE_ENTERED` and `MOUSE_EXITED` events do not participate in the capture and bubbling phases. However, all event *filters* and *handlers* are executed for the target following the rules for event handling.

The program in Listing 9-9 shows how mouse-entered and mouse-exited events are delivered. The program displays a window as shown in Figure 9-10. It shows a circle with gray fill inside an `HBox`. Event handlers for mouse-entered and mouse-exited events are added to the `HBox` and the `Circle`. Run the program and move the

mouse in and out of the circle. When the mouse enters the white area in the window, its `MOUSE_ENTERED` event is delivered to the `HBox`. When you move the mouse in and out of the circle, the output shows that

the `MOUSE_ENTERED` and `MOUSE_EXITED` events are delivered only to the `Circle`, not to the `HBox`. Notice that in the output the source and target of these events are always the same, proving that the capture and bubbling phases do not occur for these events. When you move the mouse in and out of the circle, keeping it in the white area, the `MOUSE_EXITED` event for the `HBox` does not fire, as the mouse stays on the `HBox`. To fire the `MOUSE_EXITED` event on the `HBox`, you will need to move the mouse outside the scene area, for example, outside the window or over the title bar of the window.

### ***Listing 9-9.*** Testing Mouse-Entered and Mouse-Exited Events

```
// MouseEnteredExited.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.event.EventHandler;
import javafx.stage.Stage;
import static javafx.scene.input.MouseEvent.MOUSE_ENTERED;
import static javafx.scene.input.MouseEvent.MOUSE_EXITED;

public class MouseEnteredExited extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle circle = new Circle (50, 50, 50);
        circle.setFill(Color.GRAY);

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(circle);

        // Create a mouse event handler
        EventHandler<MouseEvent> handler = e -> handle(e);

        // Add mouse-entered and mouse-exited event handlers to the HBox
        root.addEventHandler(MOUSE_ENTERED, handler);
        root.addEventHandler(MOUSE_EXITED, handler);

        // Add mouse-entered and mouse-exited event handlers to the
        Circle
        circle.addEventHandler(MOUSE_ENTERED, handler);
        circle.addEventHandler(MOUSE_EXITED, handler);

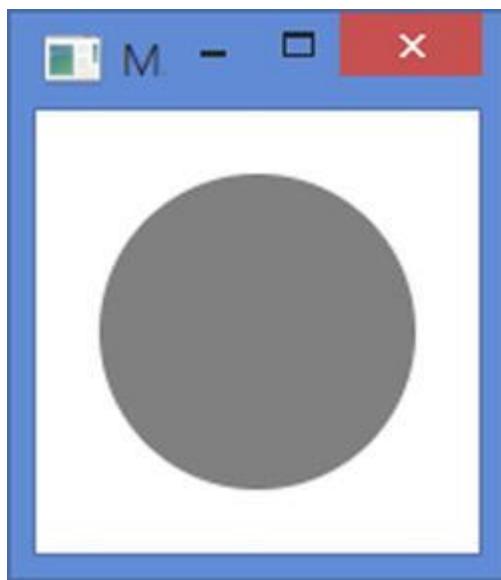
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Mouse Entered and Exited Events");
        stage.show();
    }
}
```

```

public void handle(MouseEvent e) {
    String type = e.getEventType().getName();
    String source = e.getSource().getClass().getSimpleName();
    String target = e.getTarget().getClass().getSimpleName();
    System.out.println("Type=" + type + ", Target=" + target + ", "
Source=" + source);
}
}

Type=MOUSE_ENTERED, Target=HBox, Source=HBox
Type=MOUSE_ENTERED, Target=Circle, Source=Circle
Type=MOUSE_EXITED, Target=Circle, Source=Circle
Type=MOUSE_ENTERED, Target=Circle, Source=Circle
Type=MOUSE_EXITED, Target=Circle, Source=Circle
Type=MOUSE_EXITED, Target=HBox, Source=HBox
...

```



**Figure 9-10.** Demonstrating mouse-entered and mouse-exited events

The `MOUSE_ENTERED` and `MOUSE_EXITED` event types provide the functionality needed in most cases. Sometimes you need these events to go through the normal capture and bubbling phases, so parent nodes can apply filters and provide default responses. The `MOUSE_ENTERED_TARGET` and `MOUSE_EXITED_TARGET` event types provide these features. They participate in the event capture and bubbling phases.

The `MOUSE_ENTERED` and `MOUSE_EXITED` event types are subtypes of the `MOUSE_ENTERED_TARGET` and `MOUSE_EXITED_TARGET` event types. A node interested in the mouse-entered event of its children should add event filters and handlers for the `MOUSE_ENTERED_TARGET` type. The child node can add `MOUSE_ENTERED`, `MOUSE_ENTERED_TARGET`, or both event filters and handlers. When the mouse enters the child node, parent nodes receive the `MOUSE_ENTERED_TARGET` event. Before the event is delivered to the child node, which is the target node of the event, the event type is changed to the `MOUSE_ENTERED` type. Therefore, in the same event processing, the target node receives the `MOUSE_ENTERED` event, whereas all its parent nodes receive the `MOUSE_ENTERED_TARGET` event. Because the `MOUSE_ENTERED` event type is a subtype of the `MOUSE_ENTERED_TARGET` type, either type of event handler on the

target can handle this event. The same would apply to the mouse-exited event and its corresponding event types.

Sometimes, inside the parent event handler, it is necessary to distinguish the node that fires the `MOUSE_ENTERED_TARGET` event. A parent node receives this event when the mouse enters the parent node itself or any of its child nodes. You can check the target node reference, using the `getTarget()` method of the `Event` class, for equality with the reference of the parent node, inside the event filters and handlers, to know whether or not the event was fired by the parent.

The program in Listing 9-10 shows how to use the mouse-entered-target and mouse-exited-target events. It adds a `Circle` and a `CheckBox` to an `HBox`. The `HBox` is added to the `Scene`. It adds the mouse-entered-target and mouse-exited-target event filters to the `HBox` and event handlers to the `Circle`. It also adds mouse-entered and mouse-exited event handlers to the `Circle`. When the check box is selected, events are consumed by the `HBox`, so they do not reach the `Circle`. Below are a few observations when you run the program:

- With the check box unselected, when the mouse enters or leaves the `Circle`, the `HBox` receives the `MOUSE_ENTERED_TARGET` and `MOUSE_EXITED_TARGET` events. The `Circle` receives the `MOUSE_ENTERED` and `MOUSE_EXITED` events.
- With the check box selected, the `HBox` receives the `MOUSE_ENTERED_TARGET` and `MOUSE_EXITED_TARGET` events and consumes them. The `Circle` does not receive any events.
- When the mouse enters or leaves the `HBox`, the white area in the window, the `HBox` receives the `MOUSE_ENTERED` and `MOUSE_EXITED` events, because the `HBox` is the target of the event.

Play with the application by moving the mouse around, selecting and unselecting the check box. Look at the output to get a feel for how these events are processed.

#### ***Listing 9-10.*** Using the Mouse-Entered-Target and Mouse-Exited-Target Events

```
// MouseEnteredExitedTarget.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.input.MouseEvent;
import static javafx.scene.input.MouseEvent.MOUSE_ENTERED;
import static javafx.scene.input.MouseEvent.MOUSE_EXITED;
import static javafx.scene.input.MouseEvent.MOUSE_ENTERED_TARGET;
import static javafx.scene.input.MouseEvent.MOUSE_EXITED_TARGET;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class MouseEnteredExitedTarget extends Application {
    private CheckBox consumeCbx = new CheckBox("Consume Events");
    
```

```

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

@Override
public void start(Stage stage) {
    Circle circle = new Circle(50, 50, 50);
    circle.setFill(Color.GRAY);

    HBox root = new HBox();
    root.setPadding(new Insets(20));
    root.setSpacing(20);
    root.getChildren().addAll(circle, consumeCbx);

    // Create mouse event handlers
    EventHandler<MouseEvent> circleHandler = e -> handleCircle(e);
    EventHandler<MouseEvent> circleTargetHandler =
        e -> handleCircleTarget(e);
    EventHandler<MouseEvent> hBoxTargetHandler = e ->
handleHBoxTarget(e);

    // Add mouse-entered-target and mouse-exited-target event
    // handlers to HBox
    root.addEventFilter(MOUSE_ENTERED_TARGET, hBoxTargetHandler);
    root.addEventFilter(MOUSE_EXITED_TARGET, hBoxTargetHandler);

    // Add mouse-entered-target and mouse-exited-target event
    // handlers to the Circle
    circle.addHandler(MOUSE_ENTERED_TARGET,
circleTargetHandler);
    circle.addHandler(MOUSE_EXITED_TARGET, circleTargetHandler);

    // Add mouse-entered and mouse-exited event handlers to the
Circle
    circle.addHandler(MOUSE_ENTERED, circleHandler);
    circle.addHandler(MOUSE_EXITED, circleHandler);

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Mouse Entered Target and Exited Target Events");
    stage.show();
}

public void handleCircle(MouseEvent e) {
    print(e, "Circle Handler");
}

public void handleCircleTarget(MouseEvent e) {
    print(e, "Circle Target Handler");
}

public void handleHBoxTarget(MouseEvent e) {
    print(e, "HBox Target Filter");
    if (consumeCbx.isSelected()) {
        e.consume();
        System.out.println("HBox consumed the " + e.getEventType()
+ " event");
    }
}

public void print(MouseEvent e, String msg) {
    String type = e.getEventType().getName();
    String source = e.getSource().getClass().getSimpleName();
    String target = e.getTarget().getClass().getSimpleName();
    System.out.println(msg + ": Type=" + type +

```

```

        ", Target=" + target +
        ", Source=" + source);
    }
}

```

## Handling Key Events

A key event is a type of input event that denotes the occurrence of a keystroke. It is delivered to the node that has focus. An instance of the `KeyEvent` class, which is declared in the `javafx.scene.input` package, represents a key event. Key pressed, key released, and key typed are three types of key events. Table 9-5 lists all of the constants in the `KeyEvent` class, which represent key event types.

**Table 9-5.** Constants in the `KeyEvent` Class to Represent Key Event Types

Constant	Description
ANY	It is the supertype of other key events types.
KEY_PRESSED	It occurs when a key is pressed.
KEY_RELEASED	It occurs when a key is released.
KEY_TYPED	It occurs when a Unicode character is entered.

**Tip** It may not be obvious that shapes, for example circles or rectangles, can also receive key events. The criterion for a node to receive key events is that the node should have focus. By default, shapes are not part of the focus traversal chain and mouse clicks do not bring focus to them. Shape nodes can get focus by calling the `requestFocus()` method.

The key-pressed and key-released events are lower-level events compared to the key-typed event; they occur with a key press and release, respectively, and depend of the platform and keyboard layout.

The key-typed event is a higher-level event. Generally, it does not depend on the platform and keyboard layout. It occurs when a Unicode character is typed. Typically, a key press generates a key-typed event. However, a key release may also generate a key-typed event. For example, when using the Alt key and number pad on Windows, a key-typed event is generated by the release of the Alt key, irrespective of the number of keystrokes entered on the number pad. A key-typed event can also be generated by a series of key presses and releases. For example, the character A is entered by pressing Shift + A, which includes two key presses (Shift and A). In this case, two key presses generate one key-typed event. Not all key presses or releases generate key-typed events. For example, when you press a function key (F1, F2, etc.) or modifier keys (Shift, Ctrl, etc.), no Unicode character is entered, and hence, no key-typed event is generated.

The `KeyEvent` class maintains three variables to describe the keys associated with the event: code, text, and character. These variables can be accessed using the getter methods in the `KeyEvent` class as listed in Table 9-6.

**Table 9-6.** Methods in the `KeyEvent` Class Returning Key Details

Method	Valid for	Description
KeyCode getCode ()	KEY_PRESSED KEY_RELEASED	The <code>KeyCode</code> enum contains a constant to represent all keys on a keyboard. This method returns the <code>KeyCode</code> enum constant associated with the key being pressed or released. For the key-typed events, it always returns <code>KeyCode.UNDEFINED</code> , because the key-type events are not necessarily triggered by a single keystroke.
String getText ()	KEY_PRESSED KEY_RELEASED	It returns a <code>String</code> description of the <code>KeyCode</code> associated with the key-pressed and key-released events. It always returns an empty string for key-typed events.
String getCharacter ()	KEY_TYPED	It returns a character or a sequence of character associated with the key-typed event as a <code>String</code> . For the key-pressed and key-released events, it always returns <code>KeyEvent.CHAR_UNDEFINED</code> .

It is interesting to note that the return type of the `getCharacter()` method is `String`, not `char`. The design is intentional. Unicode characters outside the basic multilingual plane cannot be represented in one character. Some devices may produce multiple characters using a single keystroke. The return type of `String` for the `getCharacter()` method covers these odd cases.

The `KeyEvent` class contains `isAltDown()`, `isControlDown()`, `isMetaDown()`, `isShiftDown()`, and `isShortcutDown()` methods that let you check whether modifier keys are down when a key event occurs.

### Handling Key-pressed and Key-released Events

Key-pressed and key-released events are handled simply by adding the event filters and handlers to nodes for the `KEY_PRESSED` and `KEY_RELEASED` event types. Typically you use these events to know which keys were pressed or released and to perform an action. For example, you can detect the F1 function key press and display a custom Help window for the node in focus.

The program in Listing 9-11 shows how to handle key-pressed and key-released events. It displays a `Label` and a `TextField`. When you run the program, the `TextField` has focus. Notice the following points when you use keystrokes while running this program:

- Press and release some keys. Output will show the details of events as they occur. A key-released event does not occur for every key-pressed event.
- The mapping between key-pressed and key-released events is not one-to-one. There may be no key-released event for a key-pressed event (refer to the next item). There may be one key-released event for several key-pressed events. This can happen when you keep a key

pressed for a longer period. Sometimes you do it to type the same character multiple times. Press the A key and hold it for some time and then release it. This will generate several key-pressed events and only one key-released event.

- Press the F1 key. It will display the Help window. Notice that pressing the F1 key does not generate an output for a key-released event, even after you release the key. Can you think of the reason for this? On the key-pressed event, the Help window is displayed, which grabs the focus. The `TextField` on the main window no longer has focus. Recall that the key events are delivered to the node that has focus, and only one node can have focus in a JavaFX application. Therefore, the key-released event is delivered to the Help window, not the `TextField`.

**Listing 9-11.** Handling Key-pressed and Key-released Events

```

        // Show the help window when the F1 key is pressed
        if (e.getEventType() == KEY_PRESSED && e.getCode() == KeyCode.F1)
    {
        displayHelp();
        e.consume();
    }
}

public void displayHelp() {
    Text helpText = new Text("Please enter a name.");
    HBox root = new HBox();
    root.setStyle("-fx-background-color: yellow;");
    root.getChildren().add(helpText);

    Scene scene = new Scene(root, 200, 100);
    Stage helpStage = new Stage();
    helpStage.setScene(scene);
    helpStage.setTitle("Help");
    helpStage.show();
}
}

```

## Handling the Key-typed Event

The typical use of the key-typed event is to detect specific keystrokes to prevent some characters from being entered. For example, you may allow users to only enter letters in a name field. You can do so by consuming all key-typed events for the field associated with all nonletters.

The program in Listing 9-12 shows a Label and a TextField. It adds a key-typed event handler to the TextField, which consumes the event if the character typed is not a letter. Otherwise, it prints the character typed on the standard output. Run the program. You should be able to enter letters in the TextField. When you press any nonletter keys, for example, 1, 2, 3, nothing happens.

This example is not a correct solution to stop users from entering nonletter characters. For example, users can still paste nonletters using the context menu (right-click on Windows) or using the keyboard shortcut Ctrl + V. The correct solution lies in detecting and handling the event on the TextField that is generated, irrespective of the method used. For now, this example serves the purpose of showing how to use key-typed events.

### **Listing 9-12.** Using the Key-typed Event

```

// KeyTyped.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class KeyTyped extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}

```

```

@Override
public void start(Stage stage) {
    Label nameLbl = new Label("Name:");
    TextField nameTfl = new TextField();

    HBox root = new HBox();
    root.setPadding(new Insets(20));
    root.setSpacing(20);
    root.getChildren().addAll(nameLbl, nameTfl);

    // Add key-typed event to the TextField
    nameTfl.setOnKeyTyped(e -> handle(e));

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Key Typed Event");
    stage.show();
}

public void handle(KeyEvent e) {
    // Consume the event if it is not a letter
    String str = e.getCharacter();
    int len = str.length();
    for(int i = 0; i < len; i++) {
        Character c = str.charAt(i);
        if (!Character.isLetter(c)) {
            e.consume();
        }
    }

    // Print the details if it is not consumed
    if (!e.isConsumed()) {
        String type = e.getEventType().getName();
        System.out.println(type + ": Character=" +
+ e.getCharacter());
    }
}
}

```

## Handling Window Events

A window event occurs when a window is shown, hidden, or closed. An instance of the `WindowEvent` class in the `javafx.stage` package represents a window event. Table 9-7 lists the constants in the `WindowEvent` class.

**Table 9-7.** Constants in the `WindowEvent` Class to Represent Window Event Types

Constant	Description
ANY	It is the supertype of all other window event types.
WINDOW_SHOWING	It occurs just before the window is shown.
WINDOW_SHOWN	It occurs just after the window is shown.
WINDOW HIDING	It occurs just before the window is hidden.

Constant	Description
WINDOW_HIDDEN	It occurs just after the window is hidden.
WINDOW_CLOSE_REQUEST	It occurs when there is an external request to close this window.

The window-showing and window-shown events are straightforward. They occur just before and after the window is shown. Event handlers for the window-showing event should have time-consuming logic, as it will delay showing the window to the user, and hence, degrading the user experience. Initializing some window-level variables is a good example of the kind of code you need to write in this event. Typically, the window-shown event sets the starting direction for the user, for example, setting focus to the first editable field on the window, showing alerts to the user about the tasks that need his attention, among others.

The window-hiding and window-hidden events are counterparts of the window-showing and window-shown events. They occur just before and after the window is hidden.

The window-close-request event occurs when there is an external request to close the window. Using the Close menu from the context menu or the Close icon in the window title bar or pressing Alt + F4 key combination on Windows is considered an external request to close the window. Note that closing a window programmatically, for example, using the `close()` method of the `Stage` class or `Platform.exit()` method, is not considered an external request. If the window-close-request event is consumed, the window is not closed.

The program in Listing 9-13 shows how to use all window events. You may get a different output than that shown below the code. It adds a check box and two buttons to the primary stage. If the check box is unselected, external requests to close the window are consumed, thus preventing the window from closing. The Close button closes the window. The Hide button hides the primary window and opens a new window, so the user can show the primary window again.

The program adds event handlers to the primary stage for window event types. When the `show()` method on the stage is called, the window-showing and window-shown events are generated. When you click the Hide button, the window-hiding and window-hidden events are generated. When you click the button on the pop-up window to show the primary window, the window-showing and window-shown events are generated again. Try clicking the Close icon on the title bar to generate the window-close-request event. If the Can Close Window check box is not selected, the window is not closed. When you use the Close button to close the window, the window-hiding and window-hidden events are generated, but not the window-close-request event, as it is not an external request to close the window.

### ***Listing 9-13.*** Using Window Events

```
// WindowEventApp.java
package com.jdojo.event;

import javafx.application.Application;
import javafx.event.EventType;
import javafx.geometry.Insets;
import javafx.scene.Scene;
```

```

import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.stage.WindowEvent;
import static javafx.stage.WindowEvent.WINDOW_CLOSE_REQUEST;

public class WindowEventApp extends Application {
    private CheckBox canCloseCbx = new CheckBox("Can Close Window");

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

    @Override
    public void start(Stage stage) {
        Button closeBtn = new Button("Close");
        closeBtn.setOnAction(e -> stage.close());

        Button hideBtn = new Button("Hide");
        hideBtn.setOnAction(e -> {showDialog(stage); stage.hide(); });

        HBox root = new HBox();
        root.setPadding(new Insets(20));
        root.setSpacing(20);
        root.getChildren().addAll(canCloseCbx, closeBtn, hideBtn);

        // Add window event handlers to the stage
        stage.setOnShowing(e -> handle(e));
        stage.setOnShown(e -> handle(e));
        stage.setOnHiding(e -> handle(e));
        stage.setOnHidden(e -> handle(e));
        stage.setOnCloseRequest(e -> handle(e));

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

    public void handle(WindowEvent e) {
        // Consume the event if the CheckBox is not selected
        // thus preventing the user from closing the window
        EventType<WindowEvent> type = e.getEventType();
        if (type == WINDOW_CLOSE_REQUEST && !canCloseCbx.isSelected()) {
            e.consume();
        }
    }

    System.out.println(type + ": Consumed=" + e.isConsumed());
}

public void showDialog(Stage mainWindow) {
    Stage popup = new Stage();

    Button closeBtn = new Button("Click to Show Main Window");
    closeBtn.setOnAction(e -> { popup.close(); mainWindow.show(); });

    HBox root = new HBox();
    root.setPadding(new Insets(20));
    root.setSpacing(20);
    root.getChildren().addAll(closeBtn);

    Scene scene = new Scene(root);
    popup.setScene(scene);
    popup.setTitle("Popup");
    popup.show();
}

```

```

        }
WINDOW_SHOWING: Consumed=false
WINDOW_SHOWN: Consumed=false
WINDOW HIDING: Consumed=false
WINDOW_HIDDEN: Consumed=false
WINDOW_SHOWING: Consumed=false
WINDOW_SHOWN: Consumed=false
WINDOW_CLOSE_REQUEST: Consumed=true

```

## Summary

In general, the term event is used to describe an occurrence of interest. In a GUI application, an event is an occurrence of a user interaction with the application such as clicking the mouse, pressing a key on the keyboard, and so forth. An event in JavaFX is represented by an object of the `javafx.event.Event` class or any of its subclasses. Every event in JavaFX has three properties: an event source, an event target, and an event type.

When an event occurs in an application, you typically perform some processing by executing a piece of code. The piece of code that is executed in response to an event is known as an event handler or an event filter. When you want to handle an event for a UI element, you need to add event handlers to the UI element, for example, a `Window`, a `Scene`, or a `Node`. When the UI element detects the event, it executes your event handlers.

The UI element that calls event handlers is the source of the event for those event handlers. When an event occurs, it passes through a chain of event dispatchers. The source of an event is the current element in the event dispatcher chain. The event source changes as the event passes through one dispatcher to another in the event dispatcher chain. The event target is the destination of an event, which determines the route the event travels through during its processing. The event type describes the type of the event that occurs. They are defined in a hierarchical fashion. Each event type has a name and a supertype.

When an event occurs, the following three steps are performed in order: event target selection, event route construction, and event route traversal. An event target is the destination node of the event that is selected based on the event type. An event travels through event dispatchers in an event dispatch chain. The event dispatch chain is the event route. The initial and default route for an event is determined by the event target. The default event route consists of the container-children path starting at the stage to the event target node.

An event route traversal consists of two phases: capture and bubbling. An event travels through each node in its route twice: once during the capture phase and once during the bubbling phase. You can register event filters and event handlers to a node for specific events types. The event filters and event handlers registered to a node are executed as the event passes through the node during the capture and the bubbling phases, respectively.

During the route traversal, a node can consume the event in event filters or handlers, thus completing the processing of the event. Consuming an event is simply calling the `consume()` method on the event object. When an event is consumed, the event processing is stopped, even though some of the nodes in the route were not traversed at all.

Interaction of the user with the UI elements using the mouse, such as clicking, moving, or pressing the mouse, triggers a mouse event. An object of the `MouseEvent` class represents a mouse event.

A key event denotes the occurrence of a keystroke. It is delivered to the node that has focus. An instance of the `KeyEvent` class represents a key event. Key pressed, key released, and key typed are three types of key events.

A window event occurs when a window is shown, hidden, or closed. An instance of the `WindowEvent` class in the `javafx.stage` package represents a window event.

The next chapter discusses layout panes that are used as containers for other controls and nodes.

## CHAPTER 10



### Understanding Layout Panes

In this chapter, you will learn:

- What a layout pane is
- Classes in JavaFX representing layout panes
- How to add children to layout panes
- Utility classes such as Insets, HPos, VPos, Side, Priority, etc.
- How to use a Group to layout nodes
- How to work with Regions and its properties
- How to use different types of layout panes such as HBox, VBox, FlowPane, BorderPane, StackPane, TilePane, GridPane, AnchorPane, and TextFlow

#### What Is a Layout Pane?

You can use two types of layouts to arrange nodes in a scene graph:

- Static layout
- Dynamic layout

In a static layout, the position and size of nodes are calculated once, and they stay the same as the window is resized. The user interface looks good when the window has the size for which the nodes were originally laid out.

In a dynamic layout, nodes in a scene graph are laid out every time a user action necessitates a change in their position, size, or both. Typically, changing the position or size of one node affects the position and size of all other nodes in the scene graph. The dynamic layout forces the recomputation of the position and size of some or all nodes as the window is resized.

Both static and dynamic layouts have advantages and disadvantages. A static layout gives developers full control on the design of the user interface. It lets you make use of the available space as you see fit. A dynamic layout requires more programming work, and the logic is much more involved. Typically, programming languages supporting GUI: for example, JavaFX, supports dynamic layouts through libraries. Libraries solve most of the use-cases for dynamic layouts. If they do not meet your needs, you must do the hard work to roll out your own dynamic layout.

A *layout pane* is a node that contains other nodes, which are known as its children (or child nodes). The responsibility of a layout pane is to lay out its children, whenever needed. A layout pane is also known as a *container* or a *layout container*.

A layout pane has a *layout policy* that controls how the layout pane lays out its children. For example, a layout pane may lay out its children horizontally, vertically, or in any other fashion.

JavaFX contains several layout-related classes, which are the topic of discussion in this chapter. A layout pane performs two things:

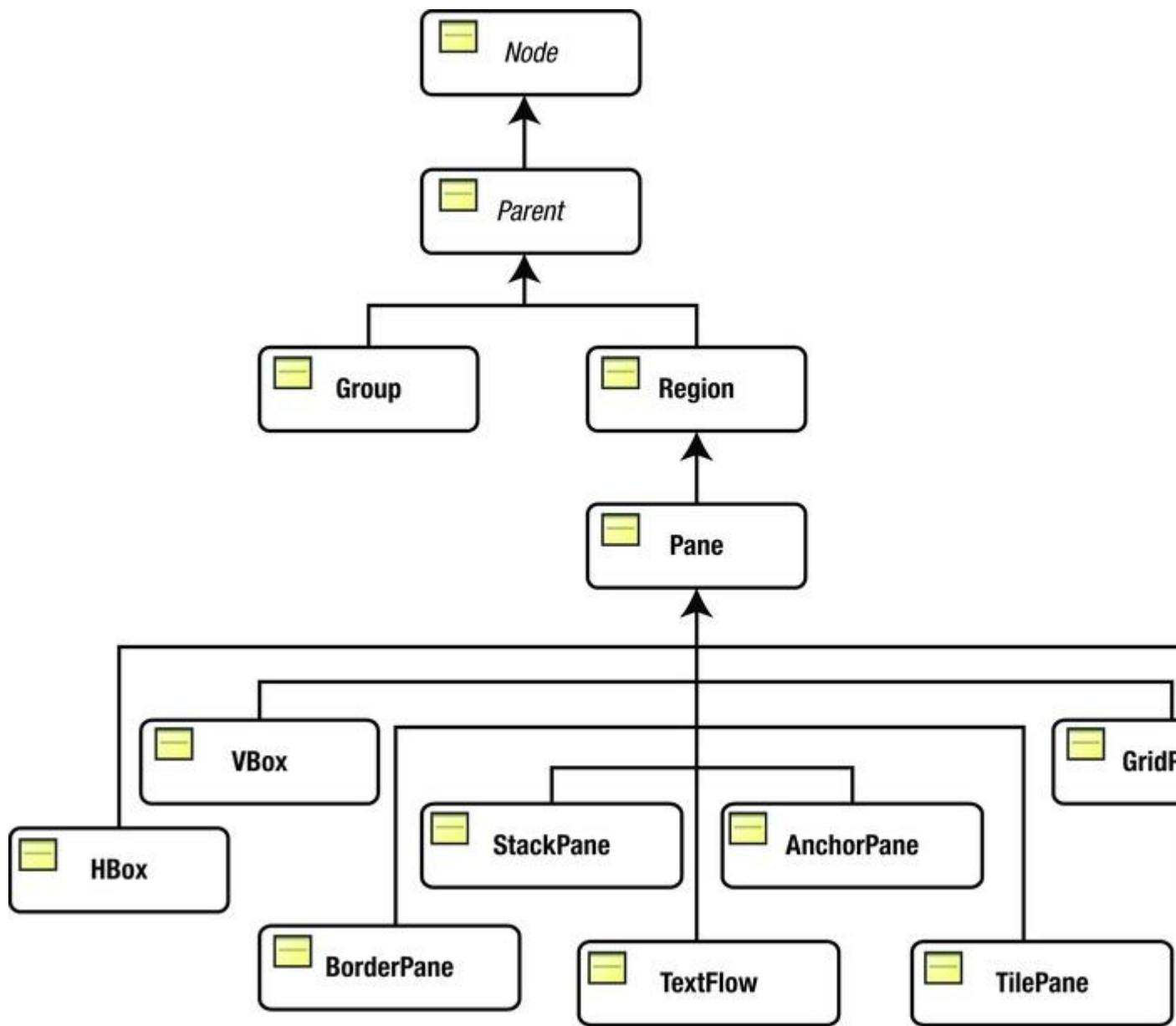
- It computes the position (the x and y coordinates) of the node within its parent.
- It computes the size (the width and height) of the node.

For a 3D node, a layout pane also computes the z coordinate of the position and the depth of the size.

The layout policy of a container is a set of rules to compute the position and size of its children. When I discuss containers in this chapter, pay attention to the layout policy of the containers as to how they compute the position and size of their children. A node has three sizes: preferred size, minimum size, and maximum size. Most of the containers attempt to give its children their preferred size. The actual (or current) size of a node may be different from its preferred size. The current size of a node depends on the size of the window, the layout policy of the container, and the expanding and shrinking policy for the node, etc.

## Layout Pane Classes

JavaFX contains several container classes. Figure 10-1 shows a class diagram for the container classes. A container class is a subclass, direct or indirect, of the `Parent` class.



**Figure 10-1.** A class diagram for container classes in JavaFX

A `Group` lets you apply effects and transformations to all its children collectively. The `Group` class is in the `javafx.scene` package.

Subclasses of the `Region` class are used to lay out children. They can be styled with CSS. The `Region` class and most of its subclasses are in the `javafx.scene.layout` package.

It is true that a container needs to be a subclass of the `Parent` class. However, not all subclasses of the `Parent` class are containers. For example, the `Button` class is a subclass of the `Parent` class; however, it is a control, not a container. A node must be added to a container to be part of a scene graph. The container lays out its children according to its layout policy. If you do not want the container to manage the layout for a node, you need to set the `managed` property of the node to false. Please refer to the chapter on *Understanding Nodes* for more details and examples on managed and unmanaged nodes.

A node can be a child node of only one container at a time. If a node is added to a container while it is already the child node of another container, the node is removed from the first container before being added to the second one. Oftentimes, it is necessary to nest containers to create a complex layout. That is, you can add a container to another container as a child node.

The `Parent` class contains three methods to get the list of children of a container:

- `protected ObservableList<Node> getChildren()`
- `public ObservableList<Node> getChildrenUnmodifiable()`
- `protected <E extends Node> List<E> getManagedChildren()`

The `getChildren()` method returns a modifiable `ObservableList` of the child nodes of a container. If you want to add a node to a container, you would add the node to this list. This is the most commonly used method for container classes. We have been using this method to add children to containers such as `Group`, `HBox`, `VBox`, etc., from the very first program.

Notice the `protected` access for the `getChildren()` method. If a subclass of the `Parent` class does not want to be a container, it will keep the access for this method as `protected`. For example, control-related classes (`Button`, `TextField`, etc.) keep this method as `protected`, so you cannot add child nodes to them. A container class overrides this method and makes it `public`. For example, the `Group` and `Pane` classes expose this method as `public`.

The `getChildrenUnmodifiable()` method is declared `public` in the `Parent` class. It returns a read-only `ObservableList` of children. It is useful in two scenarios:

- You need to pass the list of children of a container to a method that should not modify the list.
- You want to know what makes up a control, which is not a container.

The `getManagedChildren()` method has the `protected` access. Container classes do not expose it as `public`. They use it internally to get the list of managed children, during layouts. You will use this method to roll out your own container classes.

Table 10-1 has brief descriptions of the container classes. We will discuss them in detail with examples in subsequent sections.

**Table 10-1. List of Container Classes**

Container Class	Description
<code>Group</code>	A <code>Group</code> applies effects and transformations collectively to all its children.
<code>Pane</code>	It is used for absolute positioning of its children.

<b>Container Class</b>	<b>Description</b>
HBox	It arranges its children horizontally in a single row.
VBox	It arranges its children vertically in a single column.
FlowPane	It arranges its children horizontally or vertically in rows or columns. If they do not fit in column, they are wrapped at the specified width or height.
BorderPane	It divides its layout area in the top, right, bottom, left, and center regions and places each in one of the five regions.
StackPane	It arranges its children in a back-to-front stack.
TilePane	It arranges its children in a grid of uniformly sized cells.
GridPane	It arranges its children in a grid of variable sized cells.
AnchorPane	It arranges its children by anchoring their edges to the edges of the layout area.
TextFlow	It lays out rich text whose content may consist of several Text nodes.

## Adding Children to a Layout Pane

A container is meant to contain children. You can add children to a container when you create the container object or after creating it. All container classes provide constructors that take a var-args `Node` type argument to add the initial set of children. Some containers provide constructors to add an initial set of children and set initial properties for the containers.

You can also add children to a container at any time after the container is created. Containers store their children in an observable list, which can be retrieved using the `getChildren()` method. Adding a node to a container is as simple as adding a node to that observable list. The following snippet of code shows how to add children to an `HBox` when it is created and after it is created.

```
// Create two buttons
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");

// Create an HBox with two buttons as its children
HBox hBox1 = new HBox(okBtn, cancelBtn);

// Create an HBox with two buttons with 20px horizontal spacing between them
double hSpacing = 20;
HBox hBox2 = new HBox(hSpacing, okBtn, cancelBtn);
```

```
// Create an empty HBox, and afterwards, add two buttons to it
HBox hBox3 = new HBox();
hBox3.getChildren().addAll(okBtn, cancelBtn);
```

**Tip** When you need to add multiple child nodes to a container, use the `addAll()` method of the `ObservableList` rather than using the `add()` method multiple times.

## Utility Classes and Enums

While working with layout panes, you will need to use several classes and enums that are related to spacing and directions. These classes and enums are not useful when used stand-alone. They are always used as properties for nodes. This section describes some of these classes and enums.

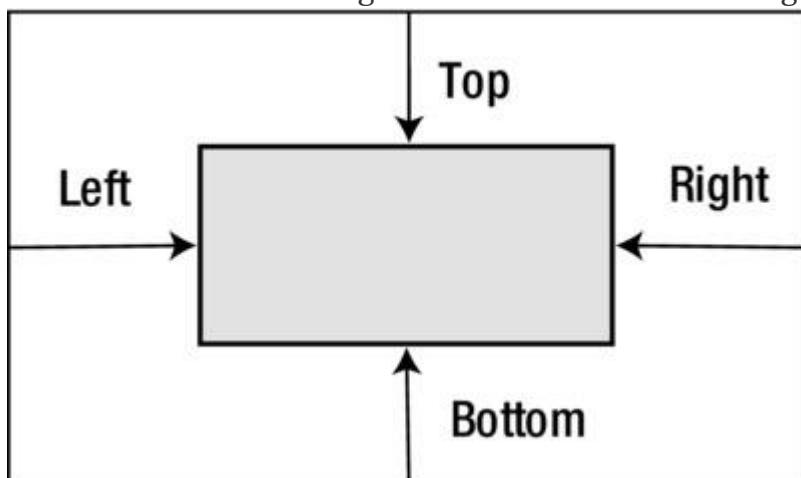
### The Insets Class

The `Insets` class represents inside offsets in four directions: top, right, bottom, and left, for a rectangular area. It is an immutable class. It has two constructors – one lets you set the same offset for all four directions and another lets you set different offsets for each direction.

- `Insets(double topRightBottomLeft)`
- `Insets(double top, double right, double bottom, double left)`

The `Insets` class declares a constant, `Insets.EMPTY`, to represent a zero offset for all four directions. Use the `getTop()`, `getRight()`, `getBottom()`, and `getLeft()` methods to get the value of the offset in a specific direction.

It is a bit confusing to understand the exact meaning of the term *insets* by looking at the description of the `Insets` class. Let us discuss its meaning in detail in this section. We talk about insets in the context of two rectangles. An inset is the distance between the same edges (from top to top, from left to left, etc.) of two rectangles. There are four inset values – one for each side of the rectangles. An object of the `Insets` class stores the four distances. Figure 10-2 shows two rectangles and the insets of the inner rectangle relative to the outer rectangle.



**Figure 10-2.** Insets of a rectangular area relative to another rectangular area

It is possible for two rectangles to overlap instead of one to be contained fully within another. In this case, some inset values may be positive and some negative. Inset values are interpreted relative to a reference rectangle. To interpret an inset value correctly, it is required that you get the position of the reference rectangle, its edge, and the direction in which the inset needs to be measured. The context where the term “insets” is used should make these pieces of information available. In the figure, we can define the same insets relative to the inner or outer rectangle. The inset values would not change. However, the reference rectangle and the direction in which the insets are measured (to determine the sign of the inset values) will change.

Typically, in JavaFX, the term insets and the `Insets` object are used in four contexts:

- Border insets
- Background insets
- Outsets
- Insets

In the first two contexts, insets mean the distances between the edges of the layout bounds and the inner edge of the border or the inner edge of the background. In these contexts, insets are measured inwards from the edges of the layout bounds. A negative value for an inset means a distance measured outward from the edges of the layout bounds.

A border stroke or image may fall outside of the layout bounds of a `Region`. Outsets are the distances between the edges of the layout bounds of a `Region` and the outer edges of its border. Outsets are also represented as an `Insets` object.

Javadoc for JavaFX uses the term insets several times to mean the sum of the thickness of the border and the padding measured inward from all edges of the layout bounds. Be careful interpreting the meaning of the term insets when you encounter it in Javadoc.

### The HPos Enum

The `HPos` enum defines three constants: `LEFT`, `CENTER`, and `RIGHT`, to describe the horizontal positioning and alignment.

### The VPos Enum

The constants of the `VPos` enum describe vertical positioning and alignment. It has four constants: `TOP`, `CENTER`, `BASELINE`, and `BOTTOM`.

### The Pos Enum

The constants in the `Pos` enum describe vertical and horizontal positioning and alignment. It has constants for all combinations of `VPos` and `HPos` constants.

#### Constants in `Pos` enum

are `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT`, and `TOP_RIGHT`. It has two methods –

`getHpos()` and `getVpos()` – that return objects of `HPos` and `VPos` enum types, describing the horizontal and vertical positioning and alignment, respectively.

## The HorizontalDirection Enum

The `HorizontalDirection` enum has two constants, `LEFT` and `RIGHT`, which denote directions to the left and right, respectively.

## The VerticalDirection Enum

The `VerticalDirection` enum has two constants, `UP` and `DOWN`, which denote up and down directions, respectively.

## The Orientation Enum

The `Orientation` enum has two constants, `HORIZONTAL` and `VERTICAL`, which denote horizontal and vertical orientations, respectively.

## The Side Enum

The `Side` enum has four constants: `TOP`, `RIGHT`, `BOTTOM`, and `LEFT`, to denote the four sides of a rectangle.

## The Priority Enum

Sometimes, a container may have more or less space available than required to layout its children using their preferred sizes. The `Priority` enum is used to denote the priority of a node to grow or shrink when its parent has more or less space. It contains three constants: `ALWAYS`, `NEVER`, and `SOMETIMES`. A node with the `ALWAYS` priority always grows or shrinks as the available space increases or decreases. A node with `NEVER` priority never grows or shrinks as the available space increases or decreases. A node with `SOMETIMES` priority grows or shrinks when there are no other nodes with `ALWAYS` priority or nodes with `ALWAYS` priority could not consume all the increased or decreased space.

## Understanding Group

A `Group` has features of a container; for example, it has its own layout policy, coordinate system, and it is a subclass of the `Parent` class. However, its meaning is best reflected by calling it a *collection of nodes* or a *group*, rather than a *container*. It is used to manipulate a collection of nodes as a single node (or as a group). Transformations, effects, and properties applied to a `Group` are applied to all nodes in the `Group`.

A `Group` has its own layout policy, which does not provide any specific layout to its children, except giving them their preferred size:

- It renders nodes in the order they are added.
- It does not position its children. All children are positioned at  $(0, 0)$  by default. You need to write code to position child nodes of a `Group`. Use the `layoutX` and `layoutY` properties of the children nodes to position them within the `Group`.
- By default, it resizes all its children to their preferred size. The auto-sizing behavior can be disabled by setting its `autoSizeChildren` property to false. Note that if you disable the

auto-sizing property, all nodes, except shapes, will be invisible as their size will be zero, by default.

A Group does not have a size of its own. It is not resizable directly. Its size is the collective bounds of its children. Its bounds change, as the bounds of any or all of its children change. The chapter on *Understanding Nodes* explains how different types of bounds of a Group are computed.

### Creating a Group Object

You can use the no-args constructor to create an empty Group.

```
Group emptyGroup = new Group();
```

Other constructors of the Group class let you add children to the Group. One constructor takes a Collection<Node> as the initial children; another takes a var-args of the Node type.

```
Button smallBtn = new Button("Small Button");
Button bigBtn = new Button("This is a big button");

// Create a Group with two buttons using its var-args constructor
Group group1 = new Group(smallBtn, bigBtn);

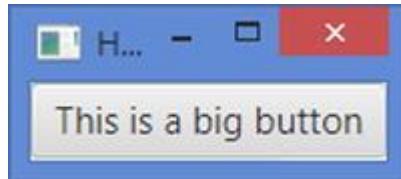
List<Node> initailList = new ArrayList<>();
initailList.add(smallBtn);
initailList.add(bigBtn);

// Create a Group with all Nodes in the initailList as its children
Group group2 = new Group(initailList);
```

### Rendering Nodes in a Group

Children of a Group are rendered in the order they are added. The following snippet of code, when displayed in a stage, looks as shown in Figure 10-3.

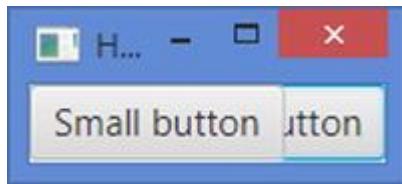
```
Button smallBtn = new Button("Small button");
Button bigBtn = new Button("This is a big button");
Group root = new Group();
root.getChildren().addAll(smallBtn, bigBtn);
Scene scene = new Scene(root);
```



**Figure 10-3.** Rendering order of the children in a Group: first smaller and second bigger

Notice that we have added two buttons to the Group. Only one of the buttons is shown. The smaller button is rendered first because it is the first one in the collection. The bigger button is rendered covering the smaller button. Both buttons exist. One is just hidden under another. If we swap the order in which buttons are added, using the following statement, the resulting screen would be as shown in Figure 10-4. Notice that the left part of the bigger button is covered by the smaller button and the right part is still showing.

```
// Add the bigger button first
root.getChildren().addAll(bigBtn, smallBtn);
```



**Figure 10-4.** Rendering order of the children in a Group: first bigger and second smaller

**Tip** If you do not want nodes in a Group to overlap, you need to set their positions.

### Positioning Nodes in a Group

You can position child nodes in a Group by assigning them absolute positions using the `layoutX` and `layoutY` properties of the nodes. Alternatively, you can use binding API to position them relative to other nodes in the Group.

Listing 10-1 shows how to use the absolute and relative positioning in a Group. Figure 10-5 shows the resulting screen. The program adds two buttons (*OK* and *Cancel*) to the Group. The *OK* button uses absolute positioning; it is placed at (10, 10). The *Cancel* button is placed relative to the *OK* button; its vertical position is the same as the *OK* button; its horizontal position is 20px after the right edge of the *OK* button. Notice the use of the *Fluent Binding API* to accomplish the relative positioning for the *Cancel* button.

### **Listing 10-1.** Laying Out Nodes in a Group

```
// NodesLayoutInGroup.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.beans.binding.NumberBinding;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class NodesLayoutInGroup extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create two buttons
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        // Set the location of the OK button
        okBtn.setLayoutX(10);
        okBtn.setLayoutY(10);

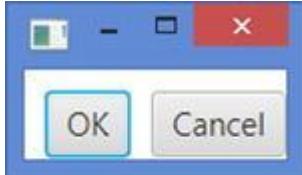
        // Set the location of the Cancel button relative to the OK
        // button
        NumberBinding layoutXBinding =
            okBtn.layoutXProperty().add(okBtn.widthProperty().add(10))
        ;
        cancelBtn.layoutXProperty().bind(layoutXBinding);
        cancelBtn.layoutYProperty().bind(okBtn.layoutYProperty());
    }
}
```

```

        Group root = new Group();
        root.getChildren().addAll(okBtn, cancelBtn);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Positioning Nodes in a Group");
        stage.show();
    }
}

```



**Figure 10-5.** A Group with two buttons using relative positions

### Applying Effects and Transformations to a Group

When you apply effects and transformations to a `Group`, they are automatically applied to all of its children. Setting a property, for example, the `disable` or `opacity` property, on a `Group`, sets the property on all of its children.

Listing 10-2 shows how to apply effects, transformations, and states to a `Group`. The program adds two buttons to the `Group`. It applies a rotation transformation of 10 degrees, a drop shadow effect, and opacity of 80%. Figure 10-6 shows that the transformation, effect, and state applied to the `Group` are applied to all of its children (two buttons in this case).

### **Listing 10-2.** Applying Effects and Transformations to a Group

```

// GroupEffect.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.effect.DropShadow;
import javafx.stage.Stage;

public class GroupEffect extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create two buttons
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        // Set the locations of the buttons
        okBtn.setLayoutX(10);
        okBtn.setLayoutY(10);
        cancelBtn.setLayoutX(80);
        cancelBtn.setLayoutY(10);

        Group root = new Group();

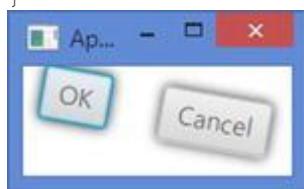
```

```

        root.setEffect(new DropShadow()); // Set a drop shadow effect
        root.setRotate(10);           // Rotate by 10 degrees
clockwise
        root.setOpacity(0.80);       // Set the opacity to 80%
        root.getChildren().addAll(okBtn, cancelBtn);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying Transformations and Effects to
a Group");
        stage.show();
    }
}

```



**Figure 10-6.** Two buttons in a Group after effects, transformations, and states are applied to the Group

## Styling a Group with CSS

The `Group` class does not offer much CSS styling. All CSS properties for the `Node` class are available for the `Group` class: for example, `-fx-cursor`, `-fx-opacity`, `-fx-rotate`, etc. A `Group` cannot have its own appearance such as padding, backgrounds, and borders.

## Understanding Region

`Region` is the base class for all layout panes. It can be styled with CSS. Unlike `Group`, it has its own size. It is resizable. It can have a visual appearance, for example, with padding, multiple backgrounds, and multiple borders. You do not use the `Region` class directly as a layout pane. If you want to roll out your own layout pane, extend the `Pane` class, which extends the `Region` class.

**Tip** The `Region` class is designed to support the CSS3 specification for backgrounds and borders, as they are applicable to JavaFX. The specification for “CSS Backgrounds and Borders Module Level 3” can be found online at <http://www.w3.org/TR/2012/CR-css3-background-20120724/>.

By default, a `Region` defines a rectangular area. However, it can be changed to any shape. The drawing area of a `Region` is divided into several parts. Depending on the property settings, a `Region` may draw outside of its layout bounds. Parts of a `Region`:

- Backgrounds (fills and images)
- Content Area
- Padding
- Borders (strokes and images)
- Margin
- Region Insets

Figure 10-7 shows parts of a Region. The margin is not directly supported as of JavaFX 2. You can get the same effect by using Insets for the border.

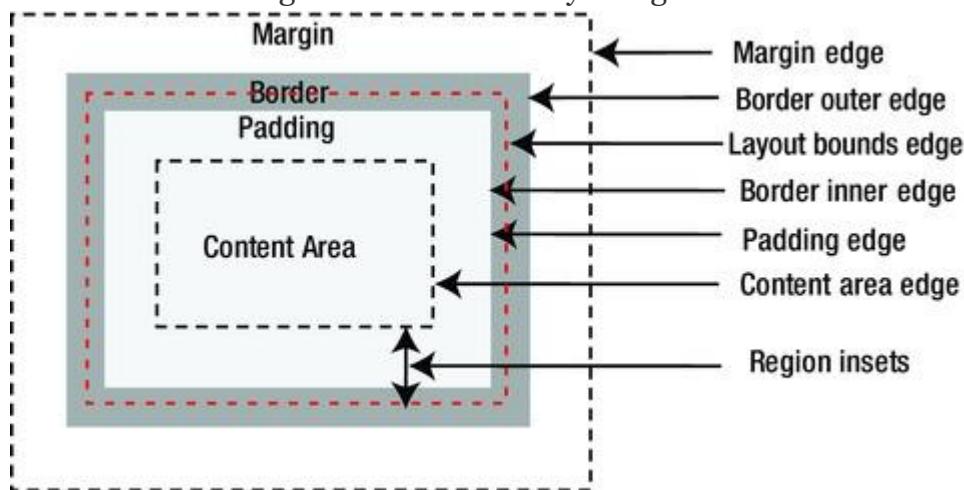


Figure 10-7. Different parts of a Region

A region may have a background that is drawn first. The content area is the area where the content of the Region (e.g., controls) are drawn.

Padding is an optional space around the content area. If the padding has a zero width, the padding edge and the content area edge are the same.

The border area is the space around the padding. If the border has a zero width, the border edge and the padding edge are the same.

Margin is the space around the border. Padding and margin are very similar. The only difference between them is that the margin defines the space around the outside edge of the border, whereas the padding defines the space around the inside edge of the border. Margins are supported for controls when they are added to panes, for example, HBox, VBox, etc. However, margins are not directly supported for a Region directly.

The content area, padding, and borders affect the layout bounds of the Region. You can draw borders outside the layout bounds of a Region, and those borders will not affect the layout bounds of the Region. Margin does not affect the layout bounds of the Region.

The distance between the edge of the layout bounds of the Region and its content area defines the insets for the Region. The Region class computes its insets automatically based on its properties. It has a read-only insets property that you can read to know its insets. Note that a layout container would need to know the area in which to place its children, and they can compute the content area knowing the layout bounds and insets.

**Tip** The background fills, background images, border strokes, border images, and content of a Region are drawn in order.

## Setting Backgrounds

A Region can have a background that consists of fills, images, or both. A fill consists of a color, radii for four corners, and insets on four sides. Fills are applied in the order they are specified. The color defines the color to be used for painting the background. The radii define the radii to be used for corners; set them to zero if you

want rectangular corners. The insets define the distance between the sides of the `Region` and the outer edges of the background fill. For example, an inset of 10px on top means that a horizontal strip of 10px inside the top edge of the layout bounds will not be painted by the background fill. An inset for the fill may be negative. A negative inset extends the painted area outside of the layout bounds of the `Region`; and in this case, the drawn area for the `Region` extends beyond its layout bounds.

The following CSS properties define the background fill for a `Region`.

- `-fx-background-color`
- `-fx-background-radius`
- `-fx-background-insets`

The following CSS properties fill the entire layout bounds of the `Region` with a red color.

```
-fx-background-color: red;  
-fx-background-insets: 0;  
-fx-background-radius: 0;
```

The following CSS properties use two fills.

```
-fx-background-color: lightgray, red;  
-fx-background-insets: 0, 4;  
-fx-background-radius: 4, 2;
```

The first fill covers the entire `Region` (see 0px insets) with a light gray color; it uses a 4px radius for all four corners, making the `Region` look like a rounded rectangle. The second fill covers the `Region` with a red color; it uses a 4px inset on all four sides, which means that 4px from the edges of the `Region` are not painted by this fill, and that area will still have the light gray color used by the first fill. A 2px radius for all four corners is used by the second fill.

Starting from JavaFX 8, you can also set the background of a `Region` in code using Java objects. An instance of the `Background` class represents the background of a `Region`. The class defines a `Background.EMPTY` constant to represent an empty background (no fills and no images).

**Tip** A `Background` object is immutable. It can be safely used as the background of multiple `Regions`.

A `Background` object has zero or more fills and images. An instance of the `BackgroundFill` class represents a fill; an instance of the `BackgroundImage` class represents an image.

The `Region` class contains a `background` property of the `ObjectProperty<Background>` type. The background of a `Region` is set using the `setBackground(Background bg)` method.

The following snippet of code creates a `Background` object with two `BackgroundFill` objects. Setting this to a `Region` produces the same effects of drawing a background with two fills as shown in the above snippet of code using the CSS style. Notice that the `Insets` and `CornerRadii` classes are used to define the insets and the radius for corners for the fills.

```
import javafx.geometry.Insets;  
import javafx.scene.layout.Background;  
import javafx.scene.layout.BackgroundFill;  
import javafx.scene.layout.CornerRadii;  
import javafx.scene.paint.Color;  
...
```

```
BackgroundFill lightGrayFill =
    new BackgroundFill(Color.LIGHTGRAY, new CornerRadii(4), new Insets(0));

BackgroundFill redFill = new BackgroundFill(Color.RED, new CornerRadii(2), new
Insets(4));

// Create a Background object with two BackgroundFill objects
Background bg = new Background(lightGrayFill, redFill);
```

The program in Listing 10-3 shows how to set the background for a `Pane`, which is a `Region`, using both the CSS properties and the `Background` object. The resulting screen is shown in Figure 10-8. The `getCSSStyledPane()` method creates a `Pane`, adds a background with two fills using CSS, and returns the `Pane`. The `getObjectStyledPane()` method creates a `Pane`, adds a background with two fills using Java classes, and returns the `Pane`. The `start()` method adds the two `Panes` to another `Pane` and positions them side-by-side.

### ***Listing 10-3.*** Using Background Fills as the Background for a Region

```
// BackgroundFillTest.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class BackgroundFillTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Pane p1 = this.getCSSStyledPane();
        Pane p2 = this.getObjectStyledPane();

        p1.setLayoutX(10);
        p1.setLayoutY(10);

        // Place p2 20px right to p1
        p2.layoutYProperty().bind(p1.layoutYProperty());
        p2.layoutXProperty().bind(p1.layoutXProperty().add(p1.widthProperty()).add(20));

        Pane root = new Pane(p1, p2);
        root.setPrefSize(240, 70);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Setting Background Fills for a Region");
        stage.show();
        stage.sizeToScene();
    }

    public Pane getCSSStyledPane() {
        Pane p = new Pane();
        p.setPrefSize(100, 50);
        p.setStyle("-fx-background-color: lightgray, red;");
    }
}
```

```

        + "-fx-background-insets: 0, 4;"  

        + "-fx-background-radius: 4, 2;");  

    return p;  

}  

public Pane getObjectStyledPane() {  

    Pane p = new Pane();  

    p.setPrefSize(100, 50);  

    BackgroundFill lightGrayFill =  

        new BackgroundFill(Color.LIGHTGRAY, new CornerRadii(4),  

new Insets(0));  

    BackgroundFill redFill =  

        new BackgroundFill(Color.RED, new CornerRadii(2), new  

Insets(4));  

    // Create a Background object with two BackgroundFill objects  

    Background bg = new Background(lightGrayFill, redFill);  

    p.setBackground(bg);  

    return p;  

}
}

```



**Figure 10-8.** Two Panes having identical backgrounds set: one using CSS and one using Java objects

The following CSS properties define the background image for a Region.

- `-fx-background-image`
- `-fx-background-repeat`
- `-fx-background-position`
- `-fx-background-size`

The `-fx-background-image` property is a CSS URL for the image. The `-fx-background-repeat` property indicates how the image will be repeated (or not repeated) to cover the drawing area of the Region. The `-fx-background-position` determines how the image is positioned with the Region. The `-fx-background-size` property determines the size of the image relative to the Region.

The following CSS properties fill the entire layout bounds of the Region with a red color.

```

-fx-background-image: URL('your_image_url_goes_here');
-fx-background-repeat: space;
-fx-background-position: center;
-fx-background-size: cover;

```

The following snippet of code and the above set of the CSS properties will produce identical effects when they are set on a Region.

```
import javafx.scene.image.Image;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundImage;
import javafx.scene.layout.BackgroundPosition;
import javafx.scene.layout.BackgroundRepeat;
import javafx.scene.layout.BackgroundSize;
...
Image image = new Image("your_image_url_goes_here");
BackgroundSize bgSize = new BackgroundSize(100, 100, true, true, false, true);
BackgroundImage bgImage = new BackgroundImage(image,
                                                BackgroundRepeat.SPACE,
                                                BackgroundRepeat.SPACE,
                                                BackgroundPosition.DEFAULT,
                                                bgSize);

// Create a Background object with an BackgroundImage object
Background bg = new Background(bgImage);
```

## Setting Padding

The padding of a Region is the space around its content area. The Region class contains a padding property of the ObjectProperty<Insets> type. You can set separate padding widths for each of the four sides.

```
// Create an HBox
HBox hb = new HBox();

// A uniform padding of 10px around all edges
hb.setPadding(new Insets(10));

// A non-uniform padding: 2px top, 4px right, 6px bottom, and 8px left
hb.setPadding(new Insets(2, 4, 6, 8));
```

## Setting Borders

A Region can have a border, which consists of strokes, images, or both. If strokes and images are not present, the border is considered empty. Strokes and images are applied in the order they are specified; all strokes are applied before images. Before JavaFX 8, you could set the border only using CSS. Starting JavaFX 8, you also set the border using the Border class in code.

**Note** We will use the phrases, “the edges of a Region” and “the layout bounds of a Region,” in this section, synonymously, which mean the edges of the rectangle defined by the layout bounds of the Region.

A stroke consists of five properties:

- A color
- A style
- A width
- Radii for four corners
- Insets on four sides

The color defines the color to be used for the stroke. You can specify four different colors for the four sides.

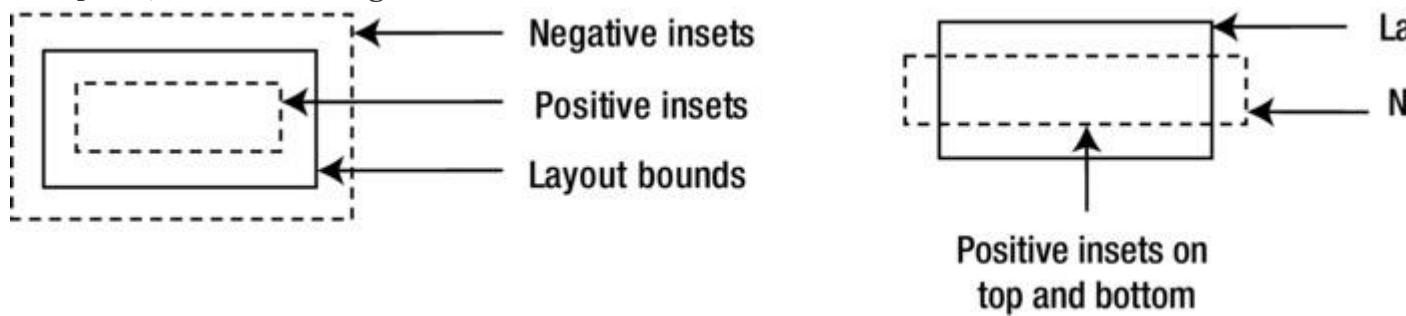
The style defines the style for the stroke: for example, solid, dashed, etc. The style also defines the location of the border relative to its insets: for

example, `inside`, `outside`, or `centered`. You can specify four different styles for the four sides.

The radii define the radii for corners; set them to zero if you want rectangular corners.

The width of the stroke defines its thickness. You can specify four different widths for the four sides.

The insets of a stroke define the distance from the sides of the layout bounds of the `Region` where the border is drawn. A positive value for the inset for a side is measured inward from the edge of the `Region`. A negative value of the inset for a side is measured outward from the edge of the `Region`. An inset of zero on a side means the edge of the layout bounds itself. It is possible to have positive insets for some sides (e.g., top and bottom) and negative insets for others (e.g., right and left). Figure 10-9 shows the positions of positive and negative insets relative to the layout bounds of a `Region`. The rectangle in solid lines is the layout bounds of a `Region`, and the rectangles in dashed lines are the insets lines.



**Figure 10-9.** Positions of positive and negative insets relative to the layout bounds

The border stroke may be drawn inside, outside, or partially inside and partially outside the layout bounds of the `Region`. To determine the exact position of a stroke relative to the layout bounds, you need to look at its two properties: `insets` and `style`.

- If the style of the stroke is `inside`, the stroke is drawn inside the insets.
- If the style is `outside`, it is drawn outside the insets.
- If the style is `centered`, it is drawn half inside and half outside the insets.

Figure 10-10 shows some examples of the border positions for a `Region`. The rectangle in dashed lines indicates the layout bounds of the `Region`. Borders are shown in a light gray color. The label below each `Region` shows the some details of the border properties (e.g., style, insets, and width).

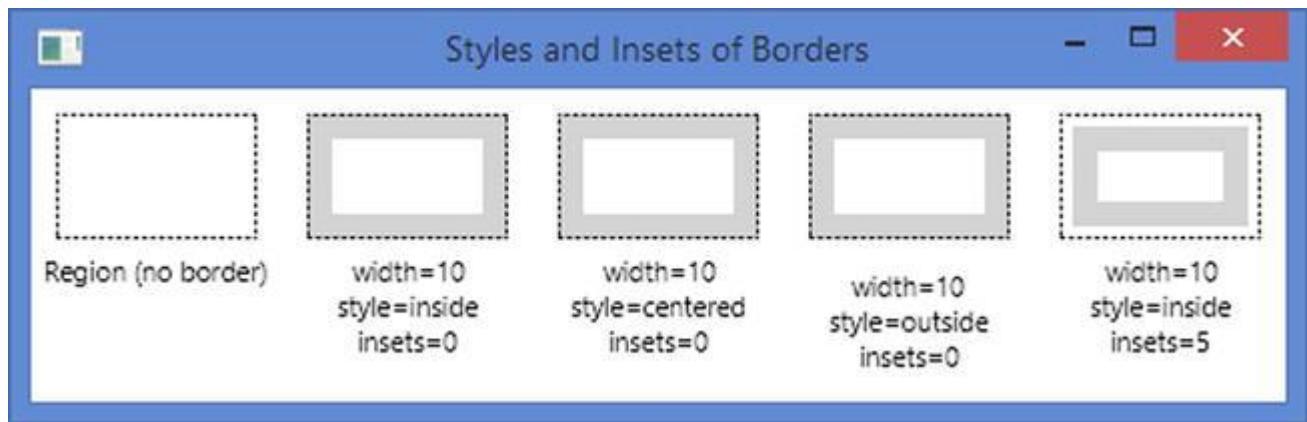


Figure 10-10. Examples of determining the position of a border based on its style and insets

The following CSS properties define border strokes for a Region.

- `-fx-border-color`
- `-fx-border-style`
- `-fx-border-width`
- `-fx-border-radius`
- `-fx-border-insets`

The following CSS properties draw a border with a stroke of 10px in width and red in color. The outside edge of the border will be the same as the edges of the Region as we have set insets and style as zero and inside, respectively. The border will be rounded on the corners as we have set the radii for all corners to 5px.

```
-fx-border-color: red;
-fx-border-style: solid inside;
-fx-border-width: 10;
-fx-border-insets: 0;
-fx-border-radius: 5;
```

The following CSS properties use two strokes for a border. The first stroke is drawn inside the edges of the Region and the second one outside.

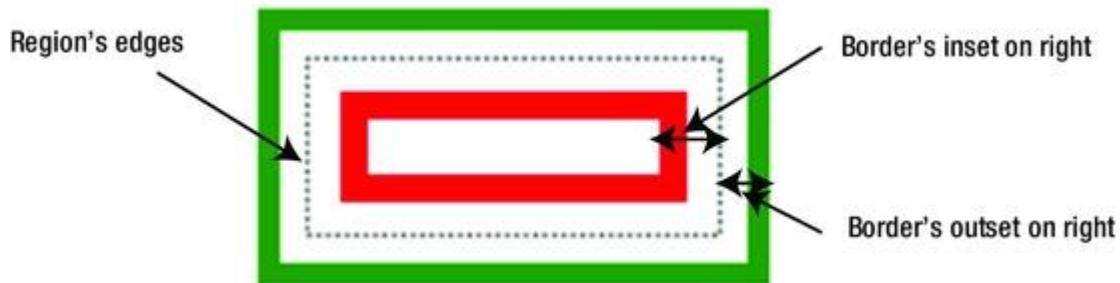
```
-fx-border-color: red, green;
-fx-border-style: solid inside, solid outside;
-fx-border-width: 5, 2 ;
-fx-border-insets: 0, 0;
-fx-border-radius: 0, 0;
```

**Tip** The part of the border drawn outside the edges of the Region does not affect its layout bounds. The part of the border drawn outside the edges of the Region is within the layout bounds of the Region. In other words, the border area that falls inside the edges of a Region influences the layout bounds for that Region.

So far, we have discussed the insets for strokes of a border. A border also has *insets* and *outsets*, which are computed automatically based on the properties for its strokes and images. The distance between the edges of the Region and the inner edges of its border, considering all strokes and images that are drawn *inside* the edges of the Region, is known as the *insets of the border*. The distance between the edges of the Region and the outer edges of its border, considering all strokes and images that are drawn *outside* the edges of the Region, is known as the *outsets of the border*. You must be able to differentiate between the insets of a stroke and insets/outsets a border. The insets of a stroke determine the location where the

stroke is drawn, whereas the insets/outsets of a border tell you how far the border extends inside/outside of the edges of the Region. Figure 10-11 shows how the insets and outsets of a border are computed. The dashed line shows the layout bounds of a Region, which has a border with two strokes: one in red and one in green. The following styles, when set on a 150px X 50px Region, results in the border as shown in Figure 10-11.

```
-fx-background-color: white;
-fx-padding: 10;
-fx-border-color: red, green, black;
-fx-border-style: solid inside, solid outside, dashed centered;
-fx-border-width: 10, 8, 1;
-fx-border-insets: 12, -10, 0;
-fx-border-radius: 0, 0, 0;
```



**Figure 10-11.** Relationship between insets/outsets of a border and the layout bounds of a Region

The insets of the border are 22px on all four sides, which is computed (10px + 12px) by adding the 10px width of the red border drawn inside 12px (insets) from the edges of the Region. The outsets of the border are 18px on all four sides, which is computed (8px + 10px) by adding the 8px width of the green border drawn outside 10px (-10 insets) from the edges of the Region.

Starting from JavaFX 8, you can also set the border of a Region in code using Java objects. An instance of the `Border` class represents the border of a Region. The class defines a `Border.EMPTY` constant to represent an empty border (no strokes and no images).

**Tip** A `Border` object is immutable. It can be safely used for multiple Regions.

A `Border` object has zero or more strokes and images. The `Border` class provides several constructors that take multiple strokes and images as arguments. The `Region` class contains a `border` property of the `ObjectProperty<Border>` type. The border of a Region is set using the `setBorder(Border b)` method.

An instance of the `BorderStroke` class represents a stroke; an instance of the `BorderImage` class represents an image. The `BorderStroke` class provides constructors to set the style of the stroke. The following are the two commonly used constructors. The third constructor allows you to set different color and style of strokes on four sides.

- `BorderStroke(Paint stroke, BorderStrokeStyle style, CornerRadii radii, BorderWidths widths)`
- `BorderStroke(Paint stroke, BorderStrokeStyle style, CornerRadii radii, BorderWidths widths, Insets insets)`

The `BorderStrokeStyle` class represents the style of a stroke. The `BorderWidths` class represents widths of a stroke on all four sides of a border. It lets you set the widths as absolute values or as a percentage of the dimensions of the Region. The following snippet of code creates a `Border` and sets it to a `Pane`.

```

BorderStrokeStyle style = new BorderStrokeStyle(StrokeType.INSIDE,
                                              StrokeLineJoin.MITER,
                                              StrokeLineCap.BUTT,
                                              10,
                                              0,
                                              null);
BorderStroke stroke = new BorderStroke(Color.GREEN,
                                         style,
                                         CornerRadii.EMPTY,
                                         new BorderWidths(8),
                                         new Insets(10));
Pane p = new Pane();
p.setPrefSize(100, 50);
Border b = new Border(stroke);
p.setBorder(b);

```

The `Border` class provides `getInsets()` and `getOutsets()` methods that return the insets and outsets for the `Border`. Both methods return an `Insets` object. Remember that the insets and outsets for a `Border` are different from insets of strokes. They are computed automatically based on the insets and styles for strokes and images that a `Border` has.

You can get all strokes and all images of a `Border` using its `getStrokes()` and `getImages()` methods, which return `List<BorderStroke>` and `List<BorderImage>`, respectively. You can compare two `Border` objects and two `BorderStroke` objects for equality using their `equals()` method.

**Listing 10-4** demonstrates how to create and set a border to a `Pane`. It displays a screen with two `Panes`. One `Pane` is styled using CSS and another using a `Border` object. The `Panes` look similar to the one shown in Figure 10-11. The program prints the insets and outsets for the borders and checks whether both borders are the same or not. Both borders use three strokes. The `getCSSStyledPane()` method returns a `Pane` styled with CSS; the `getObjectStyledPane()` method returns a `Pane` styled using a `Border` object.

#### **Listing 10-4.** Using Strokes as the Border for a Region

```

// BorderStrokeTest.java
package com.jdojo.container;

import java.util.ArrayList;
import java.util.List;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.Background;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;

```

```

import javafx.scene.shape.StrokeLineCap;
import javafx.scene.shape.StrokeLineJoin;
import javafx.scene.shape.StrokeType;
import javafx.stage.Stage;

public class BorderStrokeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Pane p1 = this.getCSSStyledPane();
        Pane p2 = this.getObjectStyledPane();

        // Place p1 and p2
        p1.setLayoutX(20);
        p1.setLayoutY(20);
        p2.layoutYProperty().bind(p1.layoutYProperty());
        p2.layoutXProperty().bind(
            p1.layoutXProperty().add(p1.widthProperty()).add(40)
        );

        Pane root = new Pane(p1, p2);
        root.setPrefSize(300, 120);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Setting Background Fills for a Region");
        stage.show();

        // Print borders details
        printBorderDetails(p1.getBorder(), p2.getBorder());
    }

    public Pane getCSSStyledPane() {
        Pane p = new Pane();
        p.setPrefSize(100, 50);
        p.setStyle("-fx-padding: 10;" +
                   "-fx-border-color: red, green, black;" +
                   "-fx-border-style: solid inside, solid outside," +
                   "dashed centered;" +
                   "-fx-border-width: 10, 8, 1;" +
                   "-fx-border-insets: 12, -10, 0;" +
                   "-fx-border-radius: 0, 0, 0;");

        return p;
    }

    public Pane getObjectStyledPane() {
        Pane p = new Pane();
        p.setPrefSize(100, 50);
        p.setBackground(Background.EMPTY);
        p.setPadding(new Insets(10));

        // Create three border strokes
        BorderStroke redStroke = new BorderStroke(Color.RED,
                                                    BorderStrokeStyle.SOLID,
                                                    CornerRadii.EMPTY,
                                                    new BorderWidths(10),
                                                    new Insets(12));

        BorderStrokeStyle greenStrokeStyle = new
        BorderStrokeStyle(StrokeType.OUTSIDE,
                         StrokeLineJoin.MITER,
                         StrokeLineCap.BUTT,
                         10,

```

```

        0,
        null);
BorderStroke greenStroke = new BorderStroke(Color.GREEN,
greenStrokeStyle,
CornerRadii.EMPTY,
new BorderWidths(8),
new Insets(-10));

List<Double> dashArray = new ArrayList<>();
dashArray.add(2.0);
dashArray.add(1.4);

BorderStrokeStyle blackStrokeStyle
= new BorderStrokeStyle(StrokeType.CENTERED,
StrokeLineJoin.MITER,
StrokeLineCap.BUTT,
10,
0,
dashArray);
BorderStroke blackStroke = new BorderStroke(Color.BLACK,
blackStrokeStyle,
CornerRadii.EMPTY,
new BorderWidths(1),
new Insets(0));

// Create a Border object with three BorderStroke objects
Border b = new Border(redStroke, greenStroke, blackStroke);
p.setBorder(b);

return p;
}

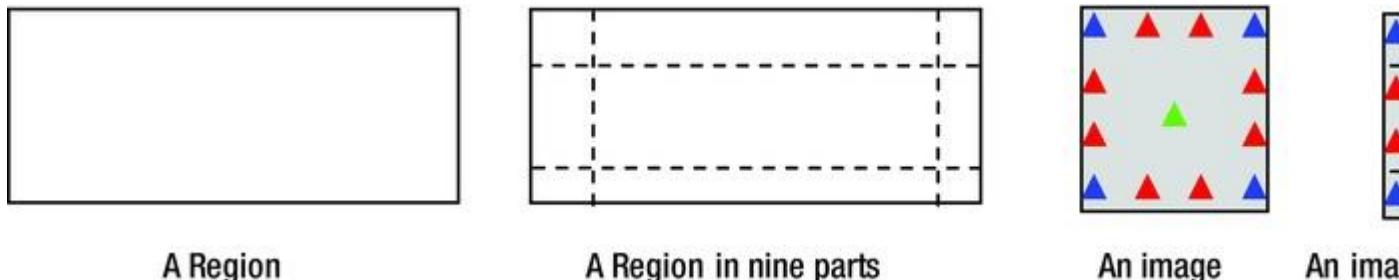
private void printBorderDetails(Border cssBorder, Border objectBorder) {
    System.out.println("cssBorder insets:" + cssBorder.getInsets());
    System.out.println("cssBorder outsets:" +
+ cssBorder.getOutsets());
    System.out.println("objectBorder insets:" +
+ objectBorder.getInsets());
    System.out.println("objectBorder outsets:" +
+ objectBorder.getOutsets());

    if (cssBorder.equals(objectBorder)) {
        System.out.println("Borders are equal.");
    } else {
        System.out.println("Borders are not equal.");
    }
}
cssBorder insets:Insets [top=22.0, right=22.0, bottom=22.0, left=22.0]
cssBorder outsets:Insets [top=18.0, right=18.0, bottom=18.0, left=18.0]
objectBorder insets:Insets [top=22.0, right=22.0, bottom=22.0, left=22.0]
objectBorder outsets:Insets [top=18.0, right=18.0, bottom=18.0, left=18.0]
Borders are equal.

```

Using an image for a border is not as straightforward as using a stroke. An image defines a rectangular area; so does a `Region`. A border is drawn around a `Region` in an area called the border image area. The border area of a `Region` may be the entire area of the `Region`; it may be partly or fully inside or outside of the `Region`. The insets on four edges of the `Region` define the border image area. To make an image a border around a `Region`, both the border image area and the image are divided into nine regions: four corners, four sides, and a middle. The border area is divided into nine parts by specifying widths on all four sides, top, right, bottom, and left. The width is the width of the border along those sides. The image is also sliced (divided)

into nine regions by specifying the slice width for each side. Figure 10-12 shows a Region, the border image area with its nine regions, an image and its nine regions (or slices). In the figure, the border image area is the same as the area of the Region.



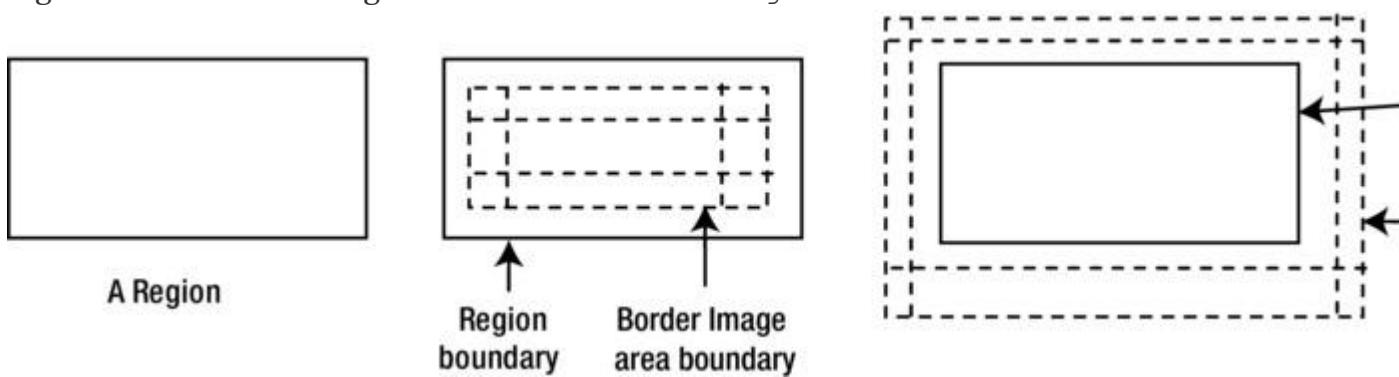
**Figure 10-12.** Slicing a Region and an image into nine parts

**Tip** The border image is not drawn if a Region uses a shape other than a rectangular shape.

Note that the four widths from the edges, while dividing a border area and an image, do not necessarily have to be uniform. For example, you can specify widths as 2px on top, 10px on right, 2px on bottom, and 10px on left.

After you have divided the border image area and the image into nine regions, you need to specify properties that control the positioning and resizing behavior of the image slices. Each of nine slices of the image has to be positioned and fit inside its corresponding part in the border image area. For example, the image slice in the upper left corner of the image has to fit in the upper-left corner part of the border image area. The two components, an image slice and its corresponding border image slice, may not be of the same size. You will need to specify how to fill the region in the border image area (scale, repeat, etc.) with the corresponding image slice. Typically, the middle slice of the image is discarded. However, if you want to fill the middle region of the border image area, you can do so with the middle slice of the image.

In Figure 10-12, the boundaries of the Region and the border image area are the same. Figure 10-13 has examples in which the boundaries of the border image area fall inside and outside of the boundary of the Region. It is possible that some regions of the border image area fall outside of the Region and some inside.



**Figure 10-13.** Relationship between the area of a Region and the border image area

The following CSS properties define border images for a Region.

- `-fx-border-image-source`
- `-fx-border-image-repeat`

- `-fx-border-image-slice`
- `-fx-border-image-width`
- `-fx-border-image-insets`

The `-fx-border-image-source` property is a CSS URL for the image. For multiple images, use a comma-separated list of CSS URLs of images.

The `-fx-border-image-repeat` property specifies how a slice of the image will cover the corresponding part of the `Region`. You can specify the property separately for the x-axis and y-axis. Valid values:

- `no-repeat`
- `repeat`
- `round`
- `space`

The `no-repeat` value specifies that the image slice should be scaled to fill the area without repeating it. The `repeat` value specifies that the image should be repeated (tiled) to fill the area. The `round` value specifies that the image should be repeated (tiled) to fill the area using a whole number of tiles, and if necessary, scale the image to use the whole number of tiles. The `space` value specifies that the image should be repeated (tiled) to fill the area using a whole number of tiles without scaling the image and by distributing the extra space uniformly around the tiles.

The `-fx-border-image-slice` property specifies inward offsets from the top, right, bottom, and left edges of the image to divide it into nine slices. The property can be specified as a number literal or a percentage of the side of the image. If the word `fill` is present in the value, the middle slice of the image is preserved and is used to fill the middle region of the border image area; otherwise, the middle slice is discarded.

The `-fx-border-image-width` property specifies the inward offsets from four sides of the border image area to divide the border image area into nine regions. Note that we divide the border image area into nine regions, not the `Region`. The property can be specified as a number literal or a percentage of the side of the border image area.

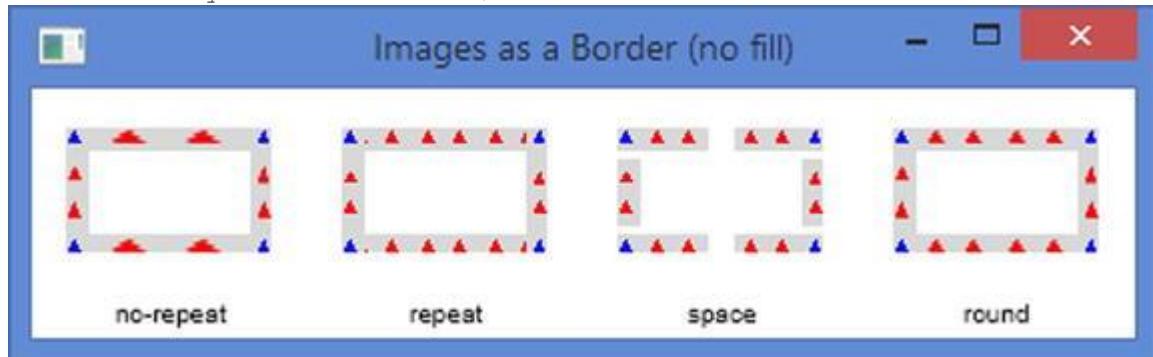
The `-fx-border-image-insets` property specifies the distance between the edges of the `Region` and the edges of the border image area on four sides. A positive inset is measured from the edge of the `Region` toward its center. A negative inset is measured outward from the edge of the `Region`. In Figure 10-13, the border image area for the `Region` in the middle has positive insets, whereas the border image area for the `Region` (third from the left) has negative insets.

Let us look at some examples of using images as a border. In all examples, we will use the image shown in Figure 10-12 as a border for a 200px X 70px `Pane`.

Listing 10-5 contains the CSS and Figure 10-14 shows the resulting `Panes` when the `-fx-border-image-repeat` property is set to `no-repeat`, `repeat`, `space`, and `round`. Notice that we have set the `-fx-border-image-width` and the `-fx-border-image-slice` properties to the same value of 9px. This will cause the corner slices to fit exactly into the corners of the border image area. The middle region of the border image area is not filled because we have not specified the `fill` value for the `-fx-border-image-slice` property. We have used a stroke to draw the boundary of the `Pane`.

***Listing 10-5.*** Using an Image as a Border Without Filling the Middle Region

```
-fx-border-image-source: url('image_url_goes_here') ;
-fx-border-image-repeat: no-repeat;
-fx-border-image-slice: 9;
-fx-border-image-width: 9;
-fx-border-image-insets: 10;
-fx-border-color: black;
-fx-border-width: 1;
-fx-border-style: dashed inside;
```

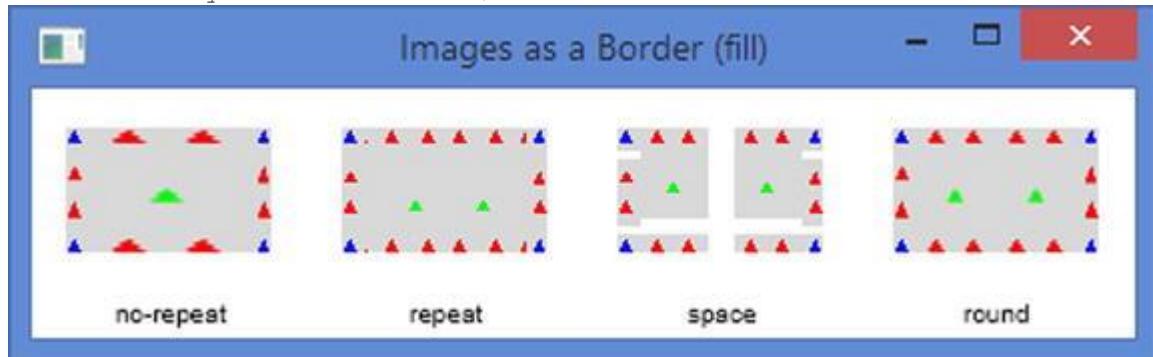


**Figure 10-14.** Using different values for `repeat` without the `fill` value for `slice` property

Listing 10-6 contains the CSS, which is a slight variation of Listing 10-5. Figure 10-15 shows the resulting Panes. This time, the middle region of the border image area is filled because we have specified the `fill` value for the `-fx-border-image-slice` property.

***Listing 10-6.*** Using an Image as a Border Filling the Middle Region

```
-fx-border-image-source: url('image_url_goes_here') ;
-fx-border-image-repeat: no-repeat;
-fx-border-image-slice: 9 fill;
-fx-border-image-width: 9;
-fx-border-image-insets: 10;
-fx-border-color: black;
-fx-border-width: 1;
-fx-border-style: dashed inside;
```



**Figure 10-15.** Using different values for `repeat` with the `fill` value for the `slice` property

The `BorderImage` class, which is immutable, represents a border image in a `Border`. All properties for the border image are specified in the constructor:

```
BorderImage(Image image,
           BorderWidths widths,
```

```

    Insets insets,
    BorderWidths slices,
    boolean filled,
    BorderRepeat repeatX,
    BorderRepeat repeatY)

```

The `BorderRepeat` enum contains `STRETCH`, `REPEAT`, `SPACE`, and `ROUND` constants that are used to indicate how the image slices are repeated in the x and y directions to fill the regions of the border image area. They have the same effect of specifying `no-repeat`, `repeat`, `space`, and `round` in CSS.

```

BorderWidths regionWidths = new BorderWidths(9);
BorderWidths sliceWidth = new BorderWidths(9);
boolean filled = false;
BorderRepeat repeatX = BorderRepeat.STRETCH;
BorderRepeat repeatY = BorderRepeat.STRETCH;
BorderImage borderImage = new BorderImage(new Image("image_url_goes_here"),
                                           regionWidths,
                                           new Insets(10),
                                           sliceWidth,
                                           filled,
                                           repeatX,
                                           repeatY);

```

**Listing 10-7** has a program that creates the border using CSS and Java classes. The resulting screen is shown in Figure 10-16. The left and right Panes are decorated with the same borders: one uses CSS and another Java classes.

### ***Listing 10-7.*** Using Strokes and Images as a Border

```

// BorderImageTest.java
package com.jdojo.container;

import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.layout.Background;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderImage;
import javafx.scene.layout.BorderRepeat;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.StrokeLineCap;
import javafx.scene.shape.StrokeLineJoin;
import javafx.scene.shape.StrokeType;
import javafx.stage.Stage;

public class BorderImageTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Get the URL of the image
        String imagePath = "resources/picture/border_with_triangles.jpg";

```

```

URL imageURL = getClass().getResource(imagePath);
String imageURLString = imageURL.toExternalForm();

Pane p1 = this.getCSSStyledPane(imageURLString);
Pane p2 = this.getObjectStyledPane(imageURLString);

// Place p1 and p2
p1.setLayoutX(20);
p1.setLayoutY(20);
p2.layoutYProperty().bind(p1.layoutYProperty());
p2.layoutXProperty().bind(p1.layoutXProperty()).add(p1.widthProperty()).add(20));

Pane root = new Pane(p1, p2);
root.setPrefSize(260, 100);

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Strokes and Images as a Border");
stage.show();
}

public Pane getCSSStyledPane(String imageURL) {
    Pane p = new Pane();
    p.setPrefSize(100, 70);
    p.setStyle("-fx-border-image-source: url('" + imageURL + "') ;" +
               "-fx-border-image-repeat: no-repeat;" +
               "-fx-border-image-slice: 9;" +
               "-fx-border-image-width: 9;" +
               "-fx-border-image-insets: 10;" +
               "-fx-border-color: black;" +
               "-fx-border-width: 1;" +
               "-fx-border-style: dashed inside;");

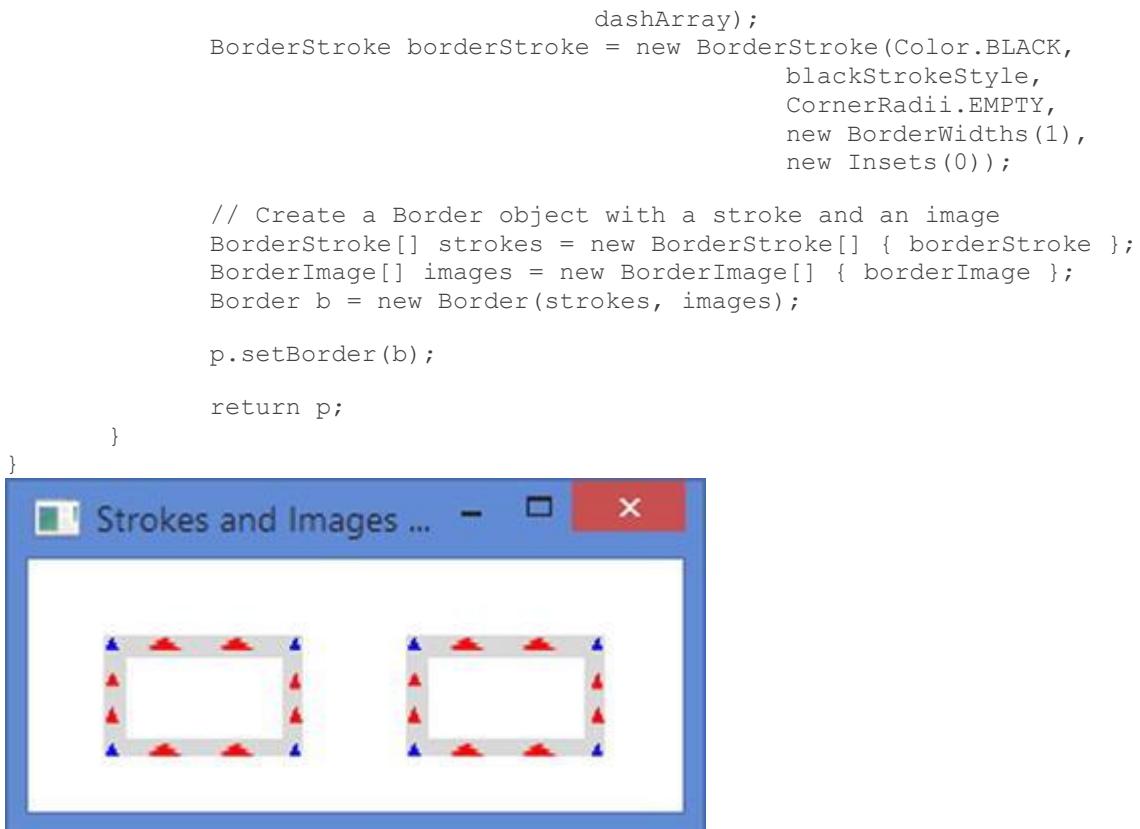
    return p;
}

public Pane getObjectStyledPane(String imageURL) {
    Pane p = new Pane();
    p.setPrefSize(100, 70);
    p.setBackground(Background.EMPTY);

    // Create a BorderImage object
    BorderWidths regionWidths = new BorderWidths(9);
    BorderWidths sliceWidth = new BorderWidths(9);
    boolean filled = false;
    BorderRepeat repeatX = BorderRepeat.STRETCH;
    BorderRepeat repeatY = BorderRepeat.STRETCH;
    BorderImage borderImage = new BorderImage(new Image(imageURL),
                                              regionWidths,
                                              new Insets(10),
                                              sliceWidth,
                                              filled,
                                              repeatX,
                                              repeatY);

    // Set the Pane's boundary with a dashed stroke
    List<Double> dashArray = new ArrayList<>();
    dashArray.add(2.0);
    dashArray.add(1.4);
    BorderStrokeStyle blackBorderStyle =
        new BorderStrokeStyle(StrokeType.INSIDE,
                             StrokeLineJoin.MITER,
                             StrokeLineCap.BUTT,
                             10,
                             0,
                             0,
                             0);
}

```



**Figure 10-16.** Creating a border with a strike and an image using CSS and Java classes

### Setting Margins

Setting margins on a `Region` is not supported directly. Most layout panes support margins for their children. If you want margins for a `Region`, add it to a layout pane, for example, an `HBox`, and use the layout pane instead of the `Region`.

```

Pane p1 = new Pane();
p1.setPrefSize(100, 20);

HBox box = new HBox();

// Set a margin of 10px around all four sides of the Pane
HBox.setMargin(p1, new Insets(10));
box.getChildren().addAll(p1);

```

Now, use `box` instead of `p1` to get the margins around `p1`.

### Understanding Panes

`Pane` is a subclass class of the `Region` class. It exposes the `getChildren()` method of the `Parent` class, which is the superclass of the `Region` class. This means that instances of the `Pane` class and its subclasses can add any children.

A `Pane` provides the following layout features:

- It can be used when absolute positioning is needed. By default, it positions all its children at (0, 0). You need to set the positions of the children explicitly.
- It resizes all resizable children to their preferred sizes.

By default, a `Pane` has minimum, preferred, and maximum sizes. Its minimum width is the sum of the left and right insets; its minimum height is the sum of the top and bottom insets. Its preferred width is the width required to display all its children at their current x location with their preferred widths; its preferred height is the height required to display all its children at their current y location with their preferred heights. Its maximum width and height are set to `Double.MAX_VALUE`.

The program in Listing 10-8 shows how to create a `Pane`, add two `Buttons` to it, and how to position the `Buttons`. The resulting screen is shown in Figure 10-17. The `Pane` uses a border to show the area it occupies in the screen. Try resizing the window, and you will find that the `Pane` shrinks and expands.

### ***Listing 10-8.*** Using Panes

```
// PaneTest.java
package com.jdojo.container;

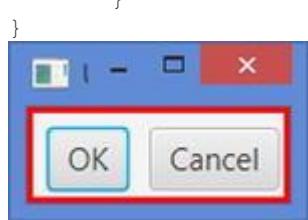
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class PaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");
        okBtn.relocate(10, 10);
        cancelBtn.relocate(60, 10);

        Pane root = new Pane();
        root.getChildren().addAll(okBtn, cancelBtn);
        root.setStyle("-fx-border-style: solid inside;" +
                     "-fx-border-width: 3;" +
                     "-fx-border-color: red;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Panes");
        stage.show();
    }
}
```



**Figure 10-17.** A pane with two Buttons

A `Pane` lets you set its preferred size:

```
Pane root = new Pane();
root.setPrefSize(300, 200); // 300px wide and 200px tall
```

You can tell the Pane to compute its preferred size based on its children sizes by resetting its preferred width and height to the computed width and height.

```
Pane root = new Pane();

// Set the preferred size to 300px wide and 200px tall
root.setPrefSize(300, 200);

/* Do some processing... */

// Set the default preferred size
root.setPrefSize(Region.USE_COMPUTED_SIZE, Region.USE_COMPUTED_SIZE);
```

**Tip** A Pane does not clip its content; its children may be displayed outside its bounds.

## Understanding HBox

An HBox lays out its children in a single horizontal row. It lets you set the horizontal spacing between adjacent children, margins for any children, resizing behavior of children, etc. It uses 0px as the default spacing between adjacent children. The default width of the content area and HBox is wide enough to display all its children at their preferred widths, and the default height is the largest of the heights of all its children.

You cannot set the locations for children in an HBox. They are automatically computed by the HBox. You can control the locations of children to some extent by customizing the properties of the HBox and setting constraints on the children.

### Creating HBox Objects

Constructors of the HBox class let you create HBox objects with or without specifying the spacing and initial set of children.

```
// Create an empty HBox with the default spacing (0px)
HBox hbox1 = new HBox();

// Create an empty HBox with a 10px spacing
HBox hbox2 = new HBox(10);

// Create an HBox with two Buttons and a 10px spacing
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
HBox hbox3 = new HBox(10, okBtn, cancelBtn);
```

The program in Listing 10-9 shows how to use an HBox. It adds a Label, a TextField, and two Buttons to an HBox. Spacing between adjacent children is set to 10px. A padding of 10px is used to maintain a distance between the edges of the HBox and the edges of its children. The resulting window is shown in Figure 10-18.

### **Listing 10-9.** Using the HBox Layout Pane

```
// HBoxTest.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
```

```

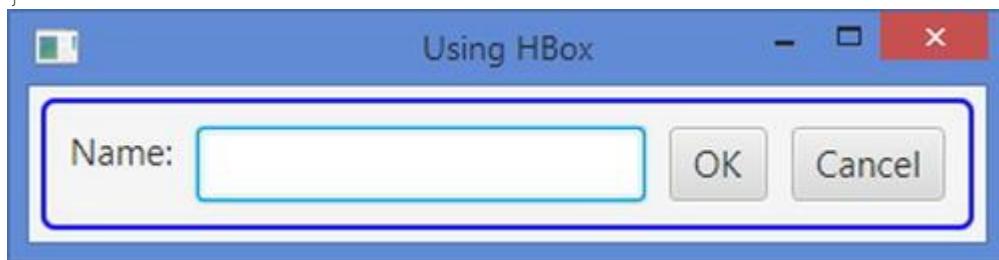
public class HBoxTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Name:");
        TextField nameFld = new TextField();
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        HBox root = new HBox(10); // 10px spacing
        root.getChildren().addAll(nameLbl, nameFld, okBtn, cancelBtn);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

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

```



**Figure 10-18.** An HBox with a Label, a TextField, and two Buttons

### HBox Properties

The `HBox` class declares three properties as listed in Table 10-2.

**Table 10-2.** Properties Declared in the `HBox` Class

Property	Type	Description
<code>alignment</code>	<code>ObjectProperty&lt;Pos&gt;</code>	It specifies the alignment of children relative to the content of the <code>HBox</code> . The <code>fillHeight</code> property is ignored if the vertical alignment is set to <code>BASELINE</code> . The default value is <code>Pos.TOP_LEFT</code> .
<code>fillHeight</code>	<code>BooleanProperty</code>	It specifies whether the resizable children are resized to fill the height of the <code>HBox</code> or they are given their preferred heights. This property is ignored, if the vertical alignment is set to <code>BASELINE</code> . The default value is <code>false</code> .

Property	Type	Description
spacing	DoubleProperty	It specifies the horizontal spacing between adjacent children. The value is zero.

## The Alignment Property

Using the alignment property is simple. It specifies how children are aligned within the content area of the `HBox`. By default, an `HBox` allocates just enough space for its content to lay out all children at their preferred size. The effect of the alignment property is noticeable when the `HBox` grows bigger than its preferred size.

The program in Listing 10-10 uses an `HBox` with two Buttons. It sets the alignment of the `HBox` to `Pos.BOTTOM_RIGHT`. It sets the preferred size of the `HBox` a little bigger than needed to accommodate all its children, so you can see the effect of the alignment. The resulting window is shown in Figure 10-19. When you resize the window, the children stay aligned in the bottom-right area.

### ***Listing 10-10.*** Using HBox Alignment Property

```
// HBoxAlignment.java
package com.jdojo.container;

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

public class HBoxAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

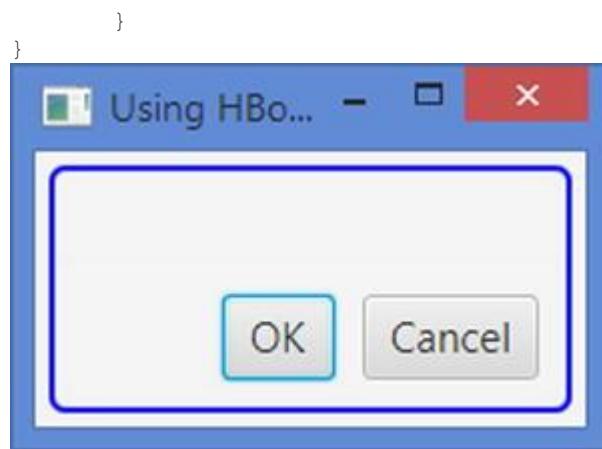
    @Override
    public void start(Stage stage) {
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        HBox hbox = new HBox(10);
        hbox.setPrefSize(200, 100);
        hbox.getChildren().addAll(okBtn, cancelBtn);

        // Set the alignment to bottom right
        hbox.setAlignment(Pos.BOTTOM_RIGHT);

        hbox.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(hbox);
        stage.setScene(scene);
        stage.setTitle("Using HBox Alignment Property");
        stage.show();
    }
}
```



**Figure 10-19.** An HBox with two Buttons and alignment property set to Pos.BOTTOM\_RIGHT

### The fillHeight Property

The `fillHeight` property specifies whether the `HBox` expands its children vertically to fill the height of its content area or keeps them to their preferred height. Note that this property affects only those child nodes that allow for the vertical expansion. For example, by default, the maximum height of a `Button` is set to its preferred height, and a `Button` does not become taller than its preferred width in an `HBox`, even if vertical space is available. If you want a `Button` to expand vertically, set its maximum height to `Double.MAX_VALUE`. By default, a `TextArea` is set to expand. Therefore, a `TextArea` inside an `HBox` will become taller as the height of the `HBox` is increased. If you do not want the resizable children to fill the height of the content area of an `HBox`, set the `fillHeight` property to false.

**Tip** The preferred height of the content area of an `HBox` is the largest of the preferred height of its children. Resizable children fill the full height of the content area, provided their maximum height property allows them to expand. Otherwise, they are kept at their preferred height.

The program in Listing 10-11 shows how the `fillHeight` property affects the height of the children of an `HBox`. It displays some controls inside an `HBox`. A `TextArea` can grow vertically by default. The maximum height of the `Cancel` button is set to `Double.MAX_VALUE`, so it can grow vertically. A `CheckBox` is provided to change the value of the `fillHeight` property of the `HBox`. The initial window is shown in Figure 10-20. Notice that the `OK` button has the preferred height, whereas the `Cancel` button expands vertically to fill the height of the content area as determined by the `TextArea`. Resize the window to make it taller and change the `fillHeight` property using the `CheckBox`; the `TextArea` and the `Cancel` button expands and shrinks vertically.

### **Listing 10-11.** Using the fillHeight Property of an HBox

```
// HBoxFillHeight.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
```

```

import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class HBoxFillHeight extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        HBox root = new HBox(10); // 10px spacing

        Label descLbl = new Label("Description:");
        TextArea desc = new TextArea();
        desc.setPrefColumnCount(10);
        desc.setPrefRowCount(3);

        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

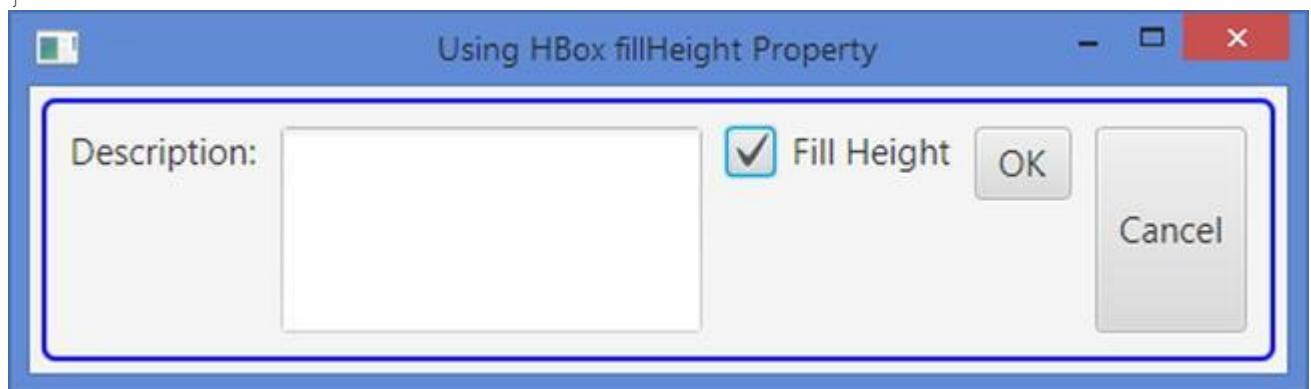
        // Let the Cancel button expand vertically
        cancelBtn.setMaxHeight(Double.MAX_VALUE);

        CheckBox fillHeightCbx = new CheckBox("Fill Height");
        fillHeightCbx.setSelected(true);

        // Add an event handler to the CheckBox, so the user can set the
        // fillHeight property using the CheckBox
        fillHeightCbx.setOnAction(e ->
            root.setFillHeight(fillHeightCbx.isSelected()));

        root.getChildren().addAll(
            descLbl, desc, fillHeightCbx, okBtn,
            cancelBtn);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");
    }
}

```



**Figure 10-20.** An HBox with some control, where the user can change the fillHeight property

## The Spacing Property

The spacing property specifies the horizontal distance between adjacent children in an `HBox`. By default, it is set to `opx`. It can be set in constructors or using the `setSpacing()` method.

### Setting Constraints for Children in HBox

`HBox` supports two types of constraints, *hgrow* and *margin*, which can be set on each child node individually. The `hgrowconstraint` specifies whether a child node expands horizontally when additional space is available. The margin constraint specifies space outside the edges of a child node. The `HBox` class provides `setHgrow()` and `setMargin()` static methods to specify these constraints. You can use `null` with these methods to remove the constraints individually. Use the `clearConstraints(Node child)` method to remove both constraints for a child node at once.

### Letting Children Grow Horizontally

By default, the children in an `HBox` get their preferred widths. If the `HBox` is expanded horizontally, its children may get the additional available space, provided their `hgrow` priority is set to grow. If an `HBox` is expanded horizontally and none of its children has its `hgrow` constraint set, the additional space is left unused.

The `hgrow` priority for a child node is set using the `setHgrow()` static method of the `HBox` class by specifying the child node and the priority.

```
// Let the TextField always grow horizontally
root.setHgrow(nameFld, Priority.ALWAYS);
```

To reset the `hgrow` priority of a child node, use `null` as the priority.

```
// Stop the TextField from growing horizontally
root.setHgrow(nameFld, null);
```

The program in Listing 10-12 shows how to set the priority of a `TextField` to `Priority.ALWAYS`, so it can take all the additional horizontal space when the `HBox` is expanded. Figure 10-21 shows the initial and expanded windows. Notice that all controls, except the `TextField`, stayed at their preferred widths, after the window is expanded horizontally.

#### **Listing 10-12.** Letting a TextField Grow Horizontally

```
// HBoxHGrow.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class HBoxHGrow extends Application {
```

```

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

@Override
public void start(Stage stage) {
    Label nameLbl = new Label("Name:");
    TextField nameFld = new TextField();

    Button okBtn = new Button("OK");
    Button cancelBtn = new Button("Cancel");

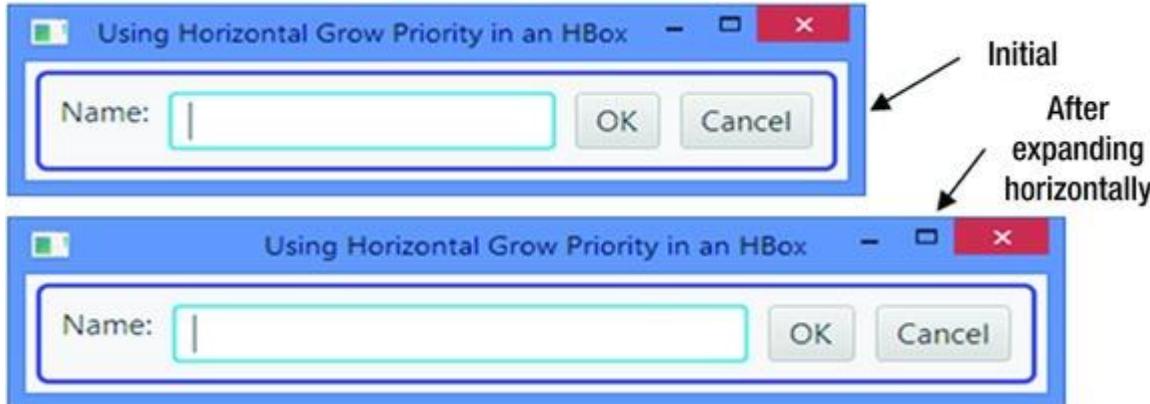
    HBox root = new HBox(10);
    root.getChildren().addAll(nameLbl, nameFld, okBtn, cancelBtn);

    // Let the TextField always grow horizontally
    HBox.setHgrow(nameFld, Priority.ALWAYS);

    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using Horizontal Grow Priority in an HBox");
    stage.show();
}
}

```



**Figure 10-21.** An HBox with a TextField set to always grow horizontally

## Setting Margins for Children

Margins are extra spaces added outside the edges of a node. The following snippet of code shows how to add margins to the children of an HBox.

```

Label nameLbl = new Label("Name:");
TextField nameFld = new TextField();
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");

HBox hbox = new HBox(nameLbl, nameFld, okBtn, cancelBtn);

// Set a margin for all children:
// 10px top, 2px right, 10px bottom, and 2px left
Insets margin = new Insets(10, 2, 10, 2);
HBox.setMargin(nameLbl, margin);
HBox.setMargin(nameFld, margin);

```

```
HBox.setMargin(okBtn, margin);
HBox.setMargin(cancelBtn, margin);
```

You can remove the margin from a child node by setting the margin value to null.

```
// Remove margins for okBtn
HBox.setMargin(okBtn, null);
```

**Tip** Be careful when using the spacing property of the HBox and the margin constraint on its children. Both will add to the horizontal gap between adjacent children. If you want margins applied, keep the horizontal spacing between children uniform, and set the right and left margins for children to zero.

## Understanding VBox

A VBox lays out its children in a single vertical column. It lets you set the vertical spacing between adjacent children, margins for any children, resizing behavior of children, etc. It uses 0px as the default spacing between adjacent children. The default height of the content area of a VBox is tall enough to display all its children at their preferred heights, and the default width is the largest of the widths of all its children.

You cannot set the locations for children in a VBox. They are automatically computed by the VBox. You can control the locations of children to some extent by customizing the properties of the VBox and setting constraints on the children.

Working with a VBox is similar to working with an HBox with a difference that they work in opposite directions. For example, in an HBox, the children fills the height of the content area by default, and in a VBox, children fill the width of the content by default; an HBox lets you set hgrow constraints on a child node and a VBox lets you set the vgrowconstraint.

## Creating VBox Objects

Constructors of the VBox class let you create VBox objects with or without specifying the spacing and initial set of children.

```
// Create an empty VBox with the default spacing (0px)
VBox vbox1 = new VBox();

// Create an empty VBox with a 10px spacing
VBox vbox2 = new VBox(10);

// Create a VBox with two Buttons and a 10px spacing
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
VBox vbox3 = new VBox(10, okBtn, cancelBtn);
```

The program in Listing 10-13 shows how to use a VBox. It adds a Label, a TextField, and two Buttons to a VBox. Spacing between adjacent children is set to 10px. A padding of 10px is used to maintain a distance between the edges of the VBox and the edges of its children. The resulting window is shown in Figure 10-22.

### **Listing 10-13.** Using the VBox Layout Pane

```
// VBoxTest.java
package com.jdojo.container;

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

```

import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

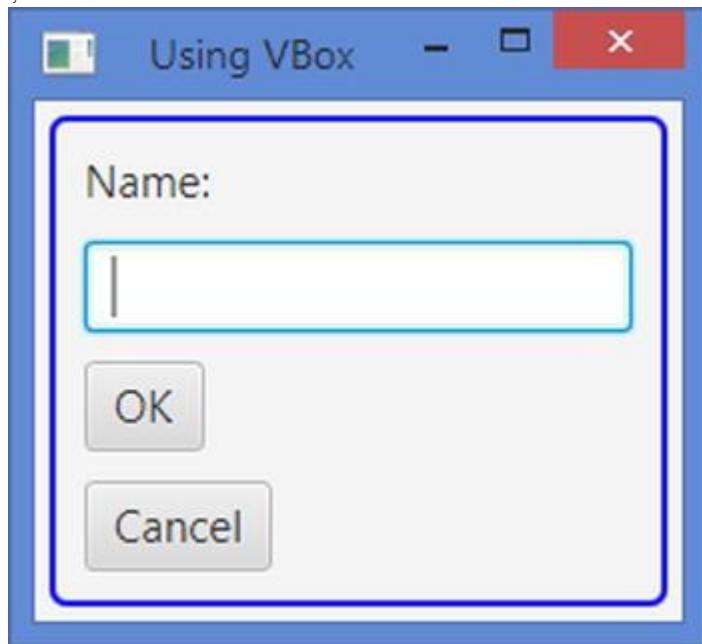
public class VBoxTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Name:");
        TextField nameFld = new TextField();
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        VBox root = new VBox(10); // 10px spacing
        root.getChildren().addAll(nameLbl, nameFld, okBtn, cancelBtn);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

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

```



**Figure 10-22.** A `VBox` with a `Label`, a `TextField`, and two `Buttons`

### VBox Properties

The `VBox` class declares three properties as listed in Table 10-3.

**Table 10-3.** Properties Declared in the `VBox` Class

Property	Type	Description
alignment	ObjectProperty<Pos>	It specifies the alignment of children relative to the content area of the VBox. The default value is Pos.TOP_LEFT.
fillWidth	BooleanProperty	It specifies whether the resizable children are resized to fill the VBox or they are given their preferred widths. The default value is false.
spacing	DoubleProperty	It specifies the vertical spacing between adjacent children. The default value is zero.

## The Alignment Property

Using the `alignment` property is simple. It specifies how children are aligned within the content area of the `VBox`. By default, a `VBox` allocates just enough space for its content to lay out all children at their preferred size. The effect of the `alignment` property is noticeable when the `VBox` grows bigger than its preferred size.

The program in Listing 10-14 uses a `VBox` with two `Buttons`. It sets the alignment of the `VBox` to `Pos.BOTTOM_RIGHT`. It sets the preferred size of the `VBox` a little bigger than needed to accommodate all its children, so you can see the effect of the alignment. The resulting window is shown in Figure 10-23. When you resize the window, the children stay aligned in the bottom-right area.

### ***Listing 10-14.*** Using VBox Alignment Property

```
// VBoxAlignment.java
package com.jdojo.container;

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

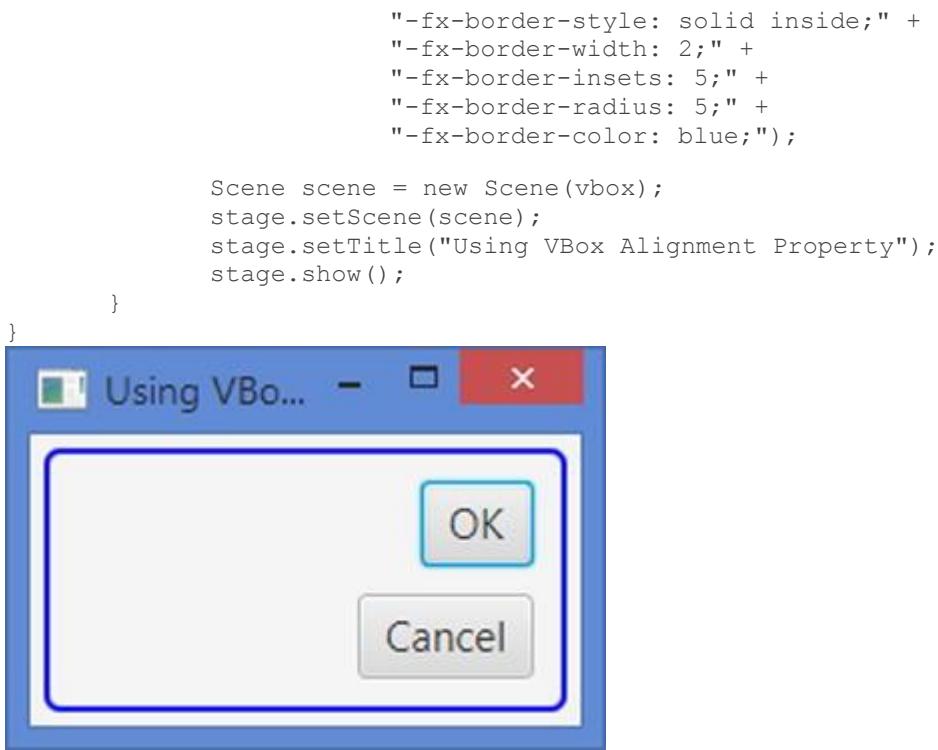
public class VBoxAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        VBox vbox = new VBox(10);
        vbox.setPrefSize(200, 100);
        vbox.getChildren().addAll(okBtn, cancelBtn);

        // Set the alignment to bottom right
        vbox.setAlignment(Pos.BOTTOM_RIGHT);

        vbox.setStyle("-fx-padding: 10;" +
                     "-fx-border-width: 1px;" +
                     "-fx-border-color: black;" +
                     "-fx-background-color: white");
    }
}
```



**Figure 10-23.** A VBox with two Buttons and alignment property set to Pos.BOTTOM\_RIGHT

### The fillWidth Property

The `fillWidth` property specifies whether the `VBox` expands its children horizontally to fill the width of its content area or keeps them to their preferred height. Note that this property affects only those child nodes that allow for the horizontal expansion. For example, by default, the maximum width of a `Button` is set to its preferred width, and a `Button` does not become wider than its preferred width in a `VBox`, even if horizontal space is available. If you want a `Button` to expand horizontally, set its maximum width to `Double.MAX_VALUE`. By default, a `TextField` is set to expand. Therefore, a `TextField` inside a `VBox` will become wider as the width of the `VBox` is increased. If you do not want the resizable children to fill the width of the content area of a `VBox`, set the `fillWidth` property to false. Run the program in Listing 10-13 and try expanding the window horizontally. The `TextField` will expand horizontally as the window expands.

**Tip** The preferred width of the content area of a `VBox` is the largest of the preferred width of its children. Resizable children fill the full width of the content area, provided their maximum width property allows them to expand. Otherwise, they are kept at their preferred width.

It is often needed in a GUI application that you need to arrange a set of `Buttons` in a vertical column and make them the same size. You need to add the buttons to a `VBox` and set the maximum width of all buttons to `Double.MAX_VALUE` so they can grow to match the width of the widest button in the group. The program in Listing 10-15 shows how to achieve this. Figure 10-24 shows the window.

**Listing 10-15.** Using the `fillWidth` Property of a `VBox`

```
// VBoxFillWidth.java
package com.jdojo.container;

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

public class VBoxFillWidth extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button b1 = new Button("New");
        Button b2 = new Button("New Modified");
        Button b3 = new Button("Not Modified");
        Button b4 = new Button("Data Modified");

        // Set the max width of the buttons to Double.MAX_VALUE,
        // so they can grow horizontally
        b1.setMaxWidth(Double.MAX_VALUE);
        b2.setMaxWidth(Double.MAX_VALUE);
        b3.setMaxWidth(Double.MAX_VALUE);
        b4.setMaxWidth(Double.MAX_VALUE);

        VBox root = new VBox(10, b1, b2, b3, b4);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using VBox fillWidth Property");
        stage.show();
    }
}
```

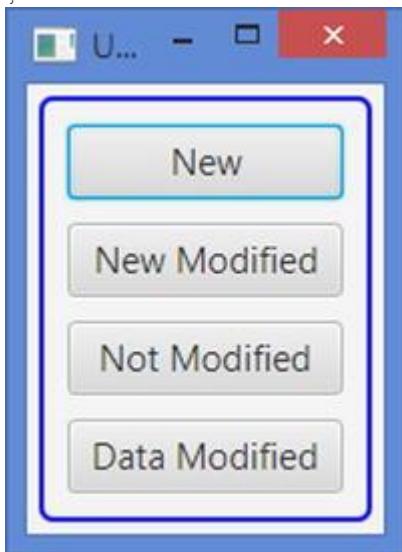


Figure 10-24. A VBox with some control, where the user can change the fillWidth property

When you expand the `VBox` horizontally in Listing 10-16, all buttons grow to fill the available extra space. To prevent the buttons growing when the `VBox` expands in the horizontal direction, you can add the `VBox` in an `HBox` and add the `HBox` to the scene.

**Tip** You can create powerful visual effects by nesting `HBox` and `VBox` layout panes. You can also add buttons (or any other types of nodes) in a column in a `GridPane` to make them the same size. Please refer to the *Understanding GridPane* section for more details.

## The Spacing Property

The spacing property specifies the vertical distance between adjacent children in a `VBox`. By default, it is set to `opx`. It can be set in the constructors or using the `setSpacing()` method.

### Setting Constraints for Children in `VBox`

`VBox` supports two types of constraints, *vgrow* and *margin*, that can be set on each child node individually. The *vgrow* constraint specifies whether a child node expands vertically when additional space is available. The margin constraint specifies space outside the edges of a child node. The `VBox` class provides `setVgrow()` and `setMargin()` static methods to specify these constraints. You can use `null` with these methods to remove the constraints individually. Use the `clearConstraints(Node child)` method to remove both constraints for a child node at once.

## Letting Children Grow Vertically

By default, the children in a `VBox` get their preferred heights. If the `VBox` is expanded vertically, its children may get the additional available space, provided their *vgrow* priority is set to grow. If a `VBox` is expanded vertically and none of its children has its *vgrow* constraint set, the additional space is left unused.

The *vgrow* priority for a child node is set using the `setVgrow()` static method of the `VBox` class by specifying the child node and the priority.

```
// Create a root VBox
VBox root = new VBox(10);
TextArea desc = new TextArea();

// Let the TextArea always grow vertically
root.setVgrow(desc, Priority.ALWAYS);
```

To reset the *vgrow* priority of a child node, use `null` as the priority.

```
// Stop the TextArea from growing horizontally
root.setVgrow(desc, null);
```

The program in Listing 10-16 shows how to set the priority of a `TextArea` to `Priority.ALWAYS`, so it can take all the additional vertical space when the `VBox` is expanded. Figure 10-25 shows the initial and expanded windows. Notice that the `Label` stays at its preferred height, after the window is expanded vertically.

### **Listing 10-16.** Letting a `TextArea` Grow Vertically

```
// VBoxVGrow.java
package com.jdojo.container;
```

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class VBoxVGrow extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label descLbl = new Label("Description:");
        TextArea desc = new TextArea();
        desc.setPrefColumnCount(10);
        desc.setPrefRowCount(3);

        VBox root = new VBox(10);
        root.getChildren().addAll(descLbl, desc);

        // Let the TextArea always grow vertically
        VBox.setVgrow(desc, Priority.ALWAYS);

        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Vertical Grow Priority in a VBox");
        stage.show();
    }
}
```

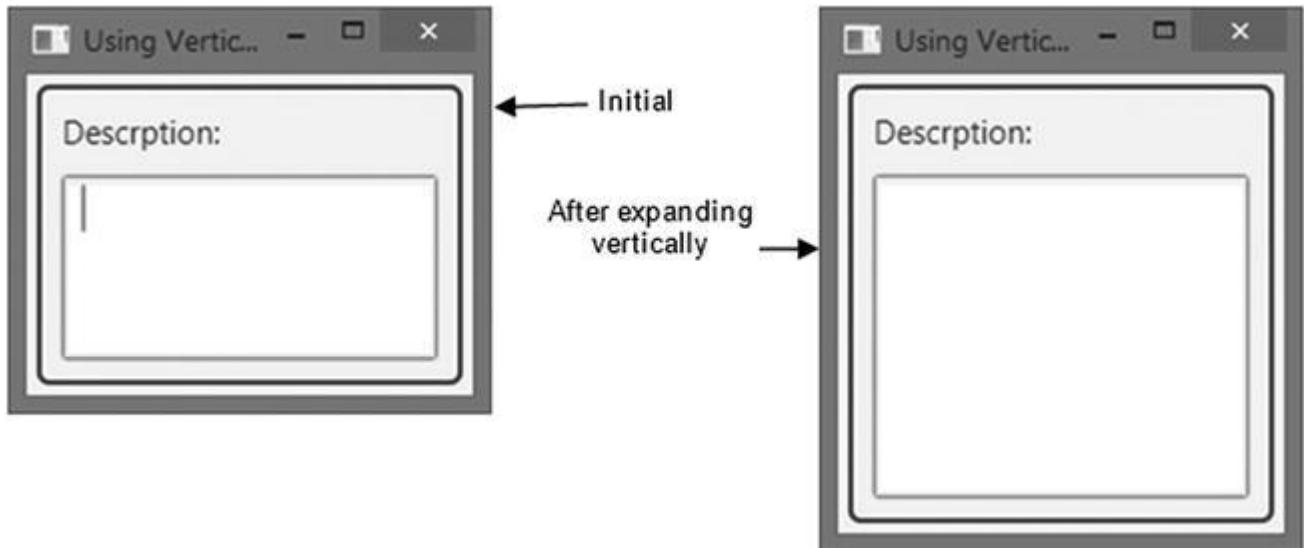


Figure 10-25. A VBox with a TextArea set to always grow vertically

## Setting Margin for Children

You can set margins for the children of a `VBox` using its `setMargin()` static method.

```
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
VBox vbox = new VBox(okBtn, cancelBtn);

// Set margins for OK and cancel buttons
Insets margin = new Insets(5);
VBox.setMargin(okBtn, margin);
VBox.setMargin(cancelBtn, margin);
...
// Remove margins for okBtn
VBox.setMargin(okBtn, null);
```

## Understanding FlowPane

A `FlowPane` is a simple layout pane that lays out its children in rows or columns wrapping at a specified width or height. It lets its children flow horizontally or vertically, and hence the name “flow pane.” You can specify a preferred wrap length, which is the preferred width for a horizontal flow and the preferred height for a vertical flow, where the content is wrapped. A `FlowPane` is used in situations where the relative locations of children are not important: for example, displaying a series of pictures or buttons. A `FlowPane` gives all its children their preferred sizes. Rows and columns may be of different heights and widths. You can customize the vertical alignments of children in rows and the horizontal alignments of children in columns.

**Tip** Children in a horizontal `FlowPane` may be arranged in rows from left to right or right to left, which is controlled by the `nodeOrientation` property declared in the `Node` class. The default value for this property is set to `NodeOrientation.LEFT_TO_RIGHT`. If you want the children to flow right to left, set the property to `NodeOrientation.RIGHT_TO_LEFT`. This applies to all layout panes that arrange children in rows (e.g., `HBox`, `TilePane`, etc.).

The orientation of a `FlowPane`, which can be set to horizontal or vertical, determines the direction of the flow for its content. In a horizontal `FlowPane`, the content flows in rows. In a vertical `FlowPane`, the content flows in columns. Figure 10-26 and Figure 10-27 show a `FlowPane` with ten buttons. The buttons are added in the order they have been labeled. That is, `Button 1` is added before `Button 2`. The `FlowPane` in Figure 10-26 has a horizontal orientation, whereas the `FlowPane` in Figure 10-27 has a vertical orientation. By default, a `FlowPane` has a horizontal orientation.

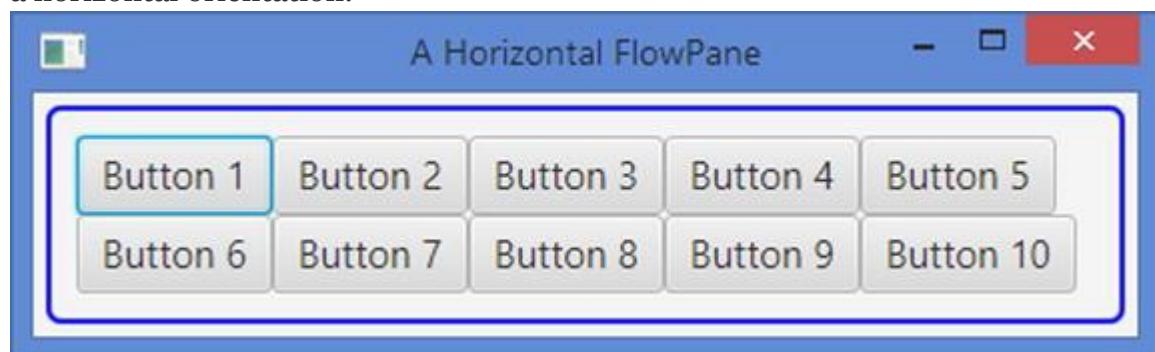
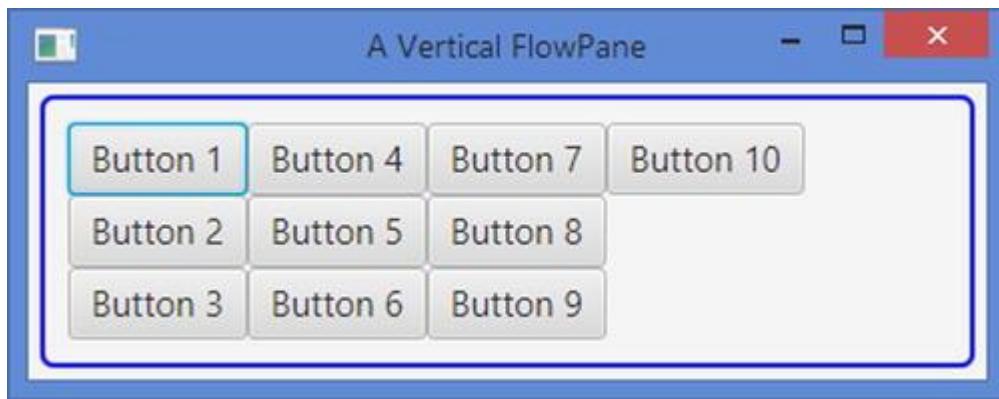


Figure 10-26. A horizontal flow pane showing ten buttons



**Figure 10-27.** A vertical flow pane showing ten buttons

### Creating FlowPane Objects

The `FlowPane` class provides several constructors to create `FlowPane` objects with a specified orientation (horizontal or vertical), a specified horizontal and vertical spacing between children, and a specified initial list of children.

```
// Create an empty horizontal FlowPane with 0px spacing
FlowPane fpane1 = new FlowPane();

// Create an empty vertical FlowPane with 0px spacing
FlowPane fpane2 = new FlowPane(Orientation.VERTICAL);

// Create an empty horizontal FlowPane with 5px horizontal and 10px vertical
// spacing
FlowPane fpane3 = new FlowPane(5, 10);

// Create an empty vertical FlowPane with 5px horizontal and 10px vertical
// spacing
FlowPane fpane4 = new FlowPane(Orientation.VERTICAL, 5, 10);

// Create a horizontal FlowPane with two Buttons and 0px spacing
FlowPane fpane5 = new FlowPane(new Button("Button 1"), new Button("Button 2"));
```

The program in Listing 10-17 shows how to create a `FlowPane` and add children. It adds ten Buttons and uses 5px horizontal and 10px vertical gaps. The window is shown in Figure 10-28.

### **Listing 10-17.** Using a Horizontal FlowPane

```
// FlowPaneTest.java
package com.jdojo.container;

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

public class FlowPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
```

```

    }

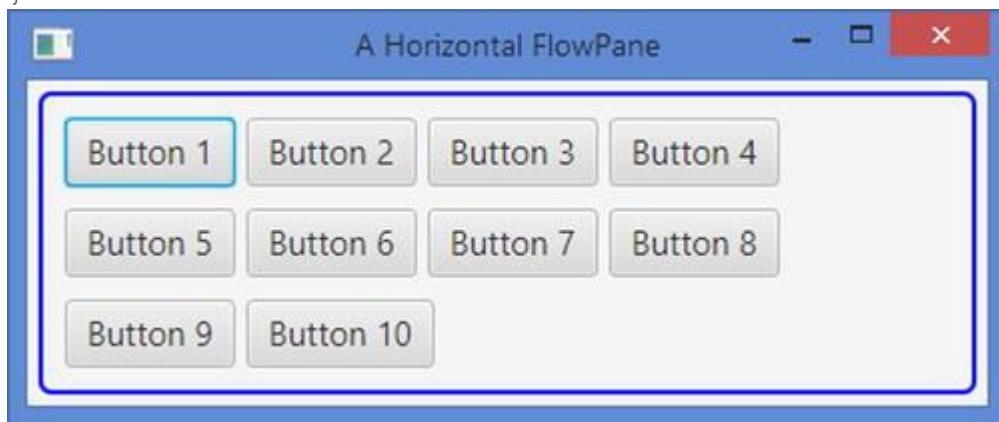
@Override
public void start(Stage stage) {
    double hgap = 5;
    double vgap = 10;
    FlowPane root = new FlowPane(hgap, vgap);

    // Add ten buttons to the flow pane
    for(int i = 1; i <= 10; i++) {
        root.getChildren().add(new Button("Button " +
+ i));
    }

    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("A Horizontal FlowPane");
    stage.show();
}
}

```



**Figure 10-28.** A horizontal pane with ten buttons using 5px hgap and 10px vgap

### FlowPane Properties

Table 10-4 lists several `FlowPane` class properties that are used to customize the layout of its children.

**Table 10-4.** The List of Properties Declared in the `FlowPane` Class

Property	Type	Description
----------	------	-------------

Property	Type	Description
alignment	ObjectProperty<Pos>	It specifies the alignment of rows and columns relative to the content area of the FlowPane. The default value is Pos.TOP_LEFT.
rowValignment	ObjectProperty<VPos>	It specifies the vertical alignment of children within each row in a horizontal FlowPane. It is ignored in a vertical FlowPane.
columnHalignment	ObjectProperty<HPos>	It specifies the horizontal alignment of children within each column in a vertical FlowPane. It is ignored in a horizontal FlowPane.
hgap, vgap	DoubleProperty	They specify the horizontal and vertical gaps between children. The default is 0.
orientation	ObjectProperty<Orientation>	It specifies the orientation of the FlowPane. It defaults to HORIZONTAL.
prefWrapLength	DoubleProperty	It is the preferred width in a horizontal FlowPane and the preferred height in a vertical FlowPane when the content should wrap. The default is 0.

## The Alignment Property

The alignment property of a FlowPane controls the alignment of its content. A Pos value contains a vertical alignment (vpos) and horizontal alignment (hpos). For example, Pos.TOP\_LEFT has the vertical alignment as top and horizontal alignment as left. In a horizontal FlowPane, each row is aligned using the hpos value of the alignment and rows (the entire content) is aligned using the vpos value. In a vertical FlowPane, each column is aligned using the vpos value.

value of the alignment and the columns (the entire content) are aligned using the hpos value.

The program in Listing 10-18 displays three FlowPanes in an HBox. Each FlowPane has a different alignment. The Text node in each FlowPane displays the alignment used. Figure 10-29 shows the window.

### ***Listing 10-18.*** Using the Alignment Property of the FlowPane

```
// FlowPaneAlignment.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class FlowPaneAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        FlowPane fp1 = createFlowPane(Pos.BOTTOM_RIGHT);
        FlowPane fp2 = createFlowPane(Pos.BOTTOM_LEFT);
        FlowPane fp3 = createFlowPane(Pos.CENTER);

        HBox root = new HBox(fp1, fp2, fp3);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("FlowPane Alignment");
        stage.show();
    }

    private FlowPane createFlowPane(Pos alignment) {
        FlowPane fp = new FlowPane(5, 5);
        fp.setPrefSize(200, 100);
        fp.setAlignment(alignment);
        fp.getChildren().addAll(new
Text(alignment.toString()),
                new Button("Button 1"),
                new Button("Button 2"),
                new Button("Button 3"));

        fp.setStyle("-fx-padding: 10;" +
                    "-fx-border-style: solid inside;" +
                    "-fx-border-width: 2;" +

```

```

        "-fx-border-insets: 5;" +
        "-fx-border-radius: 5;" +
        "-fx-border-color: blue;");

    return fp;
}
}
}

```



**Figure 10-29.** FlowPanes using different alignments for their contents

## The `rowVAlignment` and `columnHAlignment` Properties

A FlowPane lays out its children at their preferred sizes. Rows and columns could be of different sizes. You can align children in each row or column using the `rowVAlignment` and `columnHAlignment` properties. In a horizontal FlowPane, children in one row may be of different heights. The height of a row is the largest of the preferred heights of all children in the row. The `rowVAlignment` property lets you specify the vertical alignment of children in each row. Its value could be set to one of the constants of the `VPos` enum: `BASELINE`, `TOP`, `CENTER`, and `BOTTOM`. If the maximum height value of a child node allows for vertical expansion, the child node will be expanded to fill the height of the row. If the `rowVAlignment` property is set to `VPos.BASELINE`, children are resized to their preferred height instead of expanding to fill the full height of the row.

In a vertical FlowPane, children in one column may be of different widths. The width of a column is the largest of the preferred widths of all children in the column. The `columnHAlignment` property lets you specify the horizontal alignment of children in each column. Its value could be set to one of the constants of the `HPos` enum: `LEFT`, `RIGHT`, and `CENTER`. If the maximum width value of a child node allows for horizontal expansion, the child node will be expanded to fill the width of the column.

The program in Listing 10-19 creates three FlowPanes and adds them to an HBox. Figure 10-30 shows the window. The first two FlowPanes have horizontal orientations and the last one has a

vertical orientation. The row and column alignments are displayed in the Text node and the orientations for the FlowPane are displayed in the TextArea node.

### ***Listing 10-19.*** Using Row and Column Alignments in a FlowPane

```
// FlowPaneRowColAlignment.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.Orientation;
import static javafx.geometry.Orientation.HORIZONTAL;
import static javafx.geometry.Orientation.VERTICAL;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class FlowPaneRowColAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        FlowPane fp1 = createFlowPane(HORIZONTAL, VPos.TOP,
HPos.LEFT);
        FlowPane fp2 = createFlowPane(HORIZONTAL,
VPos.CENTER, HPos.LEFT);
        FlowPane fp3 = createFlowPane(VERTICAL, VPos.CENTER,
HPos.RIGHT);

        HBox root = new HBox(fp1, fp2, fp3);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("FlowPane Row and Column Alignment");
        stage.show();
    }

    private FlowPane createFlowPane(Orientation orientation,
                                    VPos rowAlign,
                                    HPos colAlign) {
        // Show the row or column alignment value in a Text
        Text t = new Text();
        if (orientation == Orientation.HORIZONTAL) {
            t.setText(rowAlign.toString());
        } else {
            t.setText(colAlign.toString());
        }
    }
}
```

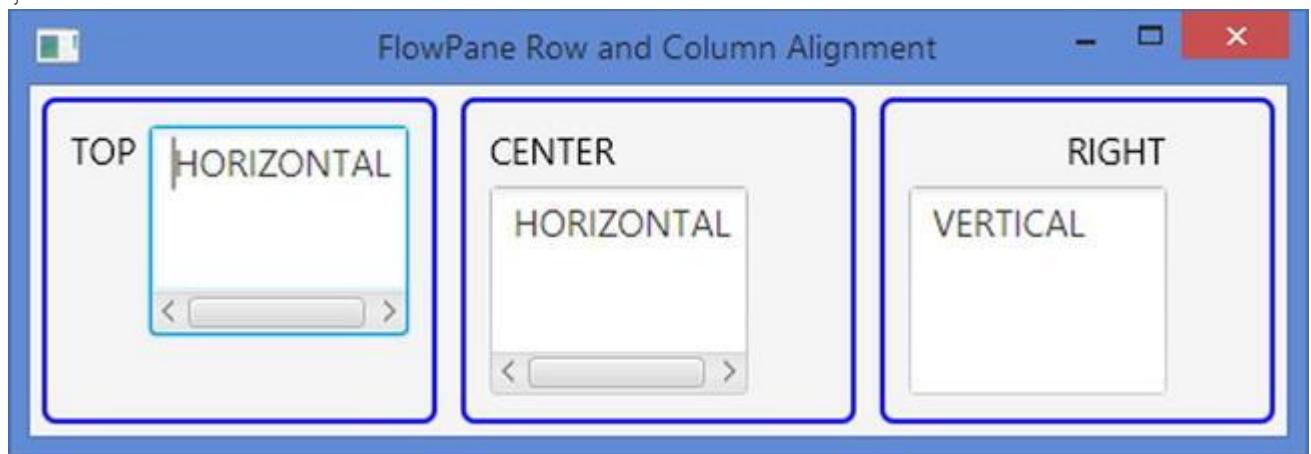
```

    // Show the orientation of the FlowPane in
a TextArea
    TextArea ta = new TextArea(orientation.toString());
    ta.setPrefColumnCount(5);
    ta.setPrefRowCount(3);

    FlowPane fp = new FlowPane(orientation, 5, 5);
    fp.setRowValignment(rowAlign);
    fp.setColumnHalignment(colAlign);
    fp.setPrefSize(175, 130);
    fp.getChildren().addAll(t, ta);
    fp.setStyle("-fx-padding: 10;" +
                "-fx-border-style: solid inside;" +
                "-fx-border-width: 2;" +
                "-fx-border-insets: 5;" +
                "-fx-border-radius: 5;" +
                "-fx-border-color: blue;");

    return fp;
}
}

```



**Figure 10-30.** FlowPanes using different row and column alignments

## The hgap and vgap Properties

Using the `hgap` and `vgap` properties is straightforward. In a horizontal `FlowPane`, the `hgap` property specifies the horizontal spacing between adjacent children in a row and the `vgap` property specifies the spacing between adjacent rows. In a vertical `FlowPane`, the `hgap` property specifies the horizontal spacing between adjacent columns and the `vgap` property specifies the spacing between adjacent children in a column. You can set these properties in the constructors or using the setter methods. We have been using these properties in our examples discussed in this section.

```
// Create a FlowPane with 5px hgap and 10px vgap
FlowPane fpane = new FlowPane(5, 10);
```

```
...
// Change the hgap to 15px and vgap to 25px
fpane.setHgap(15);
fpane.setVgap(25);
```

## The Orientation Property

The `orientation` property specifies the flow of content in a `FlowPane`. If it is set to `Orientation.HORIZONTAL`, which is the default value, the content flows in rows. If it is set to `Orientation.VERTICAL`, the content flows in columns. You can specify the `orientation` in the constructors or using the setter method.

```
// Create a horizontal FlowPane
FlowPane fpane = new FlowPane();
...
// Change the orientation of the FlowPane to vertical
fpane.setOrientation(Orientation.VERTICAL);
```

## The `prefWrapLength` Property

The `prefWrapLength` property is the preferred width in a horizontal `FlowPane` or the preferred height in a vertical `FlowPane` where content should wrap. This is only used to compute the preferred size of the `FlowPane`. It defaults to 400. Treat the value of this property as a hint to resize your `FlowPane`. Suppose you set this value to less than the largest preferred width or height of a child node. In this case, this value will not be respected, as a row cannot be shorter than the widest child node in a horizontal `FlowPane` or a column cannot be shorter than the tallest child node in a vertical `FlowPane`. If 400px is too wide or tall for your `FlowPane`, set this value to a reasonable value.

### Content Bias of a `FlowPane`

Notice that the number of rows in a horizontal `FlowPane` depends on its width and the number of columns in a vertical `FlowPane` depends on its height. That is, a horizontal `FlowPane` has a horizontal content bias and a vertical `FlowPane` has a vertical content bias. Therefore, when you are getting the size of a `FlowPane`, make sure to take into account its content bias.

## Understanding `BorderPane`

A `BorderPane` divides its layout area into five regions: top, right, bottom, left, and center. You can place at most one node in each of the

five regions. Figure 10-31 shows five Buttons placed in the five regions of the BorderPane – one Button in each region. The Buttons have been labeled the same as their regions in which they are placed. Any of the regions may be null. If a region is null, no space is allocated for it.



**Figure 10-31.** Five regions of a BorderPane

In a typical Windows application, a screen uses the five regions to place its content.

- A menu or a toolbar at the top
- A status bar at the bottom
- A navigation panel on the left
- Additional information on the right
- Main content in the center

A BorderPane satisfies all the layout requirements for a typical Windows-based GUI screen. This is the reason that a BorderPane is most often used as the root node for a scene. Typically, you have more than five nodes in a window. If you have more than one node to place in one of the five regions of a BorderPane, add the nodes to a layout pane: for example, an HBox, a VBox, etc., and then add the layout pane to the desired region of the BorderPane.

A BorderPane uses the following resizing policies for its children:

- The children in the top and bottom regions are resized to their preferred heights. Their widths are extended to fill the available extra horizontal space, provided the maximum widths of the children allow extending their widths beyond their preferred widths.
- The children in the right and left regions are resized to their preferred widths. Their heights are extended to fill the extra vertical space, provided the maximum heights of the children allow extending their heights beyond their preferred heights.

- The child node in the center will fill the rest of the available space in both directions.

Children in a `BorderPane` may overlap if it is resized to a smaller size than its preferred size. The overlapping rule is based on the order in which the children are added. The children are drawn in the order they are added. This means that a child node may overlap all child nodes added prior to it. Suppose regions are populated in the order of right, center, and left. The left region may overlap the center and right regions, and the center region may overlap the right region.

**Tip** You can set the alignments for all children within their regions. You can set the margins for children. As with all layout panes, you can also style a `BorderPane` with CSS.

## Creating BorderPane Objects

The `BorderPane` class provides constructors to create `BorderPane` objects with or without children.

```
// Create an empty BorderPane
BorderPane bpane1 = new BorderPane();

// Create a BorderPane with a TextArea in the center
TextArea center = new TextArea();
BorderPane bpane2 = new BorderPane(center);

// Create a BorderPane with a Text node in each of the five
regions
Text center = new Text("Center");
Text top = new Text("Top");
Text right = new Text("Right");
Text bottom = new Text("Bottom");
Text left = new Text("Left");
BorderPane bpane3 = new BorderPane(center, top, right, bottom,
left);
```

The `BorderPane` class declares five properties named `top`, `right`, `bottom`, `left`, and `center` that store the reference of five children in the five regions. Use the setters for these properties to add a child node to any of the five regions. For example, use the `setTop(Node topChild)` method to add a child node to the top region. To get the reference of the children in any of the five regions, use the getters for these properties. For example, the `getTop()` method returns the reference of the child node in the top region.

```
// Create an empty BorderPane and add a text node in each of the
five regions
BorderPane bpane = new BorderPane();
bpane.setTop(new Text("Top"));
bpane.setRight(new Text("Right"));
bpane.setBottom(new Text("Bottom"));
```

```
bpane.setLeft(new Text("Left"));
bpane.setCenter(new Text("Center"));
```

**Tip** Do not use the ObservableList<Node>, which is returned by the getChildren() method of the BorderPane, to add children to a BorderPane. The children added to this list are ignored. Use the top, right, bottom, left, and center properties instead.

The program in Listing 10-20 shows how to create a BorderPane and add children. It adds children to the right, bottom, and center regions. Two Labels, a TextField, and a TextArea are added to the center region. A VBox with two buttons are added to the right region. A Label to show the status is added to the bottom region. The top and left regions are set to null. The BorderPane is set as the root node for the scene. Figure 10-32 shows the window.

### ***Listing 10-20.*** Using the BorderPane Layout Pane

```
// BorderPaneTest.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BorderPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Set the top and left child nodes to null
        Node top = null;
        Node left = null;

        // Build the content nodes for the center region
        VBox center = getCenter();

        // Create the right child node
        Button okBtn = new Button("Ok");
        Button cancelBtn = new Button("Cancel");
```

```
// Make the OK and cancel buttons the same size
okBtn.setMaxWidth(Double.MAX_VALUE);
VBox right = new VBox(okBtn, cancelBtn);
right.setStyle("-fx-padding: 10;");

// Create the bottom child node
Label statusLbl = new Label("Status: Ready");
HBox bottom = new HBox(statusLbl);
BorderPane.setMargin(bottom, new Insets(10, 0, 0,
0));
bottom.setStyle("-fx-background-color: lavender;" +
"-fx-font-size: 7pt;" +
"-fx-padding: 10 0 0 0;" );

BorderPane root = new BorderPane(center, top, right,
bottom, left);
root.setStyle("-fx-background-color: lightgray;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using a BorderPane");
stage.show();
}

private VBox getCenter() {
    // A Label and a TextField in an HBox
    Label nameLbl = new Label("Name:");
    TextField nameFld = new TextField();
    HBox.setHgrow(nameFld, Priority.ALWAYS);
    HBox nameFields = new HBox(nameLbl, nameFld);

    // A Label and a TextArea
    Label descLbl = new Label("Description:");
    TextArea descText = new TextArea();
    descText.setPrefColumnCount(20);
    descText.setPrefRowCount(5);
    VBox.setVgrow(descText, Priority.ALWAYS);

    // Box all controls in a VBox
    VBox center = new VBox(nameFields, descLbl,
descText);

    return center;
}
}
```



**Figure 10-32.** A BorderPane using some controls in its top, right, bottom, and center regions

## BorderPane Properties

The BorderPane class declares five properties: `top`, `right`, `bottom`, `left`, and `center`. They are of the `ObjectProperty<Node>` type. They store the reference of the child node nodes in the five regions of the BorderPane. Use the setters of these properties to add children to the BorderPane. Use the getters of properties to get the reference of the child node in any regions.

Recall that not all of the five regions in a BorderPane need to have nodes. If a region does not have a node, no space is allocated for it. Use `null` to remove a child node from a region. For example, `setTop(null)` will remove the already added node to the top region. By default, all regions have `null` nodes as their child nodes.

## Setting Constraints for Children in BorderPane

A BorderPane allows you to set alignment and margin constraints on individual children. The alignment for a child node is defined relative to its region. The default alignments:

- `Pos.TOP_LEFT` for the top child node
- `Pos.BOTTOM_LEFT` for the bottom child node
- `Pos.TOP_LEFT` for the left child node
- `Pos.TOP_RIGHT` for the right child node

- `Pos.CENTER` for the center child node

**Use the `setAlignment(Node child, Pos value)` static method of the `BorderPane` class to set the alignment for children.**

**The `getAlignment(Node child)` static method returns the alignment for a child node.**

```
BorderPane root = new BorderPane();
Button top = new Button("OK");
root.setTop(top);

// Place the OK button in the top right corner (default is top left)
BorderPane.setAlignment(top, Pos.TOP_RIGHT);
...
// Get the alignment of the top node
Pos alignment = BorderPane.getAlignment(top);
```

**Use the `setMargin(Node child, Insets value)` static method of the `BorderPane` class to set the margin for the children.**

**The `getMargin(Node child)` static method returns the margin for a child node.**

```
// Set 10px margin around the top child node
BorderPane.setMargin(top, new Insets(10));
...
// Get the margin of the top child node
Insets margin = BorderPane.getMargin(top);
```

**Use null to reset the constraints to the default value. Use the `clearConstraints(Node child)` static method of the `BorderPane` to reset all constraints for a child at once.**

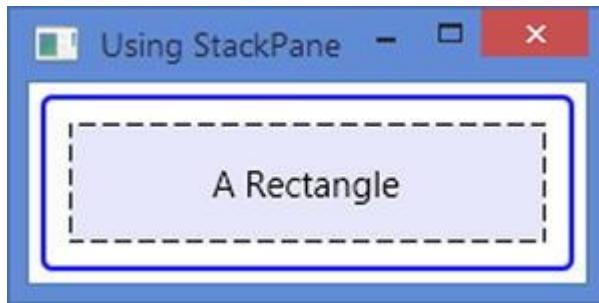
```
// Clear the alignment and margin constraints for the top child node
BorderPane.clearConstraints(top);
```

## Understanding StackPane

A `StackPane` lays out its children in a stack of nodes. It is simple to use. However, it provides a powerful means to overlay nodes. Children are drawn in the order they are added. That is, the first child node is drawn first; the second child node is drawn next, etc. For example, overlaying text on a shape is as easy as using a `StackPane`: add the shape as the first child node and the text as the second child node. The shape will be drawn first followed by the text, which makes it seem as if the text is a part of the shape.

Figure 10-33 shows a window with a `StackPane` set as the root node for its scene. A `Rectangleshape` and a `Text` node with text “A Rectangle” are added to the `StackPane`. The `Text` is added last, which overlays

the Rectangle. The outer border is the border of the StackPane. The dashed inner border is the border of the Rectangle.



**Figure 10-33.** A Text node overlaying a Rectangle in a StackPane

**Tip** You can create very appealing GUI using StackPanes by overlaying different types of nodes. You can overlay text on an image to get an effect as if the text were part of the image. And you can overlay different types of shapes to create a complex shape. Remember that the node that overlays other nodes is added last to the StackPane.

The preferred width of a StackPane is the width of its widest children. Its preferred height is the height of its tallest children. StackPane does clip its content. Therefore, its children may be drawn outside its bounds. A StackPane resizes its resizable children to fill its content area, provided their maximum size allows them to expand beyond their preferred size. By default, a StackPane aligns all its children to the center of its content area. You can change the alignment for a child node individually or for all children to use the same alignment.

### Creating StackPane Objects

The StackPane class provides constructors to create objects with or without children.

```
// Create an empty StackPane
StackPane spanel = new StackPane();

// Add a Rectangle and a Text to the StackPane
Rectangle rect = new Rectangle(200, 50);
rect.setFill(Color.LAVENDER);
Text text = new Text("A Rectangle");
spanel.getChildren().addAll(rect, text);

// Create a StackPane with a Rectangle and a Text
StackPane spane2 = new StackPane(RectangleBuilder.create()
    .width(200)
    .height(50)
    .fill(Color.LAVENDER)
    .build(),
    new Text("A Rectangle"));
```

The program in Listing 10-21 shows how to create a StackPane. It adds a Rectangle and a Text to a StackPane. The Rectangle is added first, and therefore it is overlaid with the Text. Figure 10-33 shows the window.

### ***Listing 10-21.*** Using StackPane

```
// StackPaneTest.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class StackPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a Rectangle and a Text
        Rectangle rect = new Rectangle(200, 50);
        rect.setStyle("-fx-fill: lavender;" +
                     "-fx-stroke-type: inside;" +
                     "-fx-stroke-dash-array: 5 5;" +
                     "-fx-stroke-width: 1;" +
                     "-fx-stroke: black;" +
                     "-fx-stroke-radius: 5;");

        Text text = new Text("A Rectangle");

        // Create a StackPane with a Rectangle and a Text
        StackPane root = new StackPane(rect, text);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using StackPane");
        stage.show();
    }
}
```

You must add the children to a StackPane in a specific order to create the desired overlay. Children are drawn in the order they exist in the list. The following two statements will not get the same results.

```
// Overlay a Text on a Rectangle
spanel.getChildren().addAll(rect, text);

// Overlay a Rectangle on a Text
spanel.getChildren().addAll(text, rect);
```

If the Text is smaller than the Rectangle, overlaying the Rectangle on the Text will hide the Text. If the Text size is bigger than the Rectangle, the part of the Text outside the Rectangle bounds will be visible.

The program in Listing 10-22 shows how the overlay rules work in a StackPane. The `createStackPane()` method creates a StackPane with a Rectangle and a Text. It takes the text for the Text node, the opacity of the Rectangle, and a boolean value indicating whether the Rectangle should be added first to the StackPane. The start method creates five StackPanes and adds them to an HBox. Figure 10-34 shows the window.

- In the first StackPane, the text is overlaid on the rectangle. The rectangle is drawn first and the text second. Both are visible.
- In the second StackPane, the rectangle is overlaid on the text. The text is hidden behind the rectangle as the rectangle is drawn over the text and it is bigger than the text.
- In the third StackPane, the rectangle is overlaid on the text. Unlike the second StackPane, the text is visible because we have set the opacity for the rectangle to 0.5, which makes it 50% transparent.
- In the fourth StackPane, the rectangle is overlaid on a big text. The opacity of the rectangle is 100%. Therefore, we see only the part of the text that is outside the bounds of the rectangle.
- In the fifth StackPane, the rectangle is overlaid on a big text. The opacity of the rectangle is 50%. We can see the entire text. The visibility of the text within the bounds of the rectangle is 50% and that of outside the bounds is 100%.

### ***Listing 10-22.*** Overlaying Rules in a StackPane

```
// StackPaneOverlayTest.java
package com.jdojo.container;
```

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class StackPaneOverlayTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        StackPane textOverRect = createStackPane("Hello",
1.0, true);
        StackPane rectOverText = createStackPane("Hello",
1.0, false);
        StackPane transparentRectOverText
= createStackPane("Hello", 0.5, false);
        StackPane rectOverBigText = createStackPane("A
bigger text", 1.0, false);
        StackPane transparentRectOverBigText
= createStackPane("A bigger text",
0.5,
false);

        // Add all StackPanes to an HBox
        HBox root = new HBox(textOverRect,
rectOverText,
transparentRectOverText,
rectOverBigText,
transparentRectOverBigText);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Overlaying Rules in StackPane");
        stage.show();
    }

    public StackPane createStackPane(String str, double
rectOpacity, boolean rectFirst) {
        Rectangle rect = new Rectangle(60, 50);
        rect.setStyle("-fx-fill: lavender;" + "-fx-opacity:
" + rectOpacity + ";");

        Text text = new Text(str);

        // Create a StackPane
        StackPane spane = new StackPane();

        // add the Rectangle before the Text if rectFirst is
true.
        // Otherwise add the Text first
    }
}

```

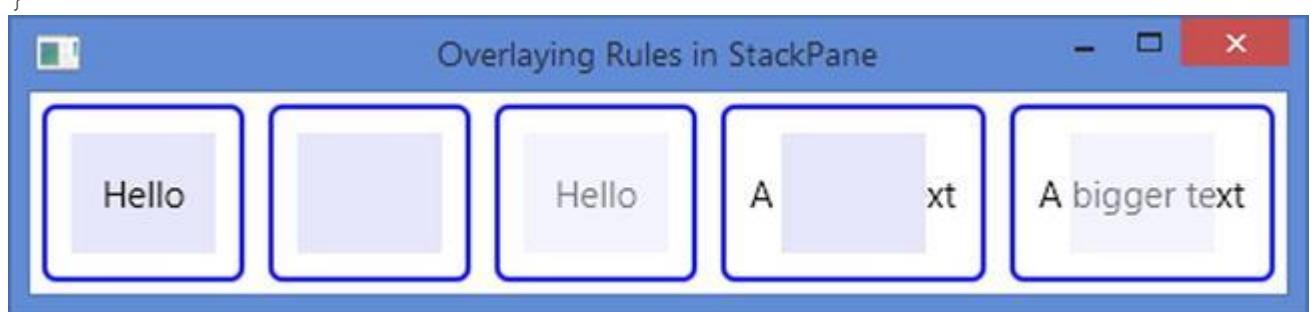
```

        if (rectFirst) {
            spane.getChildren().addAll(rect, text);
        } else {
            spane.getChildren().addAll(text, rect);
        }

        spane.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        return spane;
    }
}

```



**Figure 10-34.** Overlaying a Rectangle on a Text and vice versa

### StackPane Properties

The `StackPane` class has an `alignment` property of the `ObjectProperty<Pos>` type. The property defines the default alignment of all children within the content area of the `StackPane`. By default, its value is set to `Pos.CENTER`, which means that all children, by default, are aligned in the center of the content area of the `StackPane`. This is what we have seen in our previous examples. If you do not want the default alignment for all children, you can change it to any other alignment value. Note that changing the value of the `alignment` property sets the default alignment for all children. Individual children may override the default alignment by setting its `alignment` constraint. We will discuss how to set the `alignment` constraint on a child node in the next section.

`StackPane` has several other uses besides overlaying nodes. Whenever you have a requirement to align a node or a collection of nodes in a specific position, try using a `StackPane`. For example, if you want to display text in the center of your screen, use a `StackPane` with a `Text` node as the root node of the scene. The `StackPane` takes care of keeping the text in the center as the window is resized. Without

a StackPane, you will need to use binding to keep the text positioned in the center of the window.

The program in Listing 10-23 uses five StackPanes in an HBox. Each StackPane has a Rectangle overlaid with a Text. The alignment for the StackPane, and hence for all its children, is used as the text for the Text node. Figure 10-35 shows the window. Notice that the Rectangles in StackPanes are bigger than the Texts. Therefore, the Rectangles occupy the entire content area of the StackPanes and they seem not to be affected by the alignment property.

### ***Listing 10-23.*** Using the Alignment Property of a StackPane

```
// StackPaneAlignment.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class StackPaneAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        StackPane topLeft = createStackPane(Pos.TOP_LEFT);
        StackPane topRight = createStackPane(Pos.TOP_RIGHT);
        StackPane bottomLeft
= createStackPane(Pos.BOTTOM_LEFT);
        StackPane bottomRight
= createStackPane(Pos.BOTTOM_RIGHT);
        StackPane center = createStackPane(Pos.CENTER);

        double spacing = 10.0;
        HBox root = new HBox(spacing,
                            topLeft,
                            topRight,
                            bottomLeft,
                            bottomRight,
                            center);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using StackPane");
    }
}
```

```

        stage.show();
    }

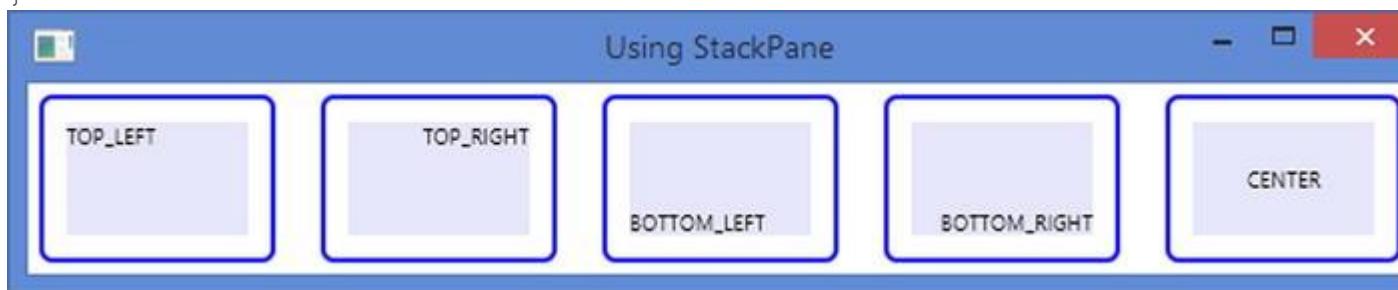
    public StackPane createStackPane(Pos alignment) {
        Rectangle rect = new Rectangle(80, 50);
        rect.setFill(Color.LAVENDER);

        Text text = new Text(alignment.toString());
        text.setStyle("-fx-font-size: 7pt;");

        StackPane spane = new StackPane(rect, text);
        spane.setAlignment(alignment);
        spane.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        return spane;
    }
}

```



**Figure 10-35.** StackPanes using different alignment values

### Setting Constraints for Children

A StackPane allows you to set alignment and margin constraints on individual children. The alignment for a child node is defined relative to the content area of the StackPane.

You should be able to differentiate between the `alignment` property of a StackPane and the alignment constraint on its children.

The `alignment` property affects all children. Its value is used to align children by default. The alignment constraint on a child node overrides the default alignment value set by the `alignment` property. The alignment constraint on a child node affects the alignment of only that child node, whereas the `alignment` property affects all child nodes.

When a child node is drawn, JavaFX uses the alignment constraint of the child node for aligning it within the content area of the StackPane. If its alignment constraint is not set, the `alignment` property of the StackPane is used.

**Tip** The default value for the alignment property of StackPane is Pos.CENTER. The default value for the alignment constraint for children is null.

Use the setAlignment(Node child, Pos value) static method of the StackPane class to set the alignment constraints for children. The getAlignment(Node child) static method returns the alignment for a child node.

```
// Place a Text node in the top left corner of the StackPane
Text topLeft = new Text("top-left");
StackPane.setAlignment(topLeft, Pos.TOP_LEFT);
StackPane root = new StackPane(topLeft);
...
// Get the alignment of the topLeft node
Pos alignment = StackPane.getAlignment(topLeft);
```

### ***Listing 10-24.*** Using the Alignment Constraints for Children in a StackPane

```
// StackPaneAlignmentConstraint.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class StackPaneAlignmentConstraint extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(200, 60);
        rect.setFill(Color.LAVENDER);

        // Create a Text node with the default CENTER
        alignment
        Text center = new Text("Center");

        // Create a Text node with a TOP_LEFT alignemnt
        constraint
        Text topLeft = new Text("top-left");
        StackPane.setAlignment(topLeft, Pos.TOP_LEFT);

        // Create a Text node with a BOTTOM_LEFT alignemnt
        constraint
```

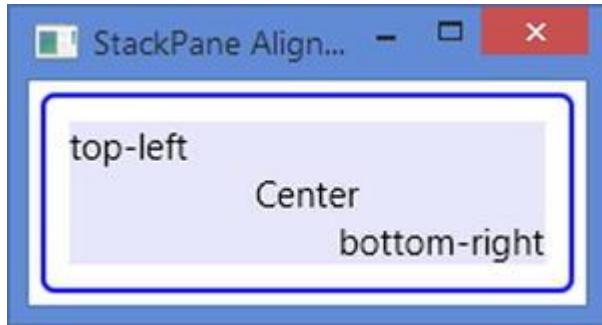
```

        Text bottomRight = new Text("bottom-right");
        StackPane.setAlignment(bottomRight,
Pos.BOTTOM_RIGHT);

        StackPane root = new StackPane(rect, center,
topLeft, bottomRight);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("StackPane Alignment Constraint");
        stage.show();
    }
}

```



**Figure 10-36.** Children using different alignment constraints in a StackPane

Use the `setMargin(Node child, Insets value)` static method of the `StackPane` class to set the margin for children.

The `getMargin(Node child)` static method returns the margin for a child node.

```

// Set 10px margin around the topLeft child node
StackPane.setMargin(topLeft, new Insets(10));
...
// Get the margin of the topLeft child node
Insets margin = StackPane.getMargin(topLeft);

```

Use `null` to reset the constraints to the default value. Use the `clearConstraints(Node child)` static method of the `StackPane` to reset all constraints for a child at once.

```

// Clear the alignment and margin constraints for the topLeft
child node
StackPane.clearConstraints(topLeft);

```

After you clear all constraints for a child node, it will use the current value of the `alignment` property of the `StackPane` as its alignment and `opx` as the margins.

## Understanding TilePane

A `TilePane` lays out its children in a grid of uniformly sized cells, known as tiles. `TilePanes` work similar to `FlowPanes` with one difference: In a `FlowPane`, rows and columns can be of different heights and widths, whereas in a `TilePane`, all rows have the same heights and all columns have the same widths. The width of the widest child node and the height of the tallest child node are the default widths and heights of all tiles in a `TilePane`.

The orientation of a `TilePane`, which can be set to horizontal or vertical, determines the direction of the flow for its content. By default, a `TilePane` has a horizontal orientation. In a horizontal `TilePane`, the content flows in rows. The content in rows may flow from left to right (the default) or from right to left. In a vertical `TilePane`, the content flows in columns. Figures 10-37, 10-38, and 10-26 show horizontal and vertical `TilePanes`.



**Figure 10-37.** A horizontal `TilePane` showing months in a year



**Figure 10-38.** A vertical `TilePane` showing months in a year

You can customize the layout in a `TilePane` using its properties or setting constraints on individual children:

- You can override the default size of tiles.
- You can customize the alignment of the entire content of a `TilePane` within its content area, which defaults to `Pos.TOP_LEFT`.
- You can also customize the alignment of each child node within its tile, which defaults to `Pos.CENTER`.
- You specify the spacing between adjacent rows and columns, which defaults to `0px`.
- You can specify the preferred number of columns in a horizontal `TilePane` and the preferred number of rows in a vertical `TilePane`. The default values for the preferred number of rows and columns are five.

## Creating `TilePane` Objects

The `TilePane` class provides several constructors to create `TilePane` objects with a specified orientation (horizontal or vertical), a specified horizontal and vertical spacing between children, and a specified initial list of children.

```
// Create an empty horizontal TilePane with 0px spacing
TilePane tpane1 = new TilePane();

// Create an empty vertical TilePane with 0px spacing
TilePane tpane2 = new TilePane(Orientation.VERTICAL);

// Create an empty horizontal TilePane with 5px horizontal
// and 10px vertical spacing
TilePane tpane3 = new TilePane(5, 10);

// Create an empty vertical TilePane with 5px horizontal
// and 10px vertical spacing
TilePane tpane4 = new TilePane(Orientation.VERTICAL, 5, 10);

// Create a horizontal TilePane with two Buttons and 0px spacing
TilePane tpane5 = new TilePane(new Button("Button 1"), new
Button("Button 2"));
```

The program in **Listing 10-25** shows how to create a `TilePane` and add children. It uses the `Monthenum` from the `java.time` package to get the names of ISO months. Note that `java.time` package was added in Java 8. The resulting window is the same as shown in Figure 10-37.

### **Listing 10-25.** Using `TilePane`

```
// TilePaneTest.java
package com.jdojo.container;

import java.time.Month;
import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class TilePaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        double hgap = 5.0;
        double vgap = 5.0;
        TilePane root = new TilePane(hgap, vgap);
        root.setPrefColumns(5);

        // Add 12 Buttons - each having the name of the 12
months
        for(Month month: Month.values()) {
            Button b = new Button(month.toString());
            b.setMaxHeight(Double.MAX_VALUE);
            b.setMaxWidth(Double.MAX_VALUE);
            root.getChildren().add(b);
        }

        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Horizontal TilePane");
        stage.show();
    }
}

```

You can modify the code in Listing 10-25 to get the window in Figure 10-38. You need to specify the orientation of the TilePane as Orientation.VERTICAL and use three as the preferred number of rows.

```

import javafx.geometry.Orientation;
...
double hgap = 5.0;
double vgap = 5.0;
TilePane root = new TilePane(Orientation.VERTICAL, hgap, vgap);
root.setPrefRows(3);

```

## TilePane Properties

The `TilePane` class contains several properties, as listed in Table 10-5, which let you customize the layout of its children.

**Table 10-5.** The List of Properties Declared in the `TilePane` Class

Property	Type	Description
<code>alignment</code>	<code>ObjectProperty&lt;Pos&gt;</code>	It specifies the alignment of the <code>TilePane</code> relative area. It defaults to <code>Pos.CENTER</code> .
<code>tileAlignment</code>	<code>ObjectProperty&lt;Pos&gt;</code>	It specifies the default alignment of children within their tiles. It defaults to <code>Pos.CENTER</code> .
<code>hgap, vgap</code>	<code>DoubleProperty</code>	The <code>hgap</code> property specifies the horizontal gap between adjacent children in a row. The <code>vgap</code> specifies the vertical gap between adjacent children in a column. The default is zero for both properties.
<code>orientation</code>	<code>ObjectProperty&lt;Orientation&gt;</code>	It specifies the orientation of the <code>TilePane</code> – horizontal or vertical. It defaults to <code>HORIZONTAL</code> .
<code>prefRows</code>	<code>IntegerProperty</code>	It specifies the preferred number of rows for a vertical <code>TilePane</code> . It is ignored for a horizontal <code>TilePane</code> .
<code>prefColumns</code>	<code>IntegerProperty</code>	It specifies the preferred number of columns for a horizontal <code>TilePane</code> . It is ignored for a vertical <code>TilePane</code> .
<code>prefTileWidth</code>	<code>DoubleProperty</code>	It specifies the preferred width of a tile. The default is to use the widest children.

Property	Type	Description
prefTileHeight	DoubleProperty	It specifies the preferred height of each tile. The default is to use the tallest children.
tileHeight	ReadOnlyDoubleProperty	It is a read-only property that returns the actual height of each tile.
tileWidth	ReadOnlyDoubleProperty	It is a read-only property that returns the actual width of each tile.

## The Alignment Property

The alignment property of a TilePane controls the alignment of its content within its content area. You can see the effects of this property when the size of the TilePane is bigger than its content. The property works the same way as the alignment property for the FlowPane. Please refer to the description of the alignment property for FlowPane for more details and illustrations.

## The tileAlignment Property

The tileAlignment property specifies the default alignment of children within their tiles. Note that this property affects children smaller than the size of tiles. This property affects the default alignment of all children within their tiles. This can be overridden on individual children by setting their alignment constraints. The program in Listing 10-26 shows how to use the tileAlignment property. It shows display windows, as shown in Figure 10-39, with two TilePanes – one has the tileAlignment property set to Pos.CENTER and another Pos.TOP\_LEFT.

### ***Listing 10-26.*** Using the TileAlignment Property of TilePane

```
// TilePaneTileAlignment.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
```

```
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class TilePaneTileAlignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TilePane tileAlignCenter
= createTilePane(Pos.CENTER);
        TilePane tileAlignTopRight
= createTilePane(Pos.TOP_LEFT);

        HBox root = new HBox(tileAlignCenter,
tileAlignTopRight);
        root.setFillHeight(false);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("The tileAlignment Property for
TilePane");
        stage.show();
    }

    public TilePane createTilePane(Pos tileAlignment) {
        Button[] buttons = new Button[] {new Button("Tile"),
                                         new Button("are"),
                                         new Button("aligned"),
                                         new Button("at"),
                                         new
Button(tileAlignment.toString())};

        TilePane tpane = new TilePane(5, 5, buttons);
        tpane.setTileAlignment(tileAlignment);
        tpane.setPrefColumns(3);
        tpane.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;" );
        return tpane;
    }
}
```



**Figure 10-39.** Using the `tileAlignment` property

## The `hgap` and `vgap` Properties

The `hgap` and `vgap` properties specify the spacing between adjacent columns and adjacent rows. They default to zero. They can be specified in the constructors or using the `setHgap(double hg)` and `setVgap(double vg)` methods of the `TilePane`.

## The Orientation Property

The `orientation` property specifies the flow of content in a `TilePane`. If it is set to `Orientation.HORIZONTAL`, which is the default value, the content flows in rows. If it is set to `Orientation.VERTICAL`, the content flows in columns. You can specify the orientation in the constructors or using the setter method.

```
// Create a horizontal TilePane
TilePane tpane = new TilePane();
...
// Change the orientation of the TilePane to vertical
tpane.setOrientation(Orientation.VERTICAL);
```

## The `prefRows` and `prefColumns` Properties

The `prefRows` property specifies the preferred number of rows for a vertical `TilePane`. It is ignored for a horizontal `TilePane`.

The `prefColumns` specifies the preferred number of columns for a horizontal `TilePane`. It is ignored for a vertical `TilePane`.

The default values for `prefRows` and `prefColumns` is 5. It is recommended that you use a sensible value for these properties.

Note that these properties are only used to compute the preferred size of the `TilePane`. If the `TilePane` is resized to a size other than its preferred size, these values may not reflect the actual number of rows or columns. In Listing 10-26, we have specified three as the preferred number of columns. If you resize the window displayed by Listing 10-

26 to a smaller width, you may get only one or two columns, and the number of rows will increase accordingly.

**Tip** Recall the `prefWrapLength` property of the `FlowPane` that is used to determine the preferred width or height of the `FlowPane`. The `prefRows` and `prefColumns` properties serve the same purpose in a `TilePane`, as does the `prefWrapLength` in a `FlowPane`.

## The `prefTileWidth` and `prefTileHeight` Properties

A `TilePane` computes the preferred size of its tiles based on the widest and the tallest children. You can override the computed width and height of tiles using the `prefTileWidth` and `prefTileHeight` properties. They default to `Region.USE_COMPUTED_SIZE`.

The `TilePane` attempts to resize its children to fit in the tile size, provided their minimum and maximum size allows them to be resized.

```
// Create a TilePane and set its preferred tile width and height
// to 40px
TilePane tpane = new TilePane();
tpane.setPrefTileWidth(40);
tpane.setPrefTileHeight(40);
```

## The `tileWidth` and `tileHeight` Properties

The `tileWidth` and `tileHeight` properties specify the actual width and height of each tile. They are read-only properties. If you specify the `prefTileWidth` and `prefTileHeight` properties, they return their values. Otherwise, they return the computed size of tiles.

### Setting Constraints for Children in `TilePane`

A `TilePane` allows you to set alignment and margin constraints on individual children. The alignment for a child node is defined within the tile that contains the child node.

You should be able to differentiate between the three:

- The `alignment` property of a `TilePane`
- The `tileAlignment` property of the `TilePane`
- The alignment constraint on individual children of the `TilePane`

The `alignment` property is used to align the content (all children) within the content area of the `TilePane`. It affects the content of `TilePane` as a whole.

The `tileAlignment` property is used to align all children within their tiles by default. Modifying this property affects all children.

The alignment constraint on a child node is used to align the child node within its tile. It affects only the child node on which it is set. It overrides the default alignment value for the child node that is set using the `tileAlignment` property of the `TilePane`.

**Tip** The default value for the `tileAlignment` property of a `TilePane` is `Pos.CENTER`. The default value for the alignment constraint for children is `null`.

Use the `setAlignment(Node child, Pos value)` static method of the `TilePane` class to set the alignment constraints for the children.

The `getAlignment(Node child)` static method returns the alignment for a child node.

```
// Place a Text node in the top left corner in a tile
Text topLeft = new Text("top-left");
TilePane.setAlignment(topLeft, Pos.TOP_LEFT);

TilePane root = new TilePane();
root.getChildren().add(topLeft);

...
// Get the alignment of the topLeft node
Pos alignment = TilePane.getAlignment(topLeft);
```

The program in Listing 10-27 adds five buttons to a `TilePane`. The button labeled “Three” uses a custom tile alignment constraint of `Pos.BOTTOM_RIGHT`. All other buttons use the default tile alignment, which is `Pos.CENTER`. Figure 10-40 shows the window.

### ***Listing 10-27.*** Using the Alignment Constraints for Children in a `TilePane`

```
// TilePaneAlignmentConstraint.java
package com.jdojo.container;

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

public class TilePaneAlignmentConstraint extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
```

```

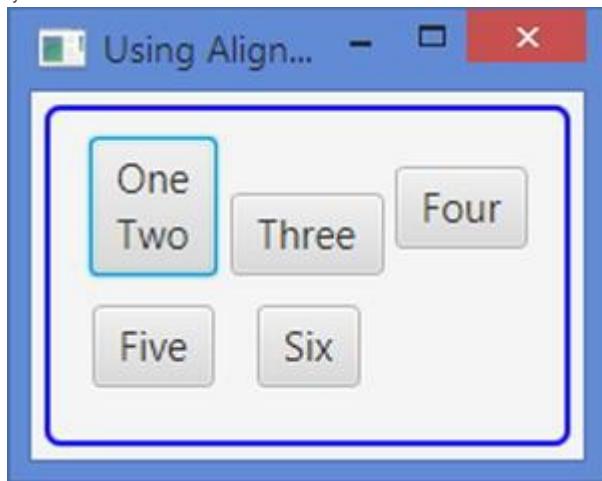
        Button b12 = new Button("One\nTwo");
        Button b3 = new Button("Three");
        Button b4 = new Button("Four");
        Button b5 = new Button("Five");
        Button b6 = new Button("Six");

        // Set the tile alignment constraint on b3 to
BOTTOM_RIGHT
        TilePane.setAlignment(b3, Pos.BOTTOM_RIGHT);

        TilePane root = new TilePane(b12, b3, b4, b5, b6);
        root.setPrefColumns(3);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Alignment Constraints in
TilePane");
        stage.show();
    }
}

```



**Figure 10-40.** Children using different alignment constraints in a TilePane

Use the `setMargin(Node child, Insets value)` static method of the `TilePane` class to set the margin for children.

The `getMargin(Node child)` static method returns the margin for a child node.

```

// Set 10px margin around the topLeft child node
TilePane.setMargin(topLeft, new Insets(10));
...
// Get the margin of the topLeft child node
Insets margin = TilePane.getMargin(topLeft);

```

Use `null` to reset the constraints to the default value. Use the `clearConstraints(Node child)` static method of the `TilePane` to reset all constraints for a child at once.

```
// Clear the tile alignment and margin constraints for the  
topLeft child node  
TilePane.clearConstraints(topLeft);
```

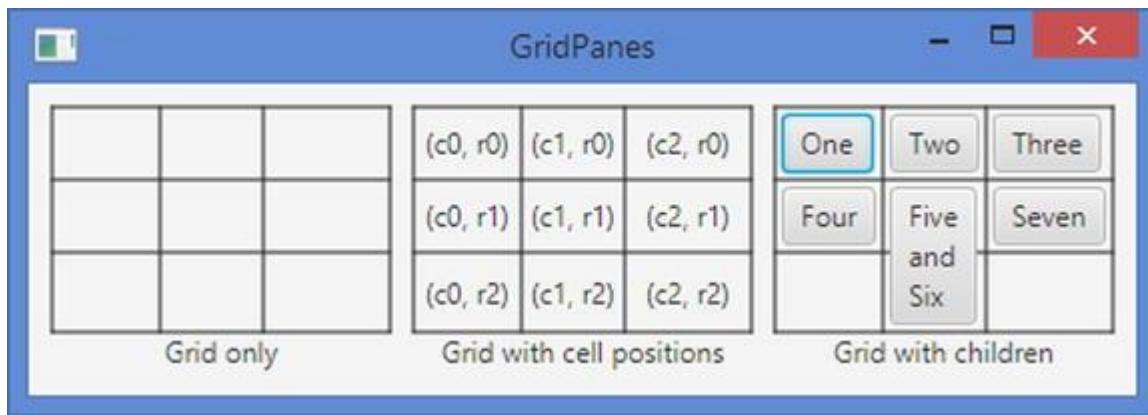
After you clear all constraints for a child node, it will use the current value of the `tileAlignment` property of the `TilePane` as its alignment and `opx` as the margins.

## Understanding GridPane

`GridPane` is one of the most powerful layout panes. With power comes complexity. Therefore, it is also a bit complex to learn.

A `GridPane` lays out its children in a dynamic grid of cells arranged in rows and columns. The grid is dynamic because the number and size of cells in the grid are determined based on the number of children. They depend on the constraints set on children. Each cell in the grid is identified by its position in the column and row. The indexes for columns and rows start at 0. A child node may be placed anywhere in the grid spanning more than one cell. All cells in a row are of the same height. Cells in different rows may have different heights. All cells in a column are of the same width. Cells in different columns may have different widths. By default, a row is tall enough to accommodate the tallest child node in it. A column is wide enough to accommodate the widest child node in it. You can customize the size of each row and column. `GridPane` also allows for vertical spacing between rows and horizontal spacing between columns.

`GridPane` does not show the grid lines by default. For debug purposes, you can show the grid lines. Figure 10-41 shows three instances of the `GridPane`. The first `GridPane` shows only the grid lines and no child nodes. The second `GridPane` shows the cell positions, which are identified by row and column indexes. In the figure, (cM, rN) means the cell at the (M+1)<sup>th</sup> column and the (N+1)<sup>th</sup> row. For example, (c3, r2) means the cell at the 4th column and the 3rd row. The third `GridPane` shows six buttons in the grid. Five of the buttons spans one row and one column; one of them spans two rows and one column.



**Figure 10-41.** GridPanes with grid only, with cell positions, and with children placed in the grid

In a `GridPane`, rows are indexed from top to bottom. The top row has an index of 0. Columns are indexed from left to right or from right to left. If the `nodeOrientation` property for the `GridPane` is set to `LEFT_TO_RIGHT`, the leftmost column has index 0. If it is set to `RIGHT_TO_LEFT`, the rightmost column has an index of 0. The second grid in Figure 10-41 shows the leftmost column having an index of 0, which means that its `nodeOrientation` property is set from `LEFT_TO_RIGHT`.

**Tip** A question that is often asked about the `GridPane` is, “How many cells, and of what sizes, do we need to lay out children in a `GridPane`?” The answer is simple but sometimes perplexing to beginners. You specify the cell positions and cell spans for the children. `GridPane` will figure out the number of cells (rows and columns) and their sizes for you. That is, `GridPane` computes the number of cells and their sizes based on the constraints that you set for the children.

## Creating GridPane Objects

The `GridPane` class contains a no-args constructor. It creates an empty `GridPane` with `opx` spacing between rows and columns, placing the children, which need to be added later, at the top-left corner within its content area.

```
GridPane gpane = new GridPane();
```

## Making Grid Lines Visible

The `GridPane` class contains a `gridLinesVisible` property of the `BooleanProperty` type. It controls the visibility of the grid lines. By default, it is set to false and the grid lines are invisible. It exists for

debugging purposes only. Use it when you want to see the positions of children in the grid.

```
GridPane gpane = new GridPane();
gpane.setGridLinesVisible(true); // Make grid lines visible
```

## Adding Children to GridPane

Like most of the other layout panes, a GridPane stores its children in an `ObservableList<Node>` whose reference is returned by the `getChildren()` method. You should not add children to the GridPane directly to the list. Rather, you should use one of the convenience methods to add children to the GridPane. You should specify constraints for children when you add them to a GridPane. The minimum constraints would be the column and row indexes to identify the cell in which they are placed.

Let us first see the effect of adding the children directly to the observable list of the GridPane. Listing 10-28 contains the program that directly adds three buttons to the list of children of a GridPane. Figure 10-42 shows the window. Notice that the buttons overlap. They are all placed in the same cell (co, ro). They are drawn in the order they are added to the list.

**Tip** In a GridPane, by default, all children are added in the first cell (co, ro) spanning only one column and one row, thus overlapping each other. They are drawn in the order they are added.

**Listing 10-28.** Adding Children to the List of Children for a GridPane Directly

```
// GridPaneChildrenList.java
package com.jdojo.container;

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

public class GridPaneChildrenList extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button b1 = new Button("One One One One One");
        Button b2 = new Button("Two Two Two");
        Button b3 = new Button("Three");
    }
}
```

```

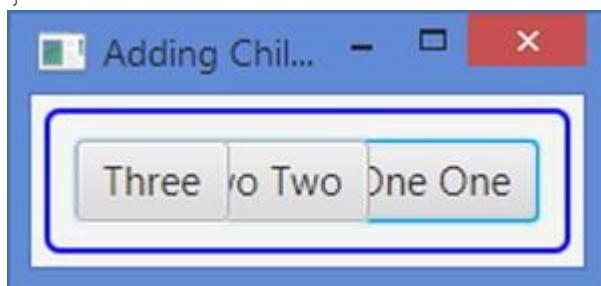
        GridPane root = new GridPane();

        // Add three buttons to the list of children of the
        GridPane directly
        root.getChildren().addAll(b1, b2, b3);

        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Adding Children to a GridPane");
        stage.show();
    }
}

```



**Figure 10-42.** Three buttons added to the list of children for a GridPane directly

There are two ways of fixing the problem of overlapping children in Listing 10-28:

- We can set the position in which they are placed, before or after adding them to the list.
- We can use convenience methods of the GridPane class that allow specifying the positions, among other constraints, while adding children to the GridPane.

## Setting Positions of Children

You can set the column and row indexes for a child node using one of the following three static methods of the GridPane class.

- public static void setColumnIndex(Node child, Integer value)
- public static void setRowIndex(Node child, Integer value)
- public static void setConstraints(Node child, int columnIndex, int rowIndex)

The program in Listing 10-29 is a modified version of the program in Listing 10-28. It adds the column and row indexes to three buttons, so they are positioned in separate columns in one row. Figure 10-43 shows the window.

### ***Listing 10-29.*** Setting Positions for Children in a GridPane

```
// GridPaneChildrenPositions.java
package com.jdojo.container;

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

public class GridPaneChildrenPositions extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button b1 = new Button("One One One One One");
        Button b2 = new Button("Two Two Two");
        Button b3 = new Button("Three");

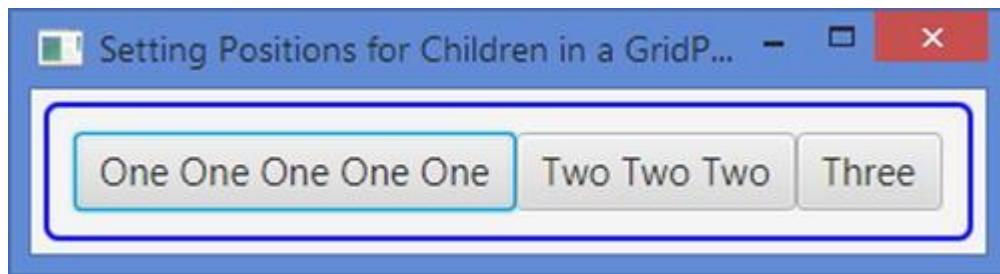
        GridPane root = new GridPane();

        // Add three buttons to the list of children of the
        // GridPane directly
        root.getChildren().addAll(b1, b2, b3);

        // Set the cells the buttons
        GridPane.setConstraints(b1, 0, 0); // (c0, r0)
        GridPane.setConstraints(b2, 1, 0); // (c1, r0)
        GridPane.setConstraints(b3, 2, 0); // (c2, r0)

        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Setting Positions for Children in
a GridPane");
        stage.show();
    }
}
```



**Figure 10-43.** Three buttons added to a `GridPane` directly and then their position set

## Using Convenience Methods to Add Children

The `GridPane` class contains the following convenience methods to add children with constraints.

- `void add(Node child, int columnIndex, int rowIndex)`
- `void add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)`
- `void addRow(int rowIndex, Node... children)`
- `void addColumn(int columnIndex, Node... children)`

The `add()` methods let you add a child node specifying the column index, row index, column span, and row span.

The `addRow()` method adds the specified `children` in a row identified by the specified `rowIndex`. Children are added sequentially. If the row already contains children, the specified `children` are appended sequentially. For example, if the `GridPane` has no children in the specified row, it will add the first child node at column index 0, the second at column index 1, etc. Suppose the specified row already has two children occupying column indexes 0 and 1. The `addRow()` method will add children starting at column index 2.

**Tip** All children added using the `addRow()` method spans only one cell. Row and column spans for a child node can be modified using the `setRowSpan(Node child, Integer value)` and `setColumnSpan(Node child, Integer value)` static methods of the `GridPane` class. When you modify the row and column spans for a child node, make sure to update row and column indexes of the affected children so they do not overlap.

The `addColumn()` method adds the specified `children` sequentially in a column identified by the specified `columnIndex`. This method adds

children to a column the same way the `addRow()` method adds children to a row.

The following snippet code creates three `GridPanes` and adds four buttons to them using three different ways. Figure 10-44 shows one of the `GridPanes`. All of them will look the same.

```
// Add a child node at a time
GridPane gpanel1 = new GridPane();
gpanel1.add(new Button("One"), 0, 0);           // (c0, r0)
gpanel1.add(new Button("Two"), 1, 0);           // (c1, r0)
gpanel1.add(new Button("Three"), 0, 1);          // (c0, r1)
gpanel1.add(new Button("Four"), 1, 1);          // (c1, r1)

// Add a row at a time
GridPane gpanel2 = new GridPane();
gpanel2.addRow(0, new Button("One"), new Button("Two"));
gpanel2.addRow(1, new Button("Three"), new Button("Four"));

// Add a column at a time
GridPane gpanel3 = new GridPane();
gpanel3.addColumn(0, new Button("One"), new Button("Three"));
gpanel3.addColumn(1, new Button("Two"), new Button("Four"));
```



**Figure 10-44.** A `GridPane` with four buttons

## Specifying Row and Column Spans

A child node may span more than one row and column, which can be specified using the `rowSpan` and `colSpan` constraints. By default, a child node spans one column and one row. These constraints can be specified while adding the child node or later using any of the following methods in the `GridPane` class.

- `void add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)`
- `static void setColumnSpan(Node child, Integer value)`
- `static void setConstraints(Node child, int columnIndex, int rowIndex, int columnspan, int rowspan)`

The `setConstraints()` method is overloaded. Other versions of the method also let you specify the column/row span.

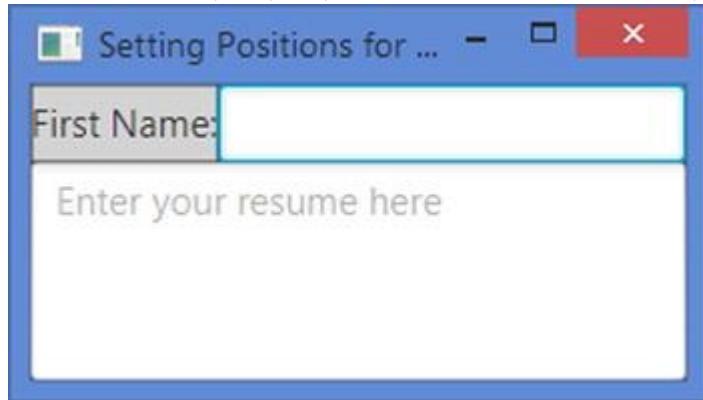
The `GridPane` class defines a constant named `REMAINING` that is used for specifying the column/row span. It means that the child node spans the remaining columns or remaining rows.

The following snippet of code adds a `Label` and a `TextField` to the first row. It adds a `TextArea` to the first column of the second row with its `colSpan` as `REMAINING`. This makes the `TextArea` occupy two columns because there are two columns created by the controls added to the first row. Figure 10-45 shows the window.

```
// Create a GridPane and set its background color to lightgray
GridPane root = new GridPane();
root.setGridLinesVisible(true);
root.setStyle("-fx-background-color: lightgray;");

// Add a Label and a TextField to the first row
root.addRow(0, new Label("First Name:"), new TextField());

// Add a TextArea in the second row to span all columns in row 2
TextArea ta = new TextArea();
ta.setPromptText("Enter your resume here");
ta.setPrefColumnCount(10);
ta.setPrefRowCount(3);
root.add(ta, 0, 1, GridPane.REMAINING, 1);
```

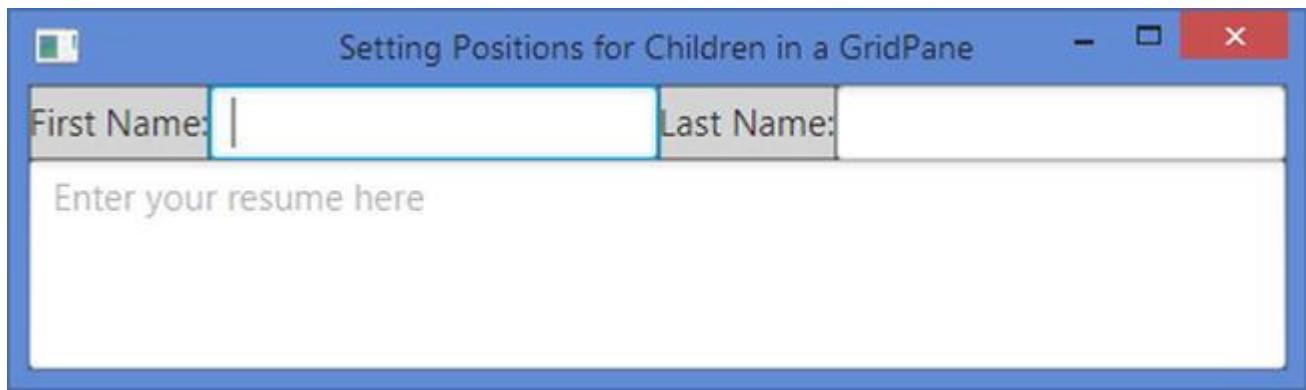


**Figure 10-45.** A `TextArea` Using `GridPane.REMAINING` as the `colSpan` value

Suppose you add two more children in the first column to occupy the third and fourth columns.

```
// Add a Label and a TextField to the first row
root.addRow(0, new Label("Last Name:"), new TextField());
```

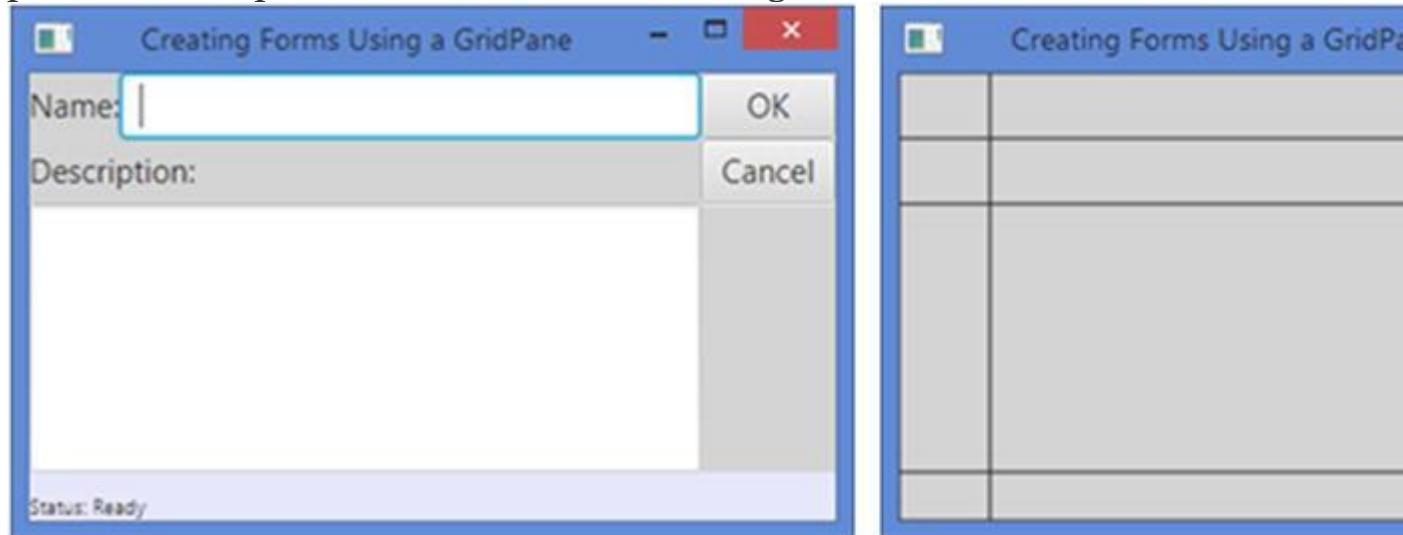
Now, the number of columns has increased from two to four. This will make the `TextArea` occupy four columns as we set its `colSpan` as `REMAINING`. Figure 10-46 shows the new window.



**Figure 10-46.** A *TextArea* using *GridPane.REMAINDER* as the *colSpan* value

### Creating Forms Using GridPanes

*GridPane* is best suited for creating forms. Let us build a form using a *GridPane*. The form will be similar to the one shown in Figure 10-32 that was created using a *BorderPane*. Our new form will look as shown in Figure 10-47. The figure shows two instances of the window: the form with children (on the left) and the form with grid only (on the right). The form with grid only is shown, so you can visualize the positions and spans of the children within the grid.



**Figure 10-47.** A *GridPane* with some controls to create a form

The grid will have three columns and four rows. It has seven children.

- A Label, a *TextField*, and an *OK* button in the first row
- A Label and a *Cancel* button in the second row
- A *TextArea* in the third row
- A Label in the fourth row

The following snippet of code creates all children.

```
// A Label and a TextField
Label nameLbl = new Label("Name:");
TextField nameFld = new TextField();

// A Label and a TextArea
Label descLbl = new Label("Description:");
TextArea descText = new TextArea();
descText.setPrefColumnCount(20);
descText.setPrefRowCount(5);

// Two buttons
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
```

All children in the first row span only one cell. The “Description” Label in the second row spans two columns (c0 and c1) and the *Cancel* button only one column. The TextArea in the third row spans two columns (c0 and c1). The Label in the fourth row spans three columns (c0, c1, and c1). The following snippet of code places all children in the grid.

```
// Create a GridPane
GridPane root = new GridPane();

// Add children to the GridPane
root.add(nameLbl, 0, 0, 1, 1);    // (c0, r0, colspan=1,
rowspan=1)
root.add(nameFld, 1, 0, 1, 1);    // (c1, r0, colspan=1,
rowspan=1)
root.add(descLbl, 0, 1, 3, 1);    // (c0, r1, colspan=3,
rowspan=1)
root.add(descText, 0, 2, 2, 1);    // (c0, r2, colspan=2,
rowspan=1)
root.add(okBtn, 2, 0, 1, 1);     // (c2, r0, colspan=1,
rowspan=1)
root.add(cancelBtn, 2, 1, 1, 1); // (c2, r1, colspan=1,
rowspan=1)

// Let the status bar start at column 0 and take up all remaining
columns
// (c0, r3, colspan=REMAINING, rowspan=1)
root.add(statusBar, 0, 3, GridPane.REMAINING, 1);
```

If we add the GridPane to a scene, it will give us the desired look of the form, but not the desired resizing behavior. The children will not resize correctly on resizing the window. We need to specify the correct resizing behavior for some of the children.

- The *OK* and *Cancel* buttons should be of the same size.
- The TextField to enter name should expand horizontally.
- The TextArea to enter the description should expand horizontally and vertically.

- The Label used as the status bar at the bottom should expand horizontally.

Making the *OK* and *Cancel* buttons the same size is easy. By default, a GridPane resizes its children to fill their cells, provided the maximum size of the children allows it. The maximum size of a Button is clamped to its preferred size. We need to set the maximum size of the *OK* button big enough, so it can expand to fill the width of its cell, which would be the same as the preferred width of the widest node in its column (the *Cancel* button).

```
// The max width of the OK button should be big enough, so it can
fill the
// width of its cell
okBtn.setMaxWidth(Double.MAX_VALUE);
```

By default, the rows and columns in a GridPane stay at their preferred size when the GridPane is resized. Their horizontal and vertical grow constraints specify how they grow when additional space is available. To let the name, description, and status bar fields grow when the GridPane is expanded, we will set their **hgrow** and **vgrow** constraints appropriately.

```
// The name field in the first row should grow horizontally
GridPane.setHgrow(nameFld, Priority.ALWAYS);

// The description field in the third row should grow vertically
GridPane.setVgrow(descText, Priority.ALWAYS);

// The status bar in the last row should fill its cell
statusBar.setMaxWidth(Double.MAX_VALUE);
```

When the GridPane is expanded horizontally, the second column, occupied by the name field, grows by taking the extra available width. It makes the description and status bar fields fill the extra width generated in the second column.

When the GridPane is expanded vertically, the third row, occupied by the description field, grows by taking the extra available height. The maximum size of a TextArea is unbounded. That is, it can grow to fill the available space in both directions. The program in Listing 10-30 contains the complete code.

### ***Listing 10-30.*** Using a GridPane to Create Forms

```
// GridPaneForm.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
```

```

import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class GridPaneForm extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // A Label and a TextField
        Label nameLbl = new Label("Name:");
        TextField nameFld = new TextField();

        // A Label and a TextArea
        Label descLbl = new Label("Description:");
        TextArea descText = new TextArea();
        descText.setPrefColumnCount(20);
        descText.setPrefRowCount(5);

        // Two buttons
        Button okBtn = new Button("OK");
        Button cancelBtn = new Button("Cancel");

        // A Label used as a status bar
        Label statusBar = new Label("Status: Ready");
        statusBar.setStyle("-fx-background-color: lavender;"

+
                    "-fx-font-size: 7pt;" +
                    "-fx-padding: 10 0 0 0;");

        // Create a GridPane and set its background color to
        lightgray
        GridPane root = new GridPane();
        root.setStyle("-fx-background-color: lightgray;");

        // Add children to the GridPane
        root.add(nameLbl, 0, 0, 1, 1);    // (c0, r0,
        colspan=1, rowspan=1)
        root.add(nameFld, 1, 0, 1, 1);    // (c1, r0,
        colspan=1, rowspan=1)
        root.add(descLbl, 0, 1, 3, 1);    // (c0, r1,
        colspan=3, rowspan=1)
        root.add(descText, 0, 2, 2, 1);    // (c0, r2,
        colspan=2, rowspan=1)
        root.add(okBtn, 2, 0, 1, 1);     // (c2, r0,
        colspan=1, rowspan=1)
        root.add(cancelBtn, 2, 1, 1, 1); // (c2, r1,
        colspan=1, rowspan=1)
        root.add(statusBar, 0, 3, GridPane.REMAINING, 1);
    }
}

```

```

        /* Set constraints for children to customize their
resizing behavior */

        // The max width of the OK button should be big
enough,
        // so it can fill the width of its cell
okBtn.setMaxWidth(Double.MAX_VALUE);

        // The name field in the first row should grow
horizontally
        GridPane.setHgrow(nameFld, Priority.ALWAYS);

        // The description field in the third row should
grow vertically
        GridPane.setVgrow(descText, Priority.ALWAYS);

        // The status bar in the last should fill its cell
statusBar.setMaxWidth(Double.MAX_VALUE);

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Creating Forms Using a GridPane");
stage.show();
}

}

```

## GridPane Properties

The `GridPane` class contains several properties, as listed in Table 10-6, to customize its layout.

**Table 10-6.** The List of Properties Declared in the `GridPane` Class

Property	Type	Description
alignment	ObjectProperty<Pos>	It specifies the alignment of the grid of the <code>GridPane</code> ) relative to its center. It defaults to <code>Pos.TOP_LEFT</code> .
gridLinesVisible	BooleanProperty	It is recommended to be used for debugging only. It controls whether grid lines are visible or not. It defaults to false.
hgap, vgap	DoubleProperty	They specify the gaps between adjacent columns and rows. The <code>hgap</code> property specifies the horizontal gap between adjacent columns. The <code>vgap</code> property specifies the vertical gap between adjacent rows.

Property	Type	Description
		vertical gap between adjacent rows. default to zero.

## The Alignment Property

The alignment property of a GridPane controls the alignment of its content within its content area. You can see the effects of this property when the size of the GridPane is bigger than its content. The property works the same way as the alignment property for the FlowPane. Please refer to the description of the alignment property for FlowPane for more details and illustrations.

## The gridLinesVisible Property

When the `gridLinesVisible` is set to true, the grid lines in a GridPane are made visible. Otherwise, they are invisible. You should use this feature only for debug purposes only.

```
GridPane gpane = new GridPane();
gpane.setGridLinesVisible(true); // Make grid lines visible
```

Sometimes, you may want to show the grid without showing the children to get an idea on how the grid is formed. You can do so by making all children invisible. The GridPane computes the size of the grid for all managed children irrespective of their visibility.

The following snippet of code creates a GridPane and sets the `gridLinesVisible` property to true. It creates four Buttons, makes them invisible, and adds them to the GridPane. Figure 10-48 shows the window when the GridPane is added to a scene as the root node.

```
GridPane root = new GridPane();

// Make the grid lines visible
root.setGridLinesVisible(true);

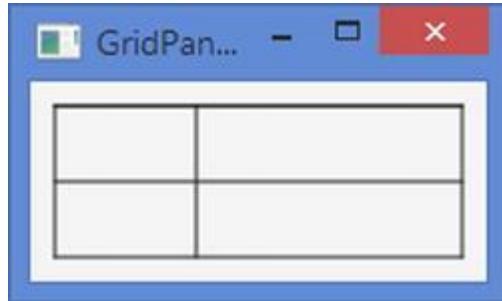
// Set the padding to 10px
root.setStyle("-fx-padding: 10;");

// Make the gridLInes
Button b1 = new Button("One");
Button b2 = new Button("Two");
Button b3 = new Button("Three");
Button b4 = new Button("Four and Five");

// Make all children invisible to see only grid lines
b1.setVisible(false);
```

```
b2.setVisible(false);
b3.setVisible(false);
b4.setVisible(false);

// Add children to the GridPane
root.addRow(1, b1, b2);
root.addRow(2, b3, b4);
```



**Figure 10-48.** A GridPane showing the grid without children

### The hgap and vgap Properties

You can specify spacing between adjacent columns and rows using the `hgap` and `vgap` properties, respectively. By default, they are zero. The program in Listing 10-31 uses these properties of a `GridPane`. The grid lines are visible to show the gaps clearly. Figure 10-49 shows the window.

### **Listing 10-31.** Using the hgap and vgap Properties of a GridPane

```
// GridPaneHgapVgap.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class GridPaneHgapVgap extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label fnameLbl = new Label("First Name:");
        TextField fnameFld = new TextField();
        Label lnameLbl = new Label("Last Name:");
        TextField lnameFld = new TextField();
        Button okBtn = new Button("OK");
    }
}
```

```

        Button cancelBtn = new Button("Cancel");

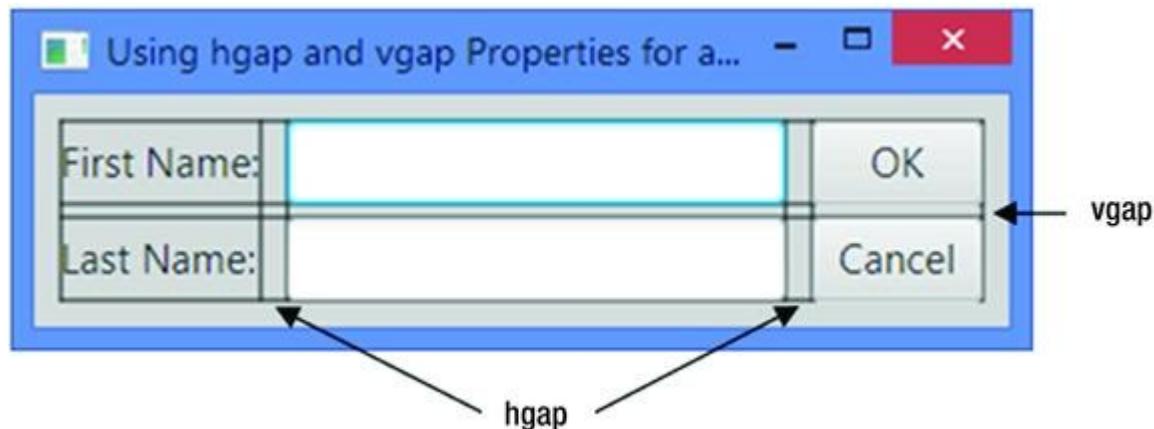
        // The Ok button should fill its cell
        okBtn.setMaxWidth(Double.MAX_VALUE);

        // Create a GridPane and set its background color to
        lightgray
        GridPane root = new GridPane();
        root.setGridLinesVisible(true); // Make grid lines
        visible
        root.setHgap(10); // hgap = 10px
        root.setVgap(5); // vgap = 5px
        root.setStyle("-fx-padding: 10;-fx-background-color:
        lightgray;");

        // Add children to the GridPane
        root.addRow(0, fnameLbl, fnameFld, okBtn);
        root.addRow(1, lnameLbl, lnameFld, cancelBtn);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using hgap and vgap Properties for
        a GridPane");
        stage.show();
    }
}

```



**Figure 10-49.** A GridPane using hgap and vgap properties

### Customizing Columns and Rows

You can customize columns and rows in a GridPane using column and row constraints. For example, for a column/row, you can specify:

- How the width/height should be computed. Should it be computed based on its content, a fixed width/height, or a percentage of the available width/height?
- Should the children fill the width/height of the column/row?

- Should the column/row grow when the `GridPane` is resized larger than its preferred width/height?
- How should the children in a column/row be aligned within its layout area (cells)?

An object of the `ColumnConstraints` class represents constraints for a column and an object of the `RowConstraints` class represents constraints for a row. Both classes declare several properties that represent the constraints. Tables 10-7 and 10-8 list the properties with a brief description for the `ColumnConstraints` and `RowConstraints` classes.

**Table 10-7.** The List of Properties for the `ColumnConstraints` Class

Property	Type	Description
<code>fillWidth</code>	<code>BooleanProperty</code>	It specifies whether the children in the column are expanded beyond their preferred width to fill the width of the column. The default value is true.
<code>halignment</code>	<code>ObjectProperty&lt;HPos&gt;</code>	It specifies the default horizontal alignment for the children in a column. Its default value is <code>null</code> . By default, all children in the column are horizontally aligned to <code>HPos.CENTER</code> . An individual child node in the column can override this constraint.
<code>hgrow</code>	<code>ObjectProperty&lt;Priority&gt;</code>	It specifies the horizontal grow priority for the column. This property is used to give additional space to the column when the <code>GridPane</code> is resized larger than its preferred width. If the <code>percentWidth</code> property is set for this property is ignored.
<code>minWidth</code> , <code>prefWidth</code> , <code>maxWidth</code>	<code>DoubleProperty</code>	They specify the minimum, preferred, and maximum widths of the column. If the <code>percentWidth</code> property is set for these properties are ignored.

Property	Type	Description
		The default values for these properties are set to USE_COMPUTED_SIZE. By default, the minimum width of a column is the largest of the minimum widths of children in the column. The preferred width is the largest of the preferred widths of children in the column, and the maximum width is the sum of the maximum widths of children in the column.
percentWidth	DoubleProperty	It specifies the width percentage of the column relative to the width of the content area of the GridPane. If it is set to a value greater than zero, the column is resized to a width that is this percentage of the total width of the GridPane. If this property is set, the minWidth, prefWidth, maxWidth, and hgrow properties are ignored.

**Table 10-8.** Properties for the RowConstraints Class

Property	Type	Description
fillHeight	BooleanProperty	It specifies whether the children in the row are expanded beyond their preferred height to the height of the row. The default value is false.
valignment	ObjectProperty<HPos>	It specifies the default vertical alignment for all children in a row. Its default value is VPos.CENTER. By default, all children in a row are aligned to VPos.CENTER. An individual node in the row may override this setting.
vgrow	ObjectProperty<Priority>	It specifies the vertical grow priority for the row. This property is used to give extra space to the row when the GridPane's height is larger than its preferred height. It has no effect if the percentHeight property is set.

Property	Type	Description
minHeight, prefHeight, maxHeight	DoubleProperty	for this property is ignored.
percentHeight	DoubleProperty	<p>They specify the minimum, preferred and maximum heights of the row. If the percentHeightproperty is set, the default values for these properties are ignored.</p> <p>The default values for these properties are set to USE_COMPUTED_SIZE. By default, the minimum height of a row is the largest of the minimum heights of children in the row; the preferred height is the largest of the preferred heights of children in the row; and the maximum height is the smallest of the maximum heights of children in the row.</p>

The ColumnConstraints and RowConstraints classes provide several constructors to create their objects. Their no-args constructors create their objects with default property values.

```
// Create a ColumnConstraints object with default property values
ColumnConstraints cc1 = new ColumnConstraints();

// Set the percentWidth to 30% and horizontal alignment to center
cc1.setPercentWidth(30);
cc1.setAlignment(HPos.CENTER);
```

If you want to create a fixed width/height column/row, you can use one of the convenience contractors.

```
// Create a ColumnConstraints object with a fixed column width of
100px
ColumnConstraints cc2 = new ColumnConstraints(100);

// Create a RowConstraints object with a fixed row height of 80px
RowConstraints rc2 = new RowConstraints(80);
```

If you want to achieve the same effect of having a fixed width column, you can do so by setting the preferred width to the desired fixed width value and setting the minimum and maximum widths to use the preferred width as shown below.

```
// Create a ColumnConstraints object with a fixed column width of
100px
ColumnConstraints cc3 = new ColumnConstraints();
cc3.setPrefWidth(100);
cc3.setMinWidth(Region.USE_PREF_SIZE);
cc3.setMaxWidth(Region.USE_PREF_SIZE);
```

The following snippet of code sets the column width to 30% of the GridPane width and the horizontal alignment for the children in the column as center.

```
ColumnConstraints cc4 = new ColumnConstraints();
cc4.setPercentWidth(30); // 30% width
cc4.setAlignment(HPos.CENTER);
```

In a GridPane, the width/height of different columns/rows may be computed differently. Some columns/row may set percent width/height, some fixed sizes, and some may choose to compute their sizes based on their content. The percent size is given the first preference in allocating the space. For example, if two columns set their widths based on percentage and one uses a fixed width, the available width will be allocated first to the two columns using the percentage width, and then, to the column using the fixed width.

**Tip** It is possible that the sum of the percentage width/height of all columns/rows exceeds 100. For example, it is permissible to set the percentage width of columns in a GridPane to 30%, 30%, 30%, and 30%. In this case, the percentage value is used as weights and each of the four columns will be given one-fourth ( $30/120$ ) of the available width. As an another example, if columns use 30%, 30%, 60%, and 60% as percentage width, they will be treated as weights, allocating them one-sixth ( $30/180$ ), one-sixth ( $30/180$ ), one-third ( $60/180$ ), and one-third ( $60/180$ ) of the available width, respectively.

A GridPane stores the constraints for columns and rows in ObservableList of ColumnConstraints and RowConstraints. You can obtain the reference of the lists using the getColumnConstraints() and getRowConstraints() methods. The element at a particular index in the list stores the constraints object for the column/row at the same index in the GridPane. The first element in the list, for example, stores the column/row constraints for the first column/row, the second elements for the second column/row, etc. It is possible to set the column/row constraints for some column/row, not for others. In this case, the constraints for column/row

for which the column/row constraints are absent will be computed based on the default values. The following snippet of code creates three `ColumnConstraints` objects, sets their properties, and adds them to the list of column constraints of a `GridPane`.

Using `RowConstraints` objects for setting row constraints would use the similar logic.

```
// Set the fixed width to 100px
ColumnConstraints cc1 = new ColumnConstraints(100);

// Set the percent width to 30% and horizontal alignment to center
ColumnConstraints cc2 = new ColumnConstraints();
cc2.setPercentWidth(30);
cc1.setAlignment(HPos.CENTER);

// Set the percent width to 50%
ColumnConstraints cc3 = new ColumnConstraints();
cc3.setPercentWidth(30);

// Add all column constraints to the column constraints list
GridPane root = new GridPane();
root.getColumnConstraints().addAll(cc1, cc2, cc3);
```

The program in Listing 10-32 uses column and row constraints to customize columns and rows in a `GridPane`. Figure 10-50 shows the window, after it is resized.

### ***Listing 10-32.*** Using Column and Row Constraints in a `GridPane`

```
// GridPaneColRowConstraints.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.RowConstraints;
import javafx.stage.Stage;

public class GridPaneColRowConstraints extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        root.setStyle("-fx-padding: 10;");
        root.setGridLinesVisible(true);
```

```

        // Add children
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                Button b = new Button(col + " " + row);
                root.add(b, col, row);
            }
        }

        // Set the fixed width for the first column to 100px
        ColumnConstraints cc1 = new ColumnConstraints(100);

        // Set the percent width for the second column to
30% and
        // the horizontal alignment to center
        ColumnConstraints cc2 = new ColumnConstraints();
        cc2.setPercentWidth(35);
        cc2.setAlignment(HPos.CENTER);

        // Set the percent width for the third column to 50%
        ColumnConstraints cc3 = new ColumnConstraints();
        cc3.setPercentWidth(35);

        // Add all column constraints to the column
constraints list
        root.getColumnConstraints().addAll(cc1, cc2, cc3);

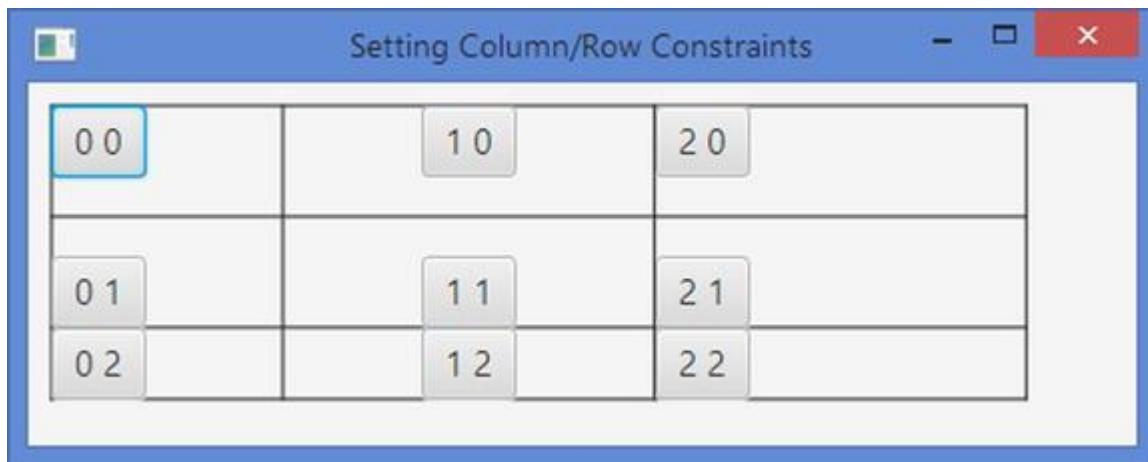
        // Create two RowConstraints objects
        RowConstraints rc1 = new RowConstraints();
        rc1.setPercentHeight(35);
        rc1.setAlignment(VPos.TOP);

        RowConstraints rc2 = new RowConstraints();
        rc2.setPercentHeight(35);
        rc2.setAlignment(VPos.BOTTOM);

        // Add RowConstraints for the first two rows
        root.getRowConstraints().addAll(rc1, rc2);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Setting Column/Row Constraints");
        stage.show();
    }
}

```



**Figure 10-50.** A GridPane using column and row constraints

The first column width is set to 100px fixed width. Each of the second and third columns is set to occupy 35% of the width. If the needed width ( $35\% + 35\% + 100\text{px}$ ) is less than the available width, the extra width will be left unused, as has been shown in the figure. The horizontal alignment for the first column is set to center, so all buttons in the first column are horizontally aligned in the center. The buttons in the other two columns use left as the horizontal alignment, which is the default setting. We have three rows. However, the program adds constraints for only the first two rows. The constraints for the third row will be computed based on its content.

When you set column/row constraints, you cannot skip some columns/rows in the middle. That is, you must set the constraints for columns/rows sequentially starting from the first column/row.

Setting null for a constraint's object throws a `NullPointerException` at runtime. If you want to skip setting custom constraints for a row/column in the list, set it to a constraints object that is created using the no-args constructor, which will use the default settings. The following snippet of code sets the column constraints for the first three columns. The second column uses default settings for the constraints.

```
// With 100px fied width
ColumnConstraints cc1 = new ColumnConstraints(100);

// Use all default settings
ColumnConstraints defaultCc2 = new ColumnConstraints();

// With 200px fied width
ColumnConstraints cc3 = new ColumnConstraints(200);

GridPane gpane = new GridPane();
gpane.getColumnConstraints().addAll(cc1, defaultCc2, cc3);
```

**Tip** Some column/row constraints set on a column/row can be overridden by children in the column/row individually. Some constraints can be set on children in a column/row and may affect the entire column/row. We will discuss these situations in the next section.

### Setting Constraints on Children in GridPane

Table 10-9 lists the constraints that can be set for the children in a GridPane. We have already discussed the column/row index and span constraints. We will discuss the rest in this section. The GridPane class contains two sets of static methods to set these constraints:

- The `setConstraints()` methods
- The `setXxx(Node child, CType cvalue)` methods, where `Xxx` is the constraint name and `CType` is its type

To remove a constraint for a child node, set it to null

**Table 10-9.** List of Constraints That Can Be Set for the Children in a GridPane

Constraint	Type	Description
<code>columnIndex</code>	<code>Integer</code>	It is the column index where the layout area of the child node starts. The first column has the index 0. The default value is 0.
<code>rowIndex</code>	<code>Integer</code>	It is the row index where the layout area of the child node starts. The first row has the index 0. The default value is 0.
<code>columnSpan</code>	<code>Integer</code>	It is the number of columns the layout area of a child node spans. The default is 1.
<code>rowSpan</code>	<code>Integer</code>	It is the number of rows the layout area of a child node spans. The default is 1.
<code>halignment</code>	<code>HPos</code>	It specifies the horizontal alignment of the child node within its layout area.
<code>valignment</code>	<code>VPos</code>	It specifies the vertical alignment of the child node within its layout area.

Constraint	Type	Description
hgrow	Priority	It specifies the horizontal grow priority of the child node.
vgrow	Priority	It specifies the vertical grow priority of the child node.
margin	Insets	It specifies the margin space around the outside of the layout of the child node.

## The halignment and valignment Constraints

The `halignment` and `valignment` constraints specify the alignment of a child node within its layout area. They default to `HPos.LEFT` and `VPos.CENTER`. They can be set on column/row affecting all children. Children may set them individually. The final value applicable to a child node depends of some rules:

- When they are not set for column/row and not for the child node, the child node will use the default values.
- When they are set for column/row and not for the child node, the child node will use the value set for the column/row.
- When they are set for column/row and for the child node, the child node will use the value set for it, not the value set for the column/row. In essence, a child node can override the default value or the value set for the column/row for these constraints.

The program in Listing 10-33 demonstrates the rules mentioned above. Figure 10-51 shows the window. The program adds three buttons to a column. The column constraints override the default value of `HPos.LEFT` for the `halignment` constraints for the children and set it to `HPos.RIGHT`. The button labeled “Two” overrides this setting to `HPos.CENTER`. Therefore, all buttons in the column are horizontally aligned to the right, except the button labeled “Two,” which is aligned to the center. We set constraints for all three rows. The first and the second rows set `valignment` to `VPos.TOP`. The third row leaves the `valignment` to the default that is `VPos.CENTER`. The button with the label “One” overrides the `valignment` constraint set on the first row to set it to `VPos.BOTTOM`. Notice that all children follow the above thee rules to use the `valignment` and `halignment` constraints.

**Listing 10-33.** Using the halignment and valignment Constraints for Children in a GridPane

```
// GridPaneHValignment.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.RowConstraints;
import javafx.stage.Stage;

public class GridPaneHValignment extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        root.setStyle("-fx-padding: 10;");
        root.setGridLinesVisible(true);

        // Add three buttons to a column
        Button b1 = new Button("One");
        Button b2 = new Button("Two");
        Button b3 = new Button("Three");
        root.addColumn(0, b1, b2, b3);

        // Set the column constraints
        ColumnConstraints cc1 = new ColumnConstraints(100);
        cc1.setAlignment(HPos.RIGHT);
        root.getColumnConstraints().add(cc1);

        // Set the row constraints
        RowConstraints rc1 = new RowConstraints(40);
        rc1.setAlignment(VPos.TOP);

        RowConstraints rc2 = new RowConstraints(40);
        rc2.setAlignment(VPos.TOP);

        RowConstraints rc3 = new RowConstraints(40);
        root.getRowConstraints().addAll(rc1, rc2, rc3);

        // Override the halignment for b2 set in the column
        GridPane.setAlignment(b2, HPos.CENTER);

        // Override the valignment for b1 set in the row
        GridPane.setAlignment(b1, VPos.BOTTOM);

        Scene scene = new Scene(root);
```

```

        stage.setScene(scene);
        stage.setTitle("halignemnt and valignment
Constraints");
        stage.show();
    }
}

```



**Figure 10-51.** Children overriding the halignment and valignment constraints in a GridPane

## The hgrow and vgrow Constraints

The `hgrow` and `vgrow` constraints specify the horizontal and vertical grow priorities for the entire column and row, even though it can be set for children individually. These constraints can also be set using the `ColumnConstraints` and `RowConstraints` objects for columns and rows. By default, columns and rows do not grow. The final value for these constraints for a column/row is computed using the following rules:

- If the constraints are not set for the column/row and are not set for any children in the column/row, the column/row does not grow if the `GridPane` is resized to a larger width/height than the preferred width/height.
- If the constraints are set for the column/row, the values set in the `ColumnConstraints` and `RowConstraints` objects for `hgrow` and `vgrow` are used, irrespective of whether the children set these constraints or not.
- If the constraints are not set for the column/row, the maximum values for these constraints set for children in the column/row are used for the entire column/row. Suppose a column has three children and no column constraints have been set for the column. The first child node sets the `hgrow` to `Priority.NEVER`; the second to `Priority.ALWAYS`; and the third to `Priority.SOMETIMES`. In this case, the maximum of the

three priorities would be `Priority.ALWAYS`, which will be used for the entire column. The `ALWAYS` priority has the highest value, `SOMETIMES` the second highest, and `NEVER` the lowest.

- If a column/row is set to have a fixed or percentage width/height, the `hgrow/vgrow` constraints will be ignored.

The program in Listing 10-34 demonstrates the above rules. Figure 10-52 shows the window when it is expanded horizontally. Notice that the second column grows, but not the first column. The program adds six buttons arranged in two columns. The first column sets the `hgrow` constraints to `Priority.NEVER`. The `hgrow` value set by the column takes priority; the first column does not grow when the `GridPane` is expanded horizontally. The second column does not use column constraints. The children in this column use three different types of priorities: `ALWAYS`, `NEVER`, and `SOMETIMES`. The maximum of the three priorities is `ALWAYS`, which makes the second column grow horizontally.

***Listing 10-34.*** Using the `hgrow` Constraints for Columns and Rows in a `GridPane`

```
// GridPaneHVgrow.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class GridPaneHVgrow extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        root.setStyle("-fx-padding: 10;");
        root.setGridLinesVisible(true);

        // Add three buttons to a column
        Button b1 = new Button("One");
        Button b2 = new Button("Two");
        Button b3 = new Button("Three");
        Button b4 = new Button("Four");

        ColumnConstraints c1 = new ColumnConstraints();
        c1.setHgrow(Priority.NEVER);
        ColumnConstraints c2 = new ColumnConstraints();
        c2.setHgrow(Priority.ALWAYS);

        root.getColumnConstraints().addAll(c1, c2);
        root.add(b1, 0, 0);
        root.add(b2, 0, 1);
        root.add(b3, 0, 2);
        root.add(b4, 1, 0);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

```

        Button b5 = new Button("Five");
        Button b6 = new Button("Six");

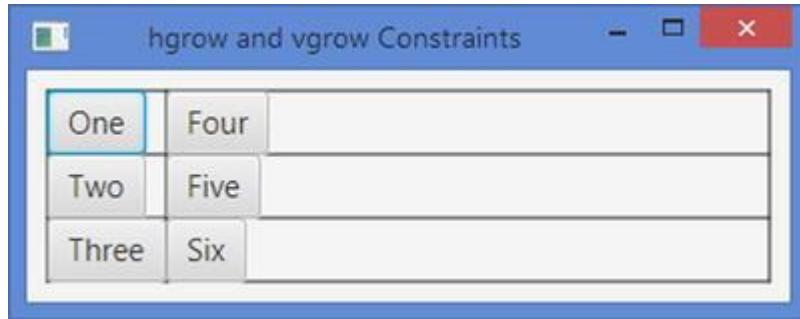
        root.addColumn(0, b1, b2, b3);
        root.addColumn(1, b4, b5, b6);

        // Set the column constraints
        ColumnConstraints cc1 = new ColumnConstraints();
        cc1.setHgrow(Priority.NEVER);
        root.getColumnConstraints().add(cc1);

        // Set three different hgrow priorities for children
        in the second
        // column. The highest priority, ALWAYS, will be
        used.
        GridPane.setHgrow(b4, Priority.ALWAYS);
        GridPane.setHgrow(b5, Priority.NEVER);
        GridPane.setHgrow(b6, Priority.SOMETIMES);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("hgrow and vgrow Constraints");
        stage.show();
    }
}

```



**Figure 10-52.** Columns and children using the hgrow constraint in a GridPane

## The Margin Constraints

Use the `setMargin(Node child, Insets value)` static method of the `GridPane` class to set the margin (the space around the layout bounds) for children. The `getMargin(Node child)` static method returns the margin for a child node.

```

// Set 10px margin around the b1 child node
GridPane.setMargin(b1, new Insets(10));
...
// Get the margin of the b1 child node
Insets margin = GridPane.getMargin(b1);

```

Use `null` to reset the margin to the default value, which is zero.

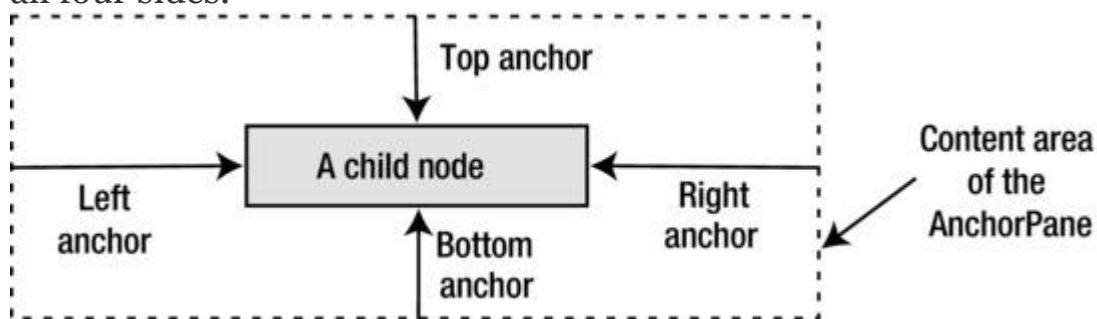
## Clearing All Constraints

Use the `clearConstraints(Node child)` static method of the `GridPane` class to reset all constraints (`columnIndex`, `rowIndex`, `columnSpan`, `rowSpan`, `halignment`, `valignment`, `hgrow`, `vgrow`, `margin`) for a child at once.

```
// Clear all constraints for the b1 child node
GridPane.clearConstraints(b1);
```

## Understanding AnchorPane

An `AnchorPane` lays out its children by anchoring the four edges of its children to its own four edges at a specified distance. Figure 10-53 shows a child node inside an `AnchorPane` with an anchor distance specified on all four sides.



**Figure 10-53.** The four side constraints for a child node in an `AnchorPane`

An `AnchorPane` may be used for two purposes:

- For aligning children along one or more edges of the `AnchorPane`
- For stretching children when the `AnchorPane` is resized

The specified distance between the edges of the children and the edges of the `AnchorPane` is called the *anchor* constraint for the sides it is specified. For example, the distance between the top edge of the children and the top edge of the `AnchorPane` is called *topAnchor constraint*, etc. You can specify at most four anchor constraints for a child node: `topAnchor`, `rightAnchor`, `bottomAnchor`, and `leftAnchor`. When you anchor a child node to the two opposite edges (top/bottom or left/right), the children are resized to maintain the specified anchor distance as the `AnchorPane` is resized.

**Tip** Anchor distance is measured from the edges of the content area of the `AnchorPane` and the edges of the children. That is, if the `AnchorPane` has a border and padding, the distance is measured from the inner edges the insets (border + padding).

## Creating AnchorPane Objects

You can create an empty AnchorPane using the no-args constructor:

```
AnchorPane apanel = new AnchorPane();
```

You can also specify the initial list of children for the AnchorPane when you create it, like so:

```
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
AnchorPane apane2 = new AnchorPane(okBtn, cancelBtn);
```

You can add children to an AnchorPane after you create it, like so:

```
Button okBtn = new Button("OK");
Button cancelBtn = new Button("Cancel");
AnchorPane apane3 = new AnchorPane();
apane3.getChildren().addAll(okBtn, cancelBtn);
```

You need to keep two points in mind while working with an AnchorPane:

- By default, an AnchorPane places its children at (0, 0). You need to specify anchor constraints for the children to anchor them to one or more edges of the AnchorPane at a specified distance.
- The preferred size of the AnchorPane is computed based on the children preferred sizes and their anchor constraints. It adds the preferred width, left anchor, and right anchor for each child node. The child having maximum of this value determines the preferred width of the AnchorPane. It adds the preferred height, left anchor, and right anchor for each child node. The child having the maximum of this value determines the preferred height of the AnchorPane. It is possible that children will overlap. Children are drawn in the order they are added.

The program in Listing 10-35 adds two buttons to an AnchorPane. One button has a long label and another has a short label. The button with the long label is added first, and hence, it is drawn first. The second button is drawn second, which overlays the first button as shown in Figure 10-54. The figure shows two views of the window: one when the program is run and another when the window is resized. Both buttons are placed at (0, 0). This program does not take advantage of the anchoring features of the AnchorPane.

### ***Listing 10-35.*** Using Default Positions in an AnchorPane

```
// AnchorPaneDefaults.java
package com.jdojo.container;

import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

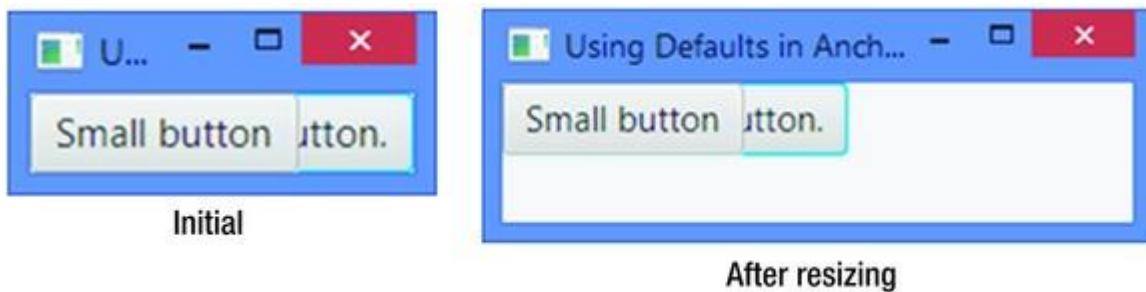
public class AnchorPaneDefaults extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button bigBtn = new Button("This is a big button.");
        Button smallBtn = new Button("Small button");

        // Create an AnchorPane with two buttons
        AnchorPane root = new AnchorPane(bigBtn, smallBtn);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Defaults in AnchorPane");
        stage.show();
    }
}

```



**Figure 10-54.** An AnchorPane with two Buttons without having anchor constraints specified

### Setting Constraints for Children in AnchorPane

Table 10-10 lists the constraints that can be set for the children in a GridPane. Note that the anchor distance is measured from the edges of the content area of the AnchorPane, not the edges of the layout bounds. Recall that a Region has padding and border insets between the edges of the content area and the layout bounds.

**Table 10-10.** The List of Constraints That Can Be Set for the Children in a GridPane

Constraint	Type	Description
topAnchor	Double	It specifies the distance between the top edge of the content area and the top edge of the anchor.

Constraint	Type	Description
		the AnchorPane and the top edge of the child node.
rightAnchor	Double	It specifies the distance between the right edge of the content and the AnchorPane and the right edge of the child node.
bottomAnchor	Double	It specifies the distance between the bottom edge of the content and the AnchorPane and the bottom edge of the child node.
leftAnchor	Double	It specifies the distance between the left edge of the content and the AnchorPane and the left edge of the child node.

The AnchorPane class contains four static methods that let you set the values for the four anchor constraints. To remove a constraint for a child node, set it to null

```
// Create a Button and anchor it to top and left edges at 10px
from each
Button topLeft = new Button("Top Left");
AnchorPane.setTopAnchor(topLeft, 10.0); // 10px from the top
edge
AnchorPane.setLeftAnchor(topLeft, 10.0); // 10px from the left
edge
```

```
AnchorPane root = new AnchorPane(topLeft);
```

Use the `clearConstraints(Node child)` static method to clear the values for all four anchor constraints for a child node.

The `setXxxAnchor(Node child, Double value)` method takes a Double value as its second parameters. Therefore, you must pass a double value or a Double object to these methods. When you pass a double value, the autoboxing feature of Java will box the value into a Double object for you. A common mistake is to pass an int value:

```
Button b1 = new Button("A button");
AnchorPane.setTopAnchor(b1, 10); // An error: 10 is an int, not
a double
```

The above code generates an error:

```
Error(18): error: method setTopAnchor in class AnchorPane cannot
be applied to given types;
```

The error is generated because we have passed 10 as the second argument. The value 10 is an int literal, which is boxed to an Integer object, not a Double object. Changing 10 to 10D or 10.0 will make it a double value and will fix the error.

The program in Listing 10-36 adds two Buttons to an AnchorPane. The first button has its top and left anchors set. The second button has its bottom and right anchors set. Figure 10-55 shows the window in two states: one when the program is run and another when the window is resized. The initial size of the window is not wide enough to display both buttons, so the buttons overlap. The JavaFX runtime computes the width of the content area of the window based on the preferred size of the bottom-right button, which has the maximum preferred width, and its right anchor value. The figure also shows the window after it is resized. You need to set a sensible preferred size for an AnchorPane, so all children are visible without overlapping.

### ***Listing 10-36.*** Using an AnchorPane to Align Children to Its Corners

```
// AnchorPaneTest.java
package com.jdojo.container;

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

public class AnchorPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Button topLeft = new Button("Top Left");
        AnchorPane.setTopAnchor(topLeft, 10.0);
        AnchorPane.setLeftAnchor(topLeft, 10.0);

        Button bottomRight = new Button("Bottom Right");
        AnchorPane.setBottomAnchor(bottomRight, 10.0);
        AnchorPane.setRightAnchor(bottomRight, 10.0);

        AnchorPane root = new AnchorPane();
        root.getChildren().addAll(topLeft, bottomRight);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using an AnchorPane");
        stage.show();
    }
}
```



**Figure 10-55.** Two Buttons in an AnchorPane aligned at top-left and bottom-right corners

When a child node in an `AnchorPane` is anchored to opposite edges, for example, top/bottom or left/right, the `AnchorPane` stretches the child node to maintain the specified anchors.

The program in Listing 10-37 adds a button to an `AnchorPane` and anchors it to the left and right edges (opposite edges) using an anchor of 10px from each edge. This will make the button stretch when the `AnchorPane` is resized to a width larger than its preferred width. The button is also anchored to the top edge. Figure 10-56 shows the initial and resized windows.

### **Listing 10-37.** Anchoring Children to Opposite Sides in an AnchorPane

```
// AnchorPaneStretching.java
package com.jdojo.container;

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

public class AnchorPaneStretching extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

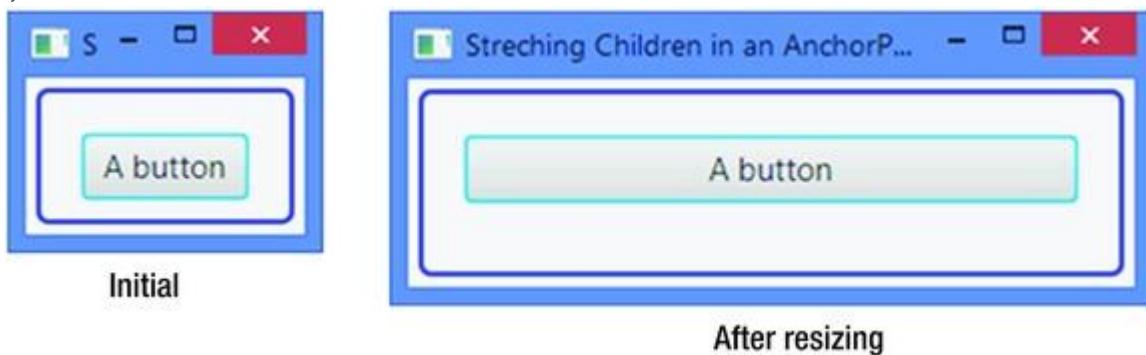
    @Override
    public void start(Stage stage) {
        Button leftRight = new Button("A button");
        AnchorPane.setTopAnchor(leftRight, 10.0);
        AnchorPane.setLeftAnchor(leftRight, 10.0);
        AnchorPane.setRightAnchor(leftRight, 10.0);
    }
}
```

```

        AnchorPane root = new AnchorPane();
        root.getChildren().addAll(leftRight);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Stretching Children in an
AnchorPane");
        stage.show();
    }
}

```

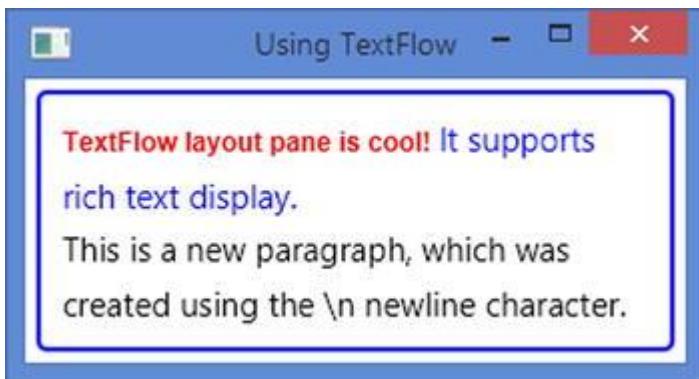


**Figure 10-56.** An AnchorPane with a Button anchored to Opposite sides

## Understanding TextFlow

A TextFlow layout pane is designed to display rich text. The rich text is composed of multiple Textnodes. The TextFlow combines the text in all Text nodes to display in a single text flow. A new line character ('\n') in the text of the Text child nodes indicates the start of a new paragraph. The text is wrapped at the width of the TextFlow.

A Text node has its position, size, and wrapping width. However, when it is added to a TextFlowpane, these properties are ignored. Text nodes are placed one after another wrapping them when necessary. A Text node in a TextFlow may span multiple lines in a TextFlow, whereas in a Textnode it is displayed in only one line. Figure 10-57 shows a window with a TextFlow as its root node.



**Figure 10-57.** A TextFlow showing rich text

The TextFlow is especially designed to display rich text using multiple Text nodes. However, you are not limited to adding only Text nodes to a TextFlow. You can add any other nodes to it, for example: Buttons, TextFields, etc. Nodes other than Text nodes are displayed using their preferred sizes.

**Tip** You can think of a TextFlow very similar to a FlowPane. Like a FlowPane, a TextFlow lays out its children in a flow from one end to another by treating Text nodes differently. When a Text node is encountered past its width boundary, it breaks the text of the Text node at its width and displays the remaining text in the next line.

### Creating TextFlow Objects

Unlike the classes for other layout panes, the TextFlow class is in the `javafx.scene.text` package where all other text related classes exist.

You can create an empty TextFlow using the no-args constructor:

```
TextFlow tflow1 = new TextFlow();
```

You can also specify the initial list of children for the TextFlow when you create it:

```
Text tx1 = new Text("TextFlow layout pane is cool! ");
Text tx2 = new Text("It supports rich text display.");
TextFlow tflow2 = new TextFlow(tx1, tx2);
```

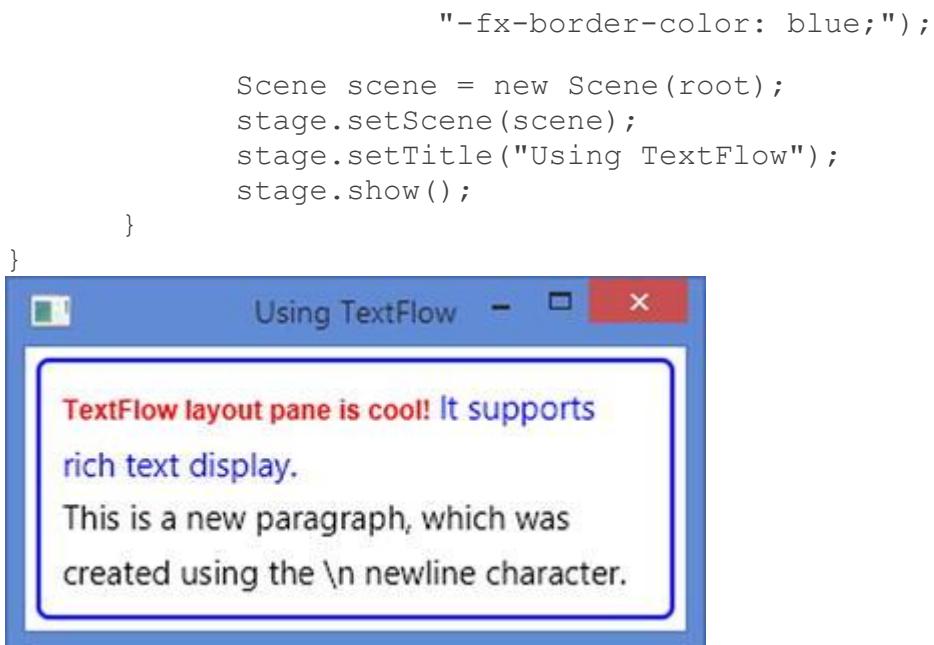
You can add children to a TextFlow after you create it.

```
Text tx1 = new Text("TextFlow layout pane is cool! ");
Text tx2 = new Text("It supports rich text display.");
TextFlow tflow3 = new TextFlow();
tflow3.getChildren().addAll(tx1, tx2);
```

The program in Listing 10-38 shows how to use a TextFlow. It adds three Text nodes to a TextFlow. The text in the third Text node starts with a newline character (\n), which starts a new paragraph. The program sets the preferred width of the TextFlow to 300px and the line

spacing to 5px. Figure 10-58 shows the window. When you resize the window, the TextFlow redraws the text wrapping, if necessary, at the new width.

***Listing 10-38.*** Using the TextFlow Layout Pane to Display Rich Text



**Figure 10-58.** Several Text nodes displayed in a TextFlow as rich text

A TextFlow also lets you embed nodes other than Text nodes. You can create a form to display text mixed with other types of nodes that users can use. The program in Listing 10-39 embeds a pair of RadioButtons, a TextField, and a Button to a TextFlow to create an online form with text. Users can use these nodes to interact with the form.

Figure 10-59 shows the window. At the time of testing this example, the RadioButtons and TextField nodes did not gain focus using the mouse. Use the Tab key to navigate to these nodes and the spacebar to select a RadioButton.

### **Listing 10-39.** Embedding Nodes Other Than Text Nodes in a TextFlow

```

// TextFlowEmbeddingNodes.java
package com.jdojo.container;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;
import javafx.scene.text.Text;
import javafx.scene.text.TextFlow;
import javafx.stage.Stage;

public class TextFlowEmbeddingNodes extends Application {
    public static void main(String[] args) {

```

```
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text tx1 = new Text("I, ");

        RadioButton rb1 = new RadioButton("Mr.");
        RadioButton rb2 = new RadioButton("Ms.");
        rb1.setSelected(true);

        ToggleGroup group = new ToggleGroup();
        rb1.setToggleGroup(group);
        rb2.setToggleGroup(group);

        TextField nameFld = new TextField();
        nameFld.setPromptText("Your Name");

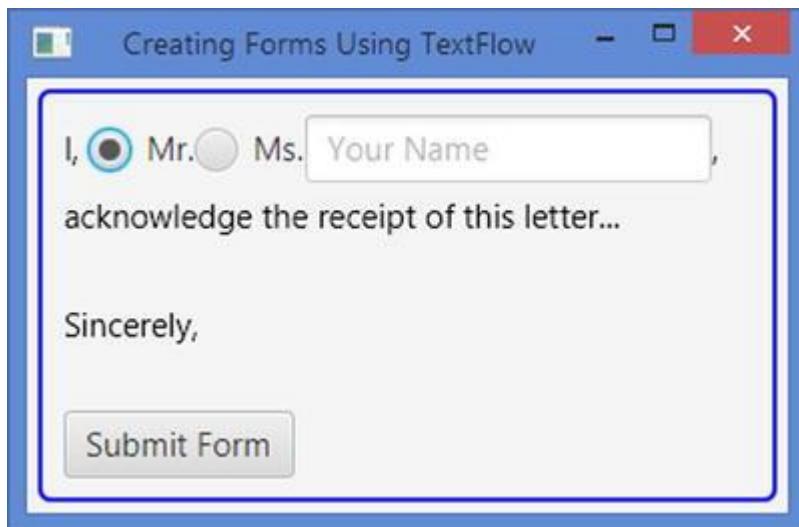
        Text tx2 = new Text(", acknowledge the receipt of
this letter...\n\n" +
                    "Sincerely,\n\n");

        Button submitFormBtn = new Button("Submit Form");

        // Create a TextFlow object with all nodes
        TextFlow root = new TextFlow(tx1, rb1, rb2, nameFld,
        tx2, submitFormBtn);

        // Set the preferred width and line spacing
        root.setPrefWidth(350);
        root.setLineSpacing(5);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Creating Forms Using TextFlow");
        stage.show();
    }
}
```



**Figure 10-59.** Nodes other than Text nodes embedded in a TextFlow

### TextFlow Properties

The TextFlow class contains two properties, as listed in Table 10-11, to customize its layout.

**Table 10-11.** The List of Properties Declared in the GridPane Class

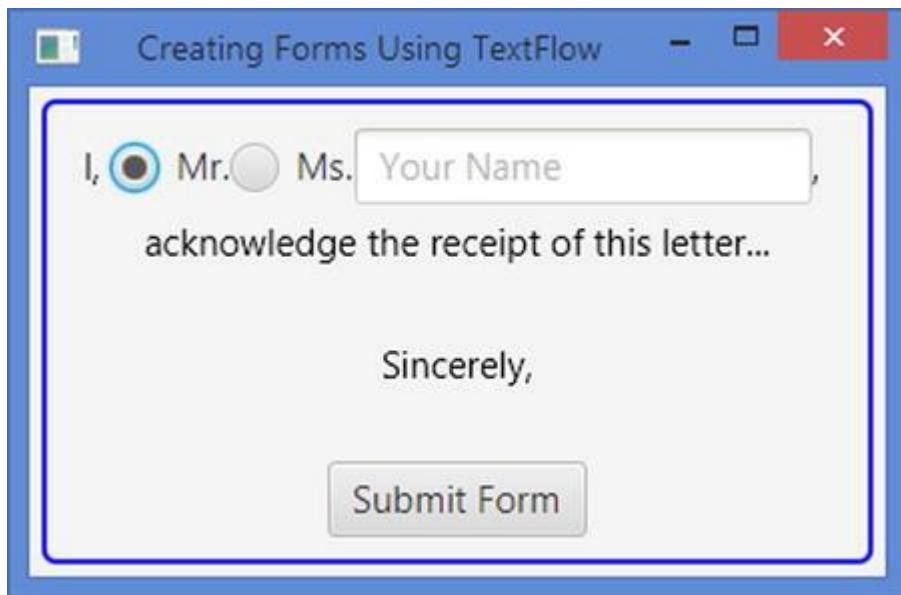
Property	Type	Description
lineSpacing	DoubleProperty	It specifies the vertical space between lines in a TextFlow. Its value is 0px.
textAlignment	ObjectProperty<TextAlignment>	It specifies the alignment of the overall content of the TextFlow. Its value is LEFT and JUSTIFY. Its default value is LEFT.

The lineSpacing property specifies the vertical space (in pixel) between lines in a TextFlow. We have used it in our previous examples.

```
TextFlow tflow = new TextFlow();
tflow.setLineSpacing(5); // 5px lineSpacing
```

The textAlignment property specifies the alignment of the overall content of the TextFlow. By default, the content is aligned to the left. Figure 10-60 shows the window for the program in Listing 10-39 when the following statement is added after the TextFlow object is created in the program.

```
// Set the textAlignment to CENTER
root.setAlignment(TextAlignment.CENTER);
```



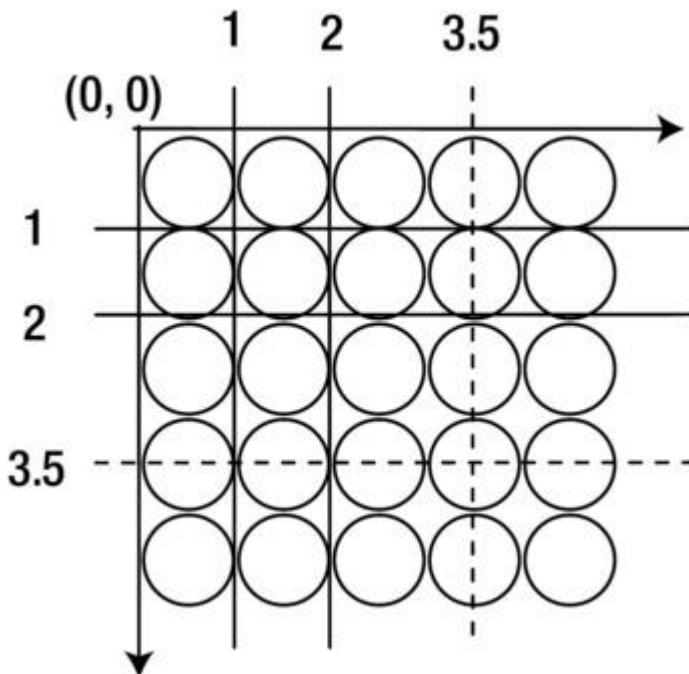
**Figure 10-60.** A TextFlow using CENTER as its textAlignment

### Setting Constraints for Children in TextFlow

TextFlow does not allow you to add any constraints to its children, not even a margin.

### Snapping to Pixel

Figure 10-61 shows a screen of a device that is five pixels wide and five pixels tall. A circle in the figure represents a pixel. A coordinate  $(0, 0)$  is mapped to the upper-left corner of the upper-left pixel. The center of the upper-left pixel maps to the coordinates  $(0.5, 0.5)$ . All integer coordinates fall in the corners and cracks between the pixels. In the figure, solid lines are drawn through the cracks of pixels and dashed lines through the centers of the pixels.



**Figure 10-61.** A 5X5 pixel region on the screen

In JavaFX, coordinates can be specified in floating-point numbers: for example, 0.5, 6.0, etc., which lets you represent any part of a pixel. If the floating-point number is an integer (e.g., 2.0, 3.0, etc.), it will represent corners of the pixel.

A `Region` using floating-point numbers as coordinates will not align exactly at the pixel boundary and its border may look fuzzy.

The `Region` class contains a `snapToPixel` property to address this issue. By default, it is set to true and a `Region` adjusts the position, spacing, and size values of its children to an integer to match the pixel boundaries, resulting in crisp boundaries for the children. If you do not want a `Region` to adjust these values to integers, set the `snapToPixel` property to false.

## Summary

A *layout pane* is a node that contains other nodes, which are known as its children (or child nodes). The responsibility of a layout pane is to lay out its children, whenever needed. A layout pane is also known as a *container* or a *layout container*. A layout pane has a *layout policy* that controls how the layout pane lays out its children. For example, a layout pane may lay out its children horizontally, vertically, or in any other fashion. JavaFX contains several layout-related classes. A layout pane computes the position and size of its children. The layout policy of a layout pane is a set of rules to compute the position and size of its children.

Objects of the following classes represent layout

panes: HBox, VBox, FlowPane, BorderPane, StackPane, TilePane, GridPane, AnchorPane, and TextFlow. All layout pane classes inherits from the Pane class.

A Group has features of a container; for example, it has its own layout policy, coordinate system, and it is a subclass of the Parent class. However, its meaning is best reflected by calling it a *collection of nodes* or a *group*, rather than a *container*. It is used to manipulate a collection of nodes as a single node (or as a group). Transformations, effects, and properties applied to a Group are applied to all nodes in the Group. A Group has its own layout policy, which does not provide any specific layout to its children, except giving them their preferred size. An HBox lays out its children in a single horizontal row. It lets you set the horizontal spacing between adjacent children, margins for any children, resizing behavior of children, etc. It uses opx as the default spacing between adjacent children. The default width of the content area and HBox is wide enough to display all its children at their preferred widths and the default height is the largest of the heights of all its children.

A VBox lays out its children in a single vertical column. It lets you set the vertical spacing between adjacent children, margins for any children, resizing behavior of children, etc. It uses opx as the default spacing between adjacent children. The default height of the content area of a VBox is tall enough to display all its children at their preferred heights, and the default width is the largest of the widths of all its children.

A FlowPane is a simple layout pane that lays out its children in rows or columns wrapping at a specified width or height. It lets its children flow horizontally or vertically, and hence the name “flow pane.” You can specify a preferred wrap length, which is the preferred width for a horizontal flow and the preferred height for a vertical flow, where the content is wrapped. A FlowPane is used in situations where the relative locations of children are not important: for example, displaying a series of pictures or buttons.

A BorderPane divides its layout area into five regions: top, right, bottom, left, and center. You can place at most one node in each of the five regions. The children in the top and bottom regions are resized to their preferred heights. Their widths are extended to fill the available extra horizontal space, provided the maximum widths of the children allow extending their widths beyond their preferred widths. The children in the right and left regions are resized to their preferred widths. Their heights are extended to fill the extra vertical space, provided the maximum heights of the children allow extending their heights beyond

their preferred heights. The child node in the center will fill the rest of the available space in both directions.

A `StackPane` lays out its children in a stack of nodes. It provides a powerful means to overlay nodes. Children are drawn in the order they are added.

A `TilePane` lays out its children in a grid of uniformly sized cells, known as tiles. `TilePanes` work similar to `FlowPanes` with one difference: In a `FlowPane`, rows and columns can be of different heights and widths, whereas in a `TilePane`, all rows have the same heights and all columns have the same widths. The width of the widest child node and the height of the tallest child node are the default widths and heights of all tiles in a `TilePane`. The orientation of a `TilePane`, which can be set to horizontal or vertical, determines the direction of the flow for its content. By default, a `TilePane` has a horizontal orientation.

A `GridPane` lays out its children in a dynamic grid of cells arranged in rows and columns. The grid is dynamic because the number and size of cells in the grid are determined based on the number of children. They depend on the constraints set on children. Each cell in the grid is identified by its position in the column and row. The indexes for columns and rows start at 0. A child node may be placed anywhere in the grid spanning more than one cell. All cells in a row are of the same height. Cells in different rows may have different heights. All cells in a column are of the same width. Cells in different columns may have different widths. By default, a row is tall enough to accommodate the tallest child node in it. A column is wide enough to accommodate the widest child node in it. You can customize the size of each row and column. `GridPane` also allows for vertical spacing between rows and horizontal spacing between columns. For debug purposes, you can show the grid lines. Figure 10-41 shows three instances of the `GridPane`.

An `AnchorPane` lays out its children by anchoring the four edges of its children to its own four edges at a specified distance.

An `AnchorPane` may be used for aligning children along one or more edges of the `AnchorPane` or for stretching children when the `AnchorPane` is resized.

The specified distance between the edges of the children and the edges of the `AnchorPane` is called the *anchor* constraint for the sides it is specified. When you anchor a child node to the two opposite edges (top/bottom or left/right), the children are resized to maintain the specified anchor distance as the `AnchorPane` is resized.

A `TextFlow` layout pane is designed to display rich text. The rich text is composed of multiple `Textnodes`. The `TextFlow` combines the text in all `Text` nodes to display in a single text flow. A new line character

(' \n ') in the text of the Text child nodes indicates the start of a new paragraph. The text is wrapped at the width of the TextFlow.

## CHAPTER 1



### Getting Started

In this chapter, you will learn:

- What JavaFX is
- The history of JavaFX
- How to write your first JavaFX application
- How to use the NetBeans Integrated Development Environment to work with a JavaFX application
- How to pass parameters to a JavaFX application
- How to launch a JavaFX application
- The life cycle of a JavaFX application
- How to terminate a JavaFX Application

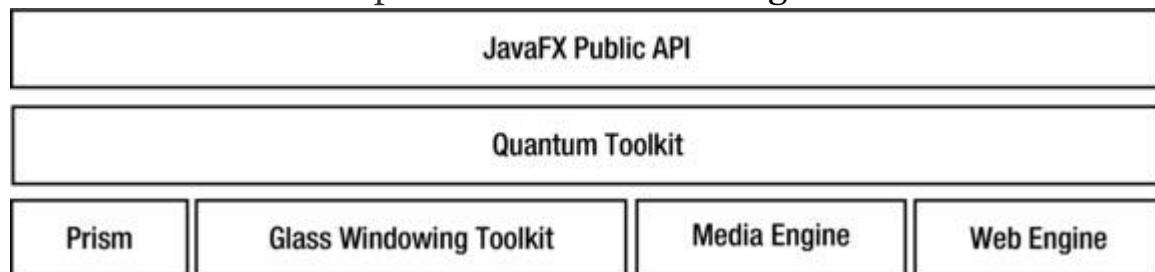
### What Is JavaFX?

JavaFX is an open source Java-based framework for developing rich client applications. It is comparable to other frameworks on the market such as Adobe Flex and Microsoft Silverlight. JavaFX is also seen as the successor of Swing in the arena of graphical user interface (GUI) development technology in Java platform. The JavaFX library is available as a public Java application programming interface (API). JavaFX contains several features that make it a preferred choice for developing rich client applications:

- JavaFX is written in Java, which enables you to take advantage of all Java features such as multithreading, generics, and lambda expressions. You can use any Java editor of your choice, such as NetBeans, to author, compile, run, debug, and package your JavaFX application.
- JavaFX supports data binding through its libraries.
- JavaFX code can be written using any Java virtual machine (JVM)-supported scripting languages such as Visage, Groovy, and Scala.
- JavaFX offers two ways to build a user interface (UI): using Java code and using FXML. FXML is an XML-based scriptable markup language to define a UI declaratively. Oracle provides a tool called Scene Builder, which is a visual editor for FXML.

- JavaFX provides a rich set of multimedia support such as playing back audios and videos. It takes advantage of available codecs on the platform.
- JavaFX lets you embed web content in the application.
- JavaFX provides out-of-the-box support for applying effects and animations, which are important for developing gaming applications. You can achieve sophisticated animations by writing a few lines of code.

Behind the JavaFX API lies a number of components to take advantage of the Java native libraries and the available hardware and software. JavaFX components are shown in Figure 1-1.



**Figure 1-1.** Components of the JavaFX platform

The GUI in JavaFX is constructed as a *scene graph*. A scene graph is a collection of visual elements, called nodes, arranged in a hierarchical fashion. A scene graph is built using the public JavaFX API. Nodes in a scene graph can handle user inputs and user gestures. They can have effects, transformations, and states. Types of nodes in a scene graph include simple UI controls such as buttons, text fields, two-dimensional (2D) and three-dimensional (3D) shapes, images, media (audio and video), web content, and charts.

*Prism* is a hardware-accelerated graphics pipeline used for rendering the scene graph. If hardware-accelerated rendering is not available on the platform, Java 2D is used as the fallback rendering mechanism. For example, before using Java 2D for rendering, it will try using DirectX on Windows and OpenGL on Mac Linux and embedded platforms.

The *Glass Windowing Toolkit* provides graphics and windowing services such as windows and the timer using the native operating system. The toolkit is also responsible for managing event queues. In JavaFX, event queues are managed by a single, operating system-level thread called *JavaFX Application Thread*. All user input events are dispatched on the JavaFX Application Thread. JavaFX requires that a live scene graph must be modified only on the JavaFX Application Thread.

Prism uses a separate thread, other than the JavaFX Application Thread, for the rendering process. It accelerates the process by rendering a frame while the next frame is being processed. When a scene graph is modified, for example, by entering some text in the text field, Prism needs to re-render the scene graph. Synchronizing the scene graph with Prism is accomplished using an event called a *pulseevent*. A pulse event is queued on the JavaFX Application Thread when the scene graph is modified and it needs to be re-rendered. A pulse event is an indication that the scene graph is not in sync with the rendering layer in Prism, and the latest frame at the Prism level should be rendered. Pulse events are throttled at 60 frames per second maximum.

The media engine is responsible for providing media support in JavaFX, for example, playing back audios and videos. It takes advantage of the available codecs on the platform. The media engine uses a separate thread to process media frames and uses the JavaFX Application Thread to synchronize the frames with the scene graph. The media engine is based on *GStreamer*, which is an open source multimedia framework.

The web engine is responsible for processing web content (HTML) embedded in a scene graph. Prism is responsible for rendering the web contents. The web engine is based on *WebKit*, which is an open source web browser engine. HTML5, Cascading Style Sheets (CSS), JavaScript, and Document Object Model (DOM) are supported.

Quantum toolkit is an abstraction over the low-level components of Prism, Glass, Media Engine, and Web Engine. It also facilitates coordination between low-level components.

**Note** Throughout this book, it is assumed that you have intermediate-level knowledge of the Java programming language. Familiarity with the new features in Java 8 such as lambda expressions and Time API is also assumed.

## History of JavaFX

JavaFX was originally developed by Chris Oliver at SeeBeyond and it was called F3 (Form Follows Function). F3 was a Java scripting language for easily developing GUI applications. It offered declarative syntax, static typing, type inference, data binding, animation, 2D graphics, and Swing components. SeeBeyond was bought by Sun Microsystems and F3 was renamed JavaFX in 2007. Oracle acquired Sun Microsystems in 2010. Oracle then open sourced JavaFX in 2013.

The first version of JavaFX was released in the fourth quarter of 2008. The current release for JavaFX is version 8.0. The version number jumped from 2.2 to 8.0. From Java 8, the version numbers of Java SE and JavaFX will be the same. The major versions for Java SE and JavaFX will be released at the same time as well. Table 1-1 contains a list

of releases of JavaFX. Starting with the release of Java SE 8, JavaFX is part of the Java SE runtime library. From Java 8, you do not need any extra set up to compile and run your JavaFX programs.

**Table 1-1. JavaFX Releases**

<b>Release Date</b>	<b>Version</b>	<b>Comments</b>
Q4, 2008	JavaFX 1.0	It was the initial release of JavaFX. It used a declaration language JavaFX Script to write the JavaFX code.
Q1, 2009	JavaFX 1.1	Support for JavaFX Mobile was introduced.
Q2, 2009	JavaFX 1.2	
Q2, 2010	JavaFX 1.3	
Q3, 2010	JavaFX 1.3.1	
Q4, 2011	JavaFX 2.0	Support for JavaFX script was dropped. It used the Java language to write the JavaFX code. Support for JavaFX Mobile was dropped.
Q2, 2012	JavaFX 2.1	Support for Mac OS for desktop only was introduced.
Q3, 2012	JavaFX 2.2	
Q1, 2014	JavaFX 8.0	JavaFX version jumped from 2.2 to 8.0. JavaFX and Java SE versions now match from Java 8.

## System Requirements

You need to have the following software installed on your computer:

- Java Development Kit 8
- NetBeans IDE 8.0 or later

It is not necessary to have the NetBeans IDE to compile and run the programs in this book. However, the NetBeans IDE has special features for creating, running, and packaging JavaFX applications to make developers' lives easier. You can use any other IDE, for example, Eclipse, JDeveloper, or IntelliJ IDEA.

## JavaFX Runtime Library

All JavaFX classes are packaged in a Java Archive (JAR) file named `jfxrt.jar`. The JAR file is located in the `jre\lib\ext` directory under the Java home directory.

If you compile and run JavaFX programs on the command line, you do not need to worry about setting the JavaFX runtime JAR file in the CLASSPATH. Java 8 compiler (the `javac` command) and launcher (the `java` command) automatically include the JavaFX runtime JAR file in the CLASSPATH.

The NetBeans IDE automatically includes the JavaFX runtime JAR file in the CLASSPATH when you create a Java or JavaFX project. If you are using an IDE other than NetBeans, you may need to include `jfxrt.jar` in the IDE CLASSPATH to compile and run a JavaFX application from inside the IDE.

## JavaFX Source Code

Experienced developers sometimes prefer to look at the source code of the JavaFX library to learn how things are implemented behind the scenes. Oracle provides the JavaFX source code. The Java 8 installation copies the source in the Java home directory. The file name is `javafx-src.zip`. Unzip the file to a directory and use your favorite Java editor to open the source code.

## Your First JavaFX Application

Let's write your first JavaFX application. It should display the text "Hello JavaFX" in a window. I will take an incremental, step-by-step approach to explain how to develop this first application. I will add as few lines of code as possible, and then, explain what the code does and why it is needed.

Creating the *HelloJavaFX* Class

A JavaFX application is a class that must inherit from the `Application` class that is in the `javafx.application` package. You will name your class `HelloFXApp` and it will be stored in the `com.jdojo.intro` package. Listing 1-1 shows the initial code for the `HelloFXApp` class. Note that the `HelloFXApp` class will not compile at this point. You will fix it in the next section.

### ***Listing 1-1.*** Inheriting Your JavaFX Application Class from the `javafx.application.Application` Class

```
// HelloFXApp.java
package com.jdojo.intro;

import javafx.application.Application;
public class HelloFXApp extends Application {
    // Application logic goes here
}
```

The program includes a package declaration, an import statement, and a class declaration. There is nothing like JavaFX in the code. It looks like any other Java program. However, you have fulfilled one of the requirements of the JavaFX application by inheriting the `HelloFXApp` class from the `Application` class.

### Overriding the `start()` Method

If you try compiling the `HelloFXApp` class, it will result in the following compile-time error: *HelloFXApp is not abstract and does not override abstract method start(Stage) in Application*. The error is stating that the `Application` class contains an abstract `start(Stage stage)` method, which has not been overridden in the `HelloFXApp` class. As a Java developer, you know what to do next: you either declare the `HelloFXApp` class as abstract or provide an implementation for the `start()` method. Here let's provide an implementation for the `start()` method. The `start()` method in the `Application` class is declared as follows:

```
public abstract void start(Stage stage) throws
java.lang.Exception
```

Listing 1-2 shows the revised code for the `HelloFXApp` class that overrides the `start()` method.

### ***Listing 1-2.*** Overriding the `start()` Method in Your JavaFX Application Class

```
// HelloFXApp.java
package com.jdojo.intro;

import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFXApp extends Application {
    @Override
    public void start(Stage stage) {
        // The logic for starting the application goes here
    }
}
```

In the revised code, you have incorporated two things:

- You have added one more `import` statement to import the `Stage` class from the `javafx.stage` package.
- You have implemented the `start()` method. The `throws` clause for the method is dropped, which is fine by the rules for overriding methods in Java.

The `start()` method is the entry point for a JavaFX application. It is called by the JavaFX application launcher. Notice that the `start()` method is passed an instance of the `Stage` class, which is known as the *primary stage* of the application. You can create more stages as necessary in your application. However, the primary stage is always created by the JavaFX runtime for you.

**Tip** Every JavaFX application class must inherit from the `Application` class and provide the implementation for the `start(Stage stage)` method.

## Showing the Stage

Similar to a stage in the real world, a JavaFX stage is used to display a scene. A scene has visuals—such as text, shapes, images, controls, animations, and effects—with which the user may interact, as is the case with all GUI-based applications.

In JavaFX, the primary stage is a container for a scene. The stage look-and-feel is different depending on the environment your application is run in. You do not need to take any action based on the environment because the JavaFX runtime takes care of all the details for you. For example, if the application runs as a desktop application, the primary stage will be a window with a title bar and an area to display the scene; if the application runs an applet in a web browser, the primary stage will be an embedded area in the browser window.

The primary stage created by the application launcher does not have a scene. You will create a scene for your stage in the next section.

You must show the stage to see the visuals contained in its scene. Use the `show()` method to show the stage. Optionally, you can set a title for the stage using the `setTitle()` method. The revised code for the `HelloFXApp` class is shown in Listing 1-3.

### ***Listing 1-3.*** Showing the Primary Stage in Your JavaFX Application Class

```
// HelloFXApp.java
package com.jdojo.intro;

import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFXApp extends Application {
    @Override
    public void start(Stage stage) {
        // Set a title for the stage
        stage.setTitle("Hello JavaFX Application");

        // Show the stage
        stage.show();
    }
}
```

### Launching the Application

You are now ready to run your first JavaFX application. You can use one of the following two options to run it:

- It is not necessary to have a `main()` method in the class to start a JavaFX application. When you run a Java class that inherits from the `Application` class, the `java` command launches the JavaFX application if the class being run does not contain the `main()` method.
- If you include a `main()` method in the JavaFX application class inside the `main()` method, call the `launch()` static method of the `Application` class to launch the application. The `launch()` method takes a `String` array as an argument, which are the parameters passed to the JavaFX application.

If you are using the first option, you do not need to write any additional code for the `HelloFXApp` class. If you are using the second option, the revised code for the `HelloFXApp` class with the `main()` method will be as shown in Listing 1-4.

### ***Listing 1-4.*** The HelloFXApp JavaFX Application Without a Scene

```
// HelloFXApp.java
package com.jdojo.intro;

import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFXApp extends Application {
    public static void main(String[] args) {
        // Launch the JavaFX application
        Application.launch(args);
    }

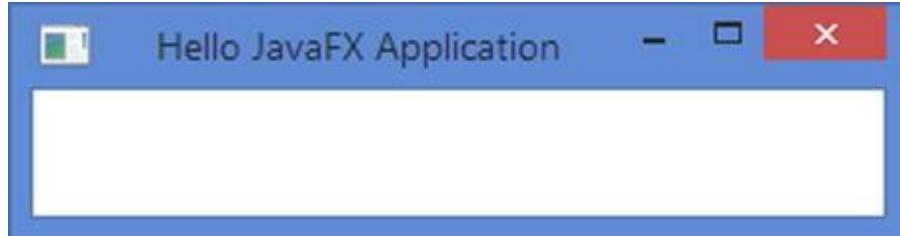
    @Override
    public void start(Stage stage) {
        stage.setTitle("Hello JavaFX Application");
        stage.show();
    }
}
```

The `main()` method calls the `launch()` method, which will do some setup work and call the `start()` method of the `HelloFXApp` class. Your `start()` method sets the title for the primary stage and shows the stage. Compile the `HelloFXApp` class using the following command:

```
javac com/jdojo/intro/HelloFXApp.java
```

Run the `HelloFXApp` class using the following command, which will display a window with a title bar as shown in Figure 1-2:

```
java com.jdojo.intro.HelloFXApp
```



**Figure 1-2.** The `HelloFXApp` JavaFX Application Without a Scene

The main area of the window is empty. This is the content area in which the stage will show its scene. Because you do not have a scene for your stage yet, you will see an empty area. The title bar shows the title that you have set in the `start()` method.

You can close the application using the Close menu option in the window title bar. Use Alt + F4 to close the window in Windows. You can use any other option to close the window as provided by your platform.

**Tip** The `launch()` method of the `Application` class does not return until all windows are closed or the application exits using the `Platform.exit()` method. The `Platform` class is in the `javafx.application` package.

You haven't seen anything exciting in JavaFX yet! You need to wait for that until you create a scene in the next section.

### Adding the `main()` Method

As described in the previous section, the Java 8 launcher (the `java` command) does not require a `main()` method to launch a JavaFX application. If the class that you want to run inherits from the `Application` class, the `java` command launches the JavaFX application by automatically calling the `Application.launch()` method for you.

If you are using the NetBeans IDE to create the JavaFX project, you do not need to have a `main()` method to launch your JavaFX application if you run the application by running the JavaFX project. However, the NetBeans IDE requires you to have a `main()` method when you run the JavaFX application class as a file, for example, by selecting the `HelloFXApp` file, right-clicking it, and selecting the Run File option from the menu.

Some IDEs still require the `main()` method to launch a JavaFX application. All examples in this chapter will include the `main()` method that will launch the JavaFX applications.

### Adding a Scene to the Stage

An instance of the `Scene` class, which is in the `javafx.scene` package, represents a scene. A stage contains one scene, and a scene contains visual contents.

The contents of the scene are arranged in a tree-like hierarchy. At the top of the hierarchy is the *rootnode*. The root node may contain child nodes, which in turn may contain their child nodes, and so on. You must have a root node to create a scene. You will use a `VBox` as the root node. `VBox` stands for vertical box, which arranges its children vertically in a column. The following statement creates a `VBox`:

```
VBox root = new VBox();
```

**Tip** Any node that inherits from the `javafx.scene.Parent` class can be used as the root node for a scene. Several nodes, known as layout panes or containers such as `VBox`, `HBox`, `Pane`, `FlowPane`, `GridPane`, or `TilePane` can be used as a root node. `Group` is a special container that groups its children together.

A node that can have children provides a `getChildren()` method that returns an `ObservableList` of its children. To add a child node to

a node, simply add the child node to the `ObservableList`. The following snippet of code adds a `Text` node to a `VBox`:

```
// Create a VBox node
VBox root = new VBox();

// Create a Text node
Text msg = new Text("Hello JavaFX");

// Add the Text node to the VBox as a child node
root.getChildren().add(msg);
```

The `Scene` class contains several constructors. You will use the one that lets you specify the root node and the size of the scene. The following statement creates a scene with the `VBox` as the root node, with 300px width and 50px height:

```
// Create a scene
Scene scene = new Scene(root, 300, 50);
```

You need to set the scene to the stage by calling the `setScene()` method of the `Stage` class:

```
// Set the scene to the stage
stage.setScene(scene);
```

That's it. You have completed your first JavaFX program with a scene. Listing 1-5 contains the complete program. The program displays a window as shown in Figure 1-3.

### ***Listing 1-5.*** A JavaFX Application with a Scene Having a Text Node

```
// HelloFXAppWithAScene.java
package com.jdojo.intro;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class HelloFXAppWithAScene extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text msg = new Text("Hello JavaFX");
        VBox root = new VBox();
        root.getChildren().add(msg);

        Scene scene = new Scene(root, 300, 50);
        stage.setScene(scene);
        stage.setTitle("Hello JavaFX Application with
a Scene");
    }
}
```



**Figure 1-3.** A JavaFX application with a scene having a *Text* node

## Improving the *HelloFX* Application

JavaFX is capable of doing much more than you have seen so far. Let's enhance the first program and add some more user interface elements such as buttons and text fields. This time, the user will be able to interact with the application. Use an instance of the `Button` class to create a button as shown:

```
// Create a button with "Exit" text
Button exitBtn = new Button("Exit");
```

When a button is clicked, an `ActionEvent` is fired. You can add an `ActionEvent` handler to handle the event. Use the `setOnAction()` method to set an `ActionEvent` handler for the button. The following statement sets an `ActionEvent` handler for the button. The handler terminates the application. You can use a lambda expression or an anonymous class to set the `ActionEvent` handler. The following snippet of code shows both approaches:

```
// Using a lambda expression
exitBtn.setOnAction(e -> Platform.exit());

// Using an anonymous class
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
...
exitBtn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        Platform.exit();
    }
});
```

The program in Listing 1-6 shows how to add more nodes to the scene. The program uses the `setStyle()` method of the `Label` class to set the fill color of the `Label` to blue. I will discuss using CSS in JavaFX later.

## **Listing 1-6.** Interacting with Users in a JavaFX Application

```
// ImprovedHelloFXApp.java
package com.jdojo.intro;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ImprovedHelloFXApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Enter your name:");
        TextField nameFld = new TextField();

        Label msg = new Label();
        msg.setStyle("-fx-text-fill: blue;");

        // Create buttons
        Button sayHelloBtn = new Button("Say Hello");
        Button exitBtn = new Button("Exit");

        // Add the event handler for the Say Hello button
        sayHelloBtn.setOnAction(e -> {
            String name = nameFld.getText();
            if (name.trim().length() > 0) {
                msg.setText("Hello " + name);
            } else {
                msg.setText("Hello there");
            }
        });

        // Add the event handler for the Exit button
        exitBtn.setOnAction(e -> Platform.exit());

        // Create the root node
        VBox root = new VBox();

        // Set the vertical spacing between children to 5px
        root.setSpacing(5);

        // Add children to the root node
        root.getChildren().addAll(nameLbl, nameFld, msg,
sayHelloBtn, exitBtn);

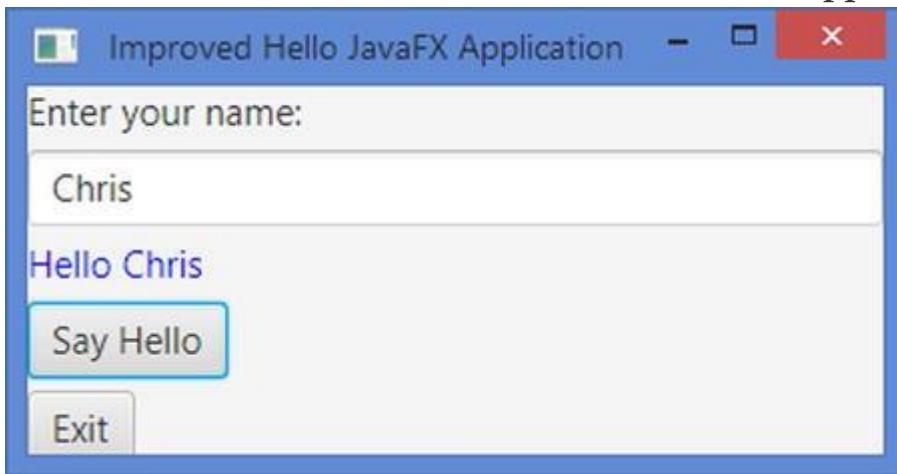
        Scene scene = new Scene(root, 350, 150);
        stage.setScene(scene);
        stage.setTitle("Improved Hello JavaFX Application");
    }
}
```

```

        stage.show();
    }
}

```

The improved HelloFX program displays a window as shown in Figure 1-4. The window contains two labels, a text field, and two buttons. A `VBox` is used as the root node for the scene. Enter a name in the text field and click the Say Hello button to see a hello message. Clicking the Say Hello button without entering a name displays the message Hello there. The application displays a message in a `Label` control. Click the Exit button to exit the application.



**Figure 1-4.** A JavaFX Application with few controls in its scene

## Using the NetBeans IDE

You can use the NetBeans IDE to create, compile, package, and run new JavaFX applications. The source code used in this book is available with a NetBeans project.

### Creating a New JavaFX Project

Use the following steps to create a new JavaFX project:

1. Select the **New Project...** menu option from the **File** menu. Alternatively, use the keyboard shortcut **Ctrl + Shift + N**.
2. A **New Project** dialog appears as shown in Figure 1-5. From the **Categories** list, select **JavaFX**. From the **Projects** list, select **JavaFX Application**. Click the **Next** button.
3. The **New JavaFX Application** dialog appears as shown in Figure 1-6. Enter the details of the project such as project name and location. The **Create Application Class** check box is checked by default. You can enter the full-qualified name of the JavaFX application in the box next to

the check box. NetBeans will create the class and add the initial code for you. When you run the project from inside the IDE, this class is run. You can change this class later.

4. Click the Finish button when you are done.

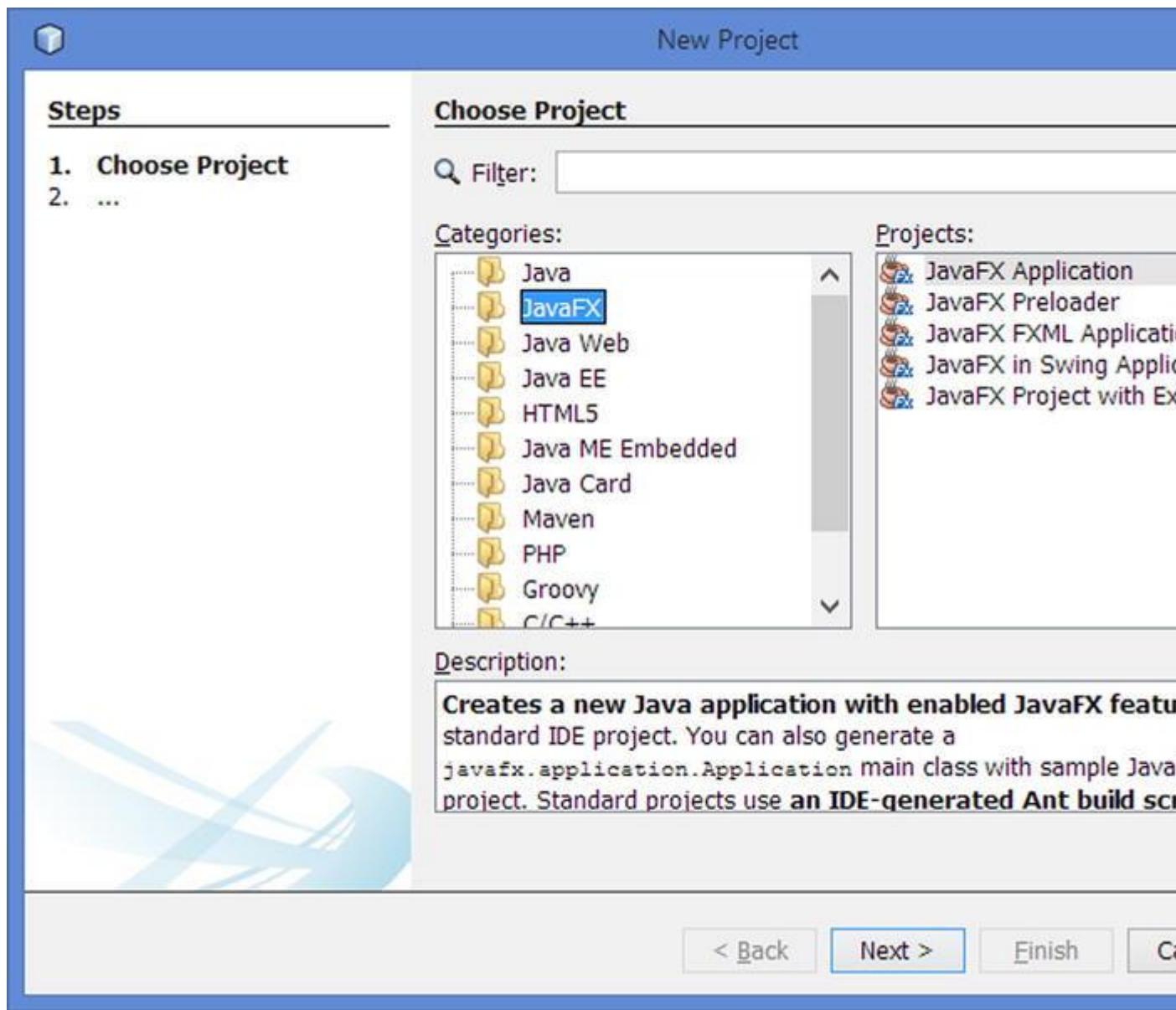
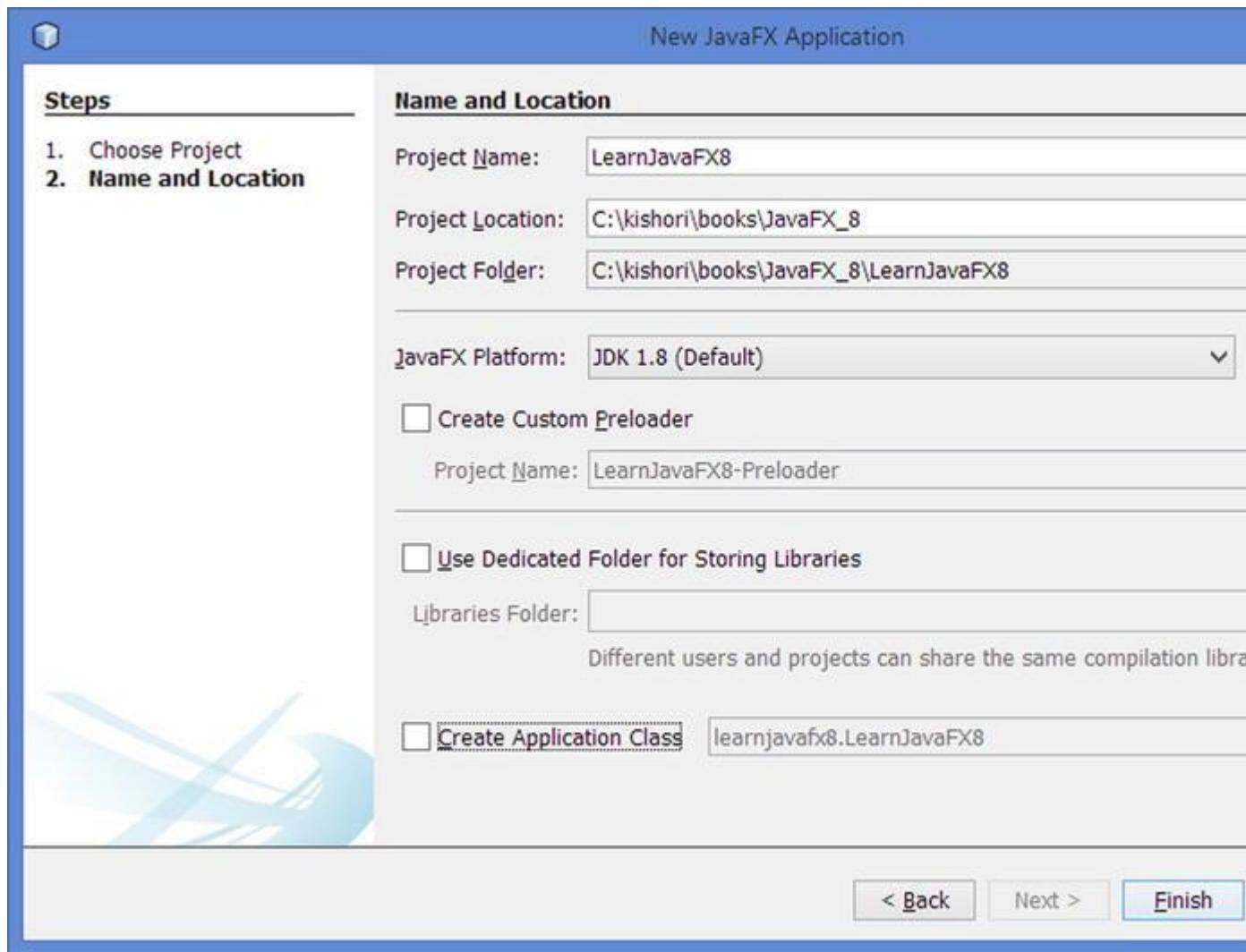


Figure 1-5. The New Project dialog

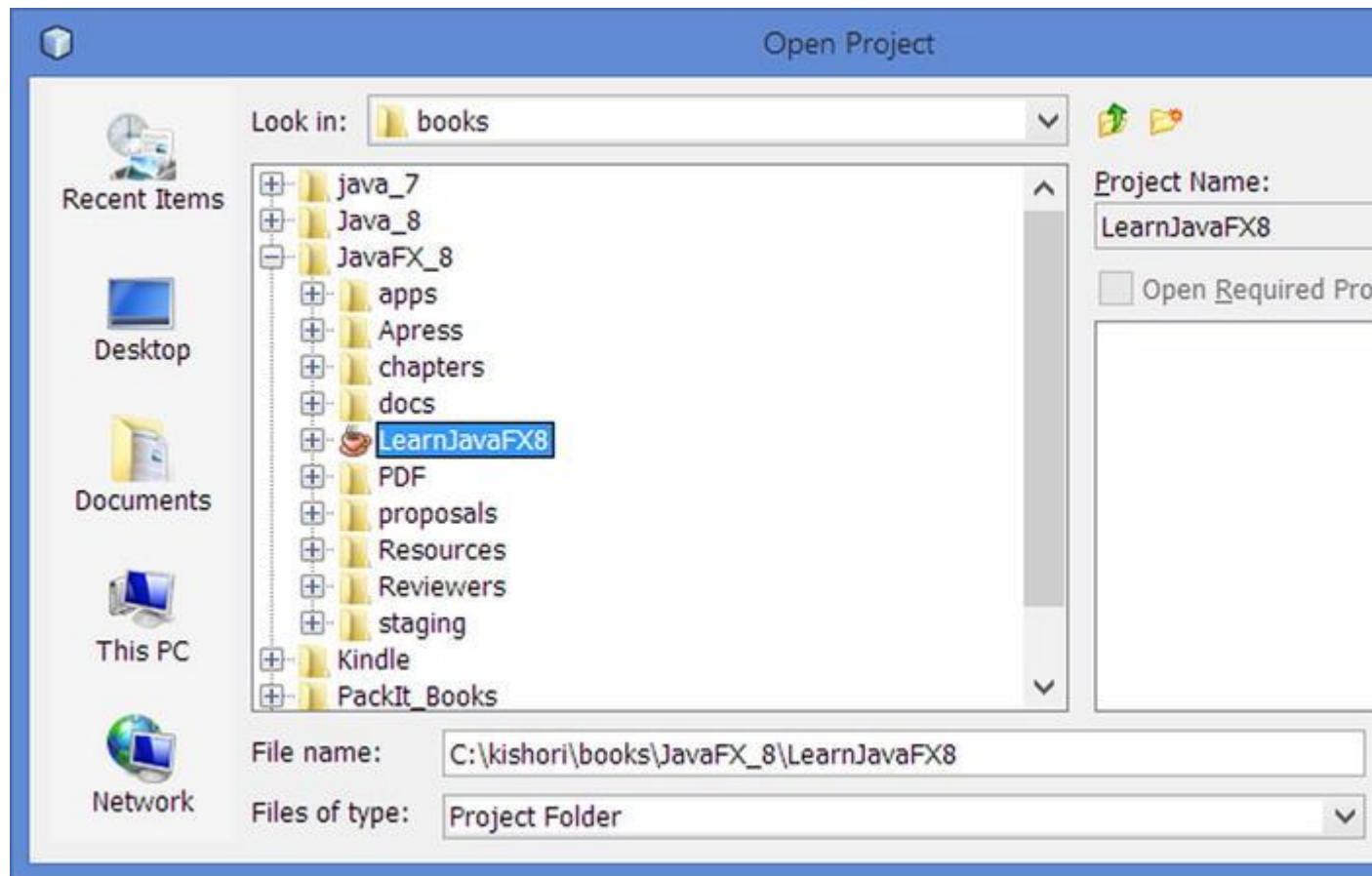


**Figure 1-6.** The New JavaFX Application dialog

### Opening an Existing JavaFX Project

The source code for this book is provided with a NetBeans project. You can use the following steps to open the project. If you have not downloaded the source code for this book, please do so before proceeding.

1. From inside the NetBeans IDE, select the Open Project... menu option from the File menu. Alternatively, use the keyboard shortcut Ctrl + Shift + O.
2. An Open Project dialog appears. Navigate to the directory containing the downloaded source code for this book. You should see the project LearnJavaFX8, as shown in Figure 1-7. Select the project name and click the Open Project button. The project should appear in the IDE.



**Figure 1-7.** The Open Project dialog

### Running a JavaFX Project from the NetBeans IDE

You can compile and run a JavaFX application from inside the NetBeans IDE. You have the options to run a Java application in one of three ways:

- Run as a standalone desktop application
- Run as a WebStart
- Run in a browser

By default, NetBeans runs a JavaFX application as a standalone desktop application. You can change the way your application is run on the project properties page under the Run category. To access the project properties page, select your project in the IDE, right-click, and select the Properties menu option. The Project Properties dialog box appears. Select the Run item from the Categories tree. Enter the desired Run properties for your project on the right side of the screen.

### Passing Parameters to a JavaFX Application

Like a Java application, you can pass parameters to a JavaFX application. There are two ways to pass parameters to a JavaFX application:

- On the command line for a standalone application
- In a Java Network Launching Protocol (JNLP) file for an applet and WebStart application

The `Parameters` class, which is a static inner class of the `Application` class, encapsulates the parameters passed to a JavaFX application. It divides parameters into three categories:

- Named parameters
- Unnamed parameters
- Raw parameters (a combination of named and unnamed parameters)

You need to use the following three methods of the `Parameters` class to access the three types of parameters:

- `Map<String, String> getNamed()`
- `List<String> getUnnamed()`
- `List<String> getRaw()`

A parameter can be named or unnamed. A named parameter consists of a (name, value) pair. An unnamed parameter consists of a single value. The `getNamed()` method returns a `Map<String, String>` that contains the key-value pairs of the name parameters. The `getUnnamed()` method returns a `List<String>` where each element is an unnamed parameter value.

You pass only named and unnamed parameters to a JavaFX application. You do not pass raw type parameters. The JavaFX runtime makes all parameters, named and unnamed, passed to an application available as a `List<String>` through the `getRaw()` method of the `Parameters` class. The following discussion will make the distinction between the returned values from the three methods clear.

The `getParameters()` method of the `Application` class returns the reference of the `Application.Parameters` class. The reference to the `Parameters` class is available in the `init()` method of the `Application` class and the code that executes afterward. The parameters are not available in the constructor of the application as it is called before the `init()` method. Calling the `getParameters()` method in the constructor returns `null`.

The program in Listing 1-7 reads all types of parameters passed to the application and displays them in a `TextArea`. A `TextArea` is a UI node that displays multiple lines of text.

### ***Listing 1-7.*** Accessing Parameters Passed to a JavaFX Application

```
// FXParamApp.java
package com.jdojo.intro;

import java.util.List;
import java.util.Map;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class FXParamApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Get application parameters
        Parameters p = this.getParameters();
        Map<String, String> namedParams = p.getNamed();
        List<String> unnamedParams = p.getUnnamed();
        List<String> rawParams = p.getRaw();

        String paramStr = "Named Parameters: " + namedParams
+ "\n" +
                "Unnamed Parameters: " + unnamedParams + "\n" +
                "Raw Parameters: " + rawParams;

        TextArea ta = new TextArea(paramStr);
        Group root = new Group(ta);
        stage.setScene(new Scene(root));
        stage.setTitle("Application Parameters");
        stage.show();
    }
}
```

Let's look at a few cases of passing the parameters to the `FXParamApp` class. The output mentioned in the following cases is displayed in the `TextArea` control in the window when you run the `FXParamApp` class.

#### Case 1

The class is run as a standalone application using the following command:

```
java com.jdojo.stage.FXParamApp Anna Lola
```

The above command passes no named parameters and two unnamed parameters: Anna and Lola. The list of the raw parameters will contain the two unnamed parameters. The output will be as shown:

```
Named Parameters: {}
Unnamed Parameters: [Anna, Lola]
Raw Parameters: [Anna, Lola]
```

## Case 2

The class is run as a standalone application using the command:

```
java com.jdojo.stage.FXParamApp Anna Lola width=200 height=100
```

The above command passes no named parameters even though it seems that the last two parameters would be passed as named parameters. Using an equals (=) sign in a parameter value on the command line does not make the parameter a named parameter. The next case explains how to pass named parameters from the command line.

It passes four unnamed parameters: Anna, Lola, width=200, and height=100. The list of the raw parameters will contain the four unnamed parameters. The output will be as shown:

```
Named Parameters: {}
Unnamed Parameters: [Anna, Lola, width=200, height=100]
Raw Parameters: [Anna, Lola, width=200, height=100]
```

## Case 3

To pass a named parameter from the command line, you need to precede the parameter with exactly two hyphens (--). That is, a named parameter should be entered in the form:

```
--key=value
```

The class is run as a standalone application using the command:

```
java com.jdojo.stage.FXParamApp Anna Lola --width=200 --
height=100
```

The above command passes two named parameters: width=200 and height=100. It passes two unnamed parameters: Anna and Lola. The list of the raw parameters will contain four elements: two named parameters and two unnamed parameters. Named parameter values in the raw parameter list are preceded by two hyphens. The output will be as shown:

```
Named Parameters: {height=100, width=200}
Unnamed Parameters: [Anna, Lola]
Raw Parameters: [Anna, Lola, --width=200, --height=100]
```

## Case 4

The class `FXParamApp` is run as an applet or a WebStart application. In these cases, you have different ways to specify the named and unnamed parameters. However, they are accessed inside the application in the same way. Note that when a named parameter is accessed using the `getRaw()` method, it is preceded by two hyphens. However, you do not add two hyphens before a named parameter when you specify it in web and WebStart deployment files.

The partial content of a JNLP file to start the `FXParamApp` application using WebStart is shown below. It specifies two named and two unnamed parameters:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" xmlns:jfx="http://javafx.com"
 href="FX_NetBeans_Only.jnlp">
...
    <jfx: javafx-desc ... >
        <fx:param name="width" value="200"/>
        <fx:param name="height" value="100"/>
        <fx:argument>Anna</fx:argument>
        <fx:argument>Lola</fx:argument>
    </jfx: javafx-desc>
</jnlp>
```

## Launching a JavaFX Application

Earlier I touched on the topic of launching the JavaFX application while developing the JavaFX first application. This section gives more details on launching a JavaFX application.

Every JavaFX application class inherits from the `Application` class. The `Application` class is in the `javafx.application` package. It contains a static `launch()` method. Its sole purpose is to launch a JavaFX application. It is an overloaded method with the following two variants:

- `static void launch(Class<? extends Application> appClass, String... args)`
- `static void launch(String... args)`

Notice that you do not create an object of your JavaFX application class to launch it. The JavaFX runtime creates an object of your application class when the `launch()` method is called.

**Tip** Your JavaFX application class must have a no-args constructor, otherwise a runtime exception will be thrown when an attempt is made to launch it.

The first variant of the `launch()` method is clear. You pass the class reference of your application class as the first argument, and the `launch()` method will create an object of that class. The second argument is comprised of the command-line arguments passed to the application. The following snippet of code shows how to use the first variant of the `launch()` method:

```
public class MyJavaFXApp extends Application {  
    public static void main(String[] args) {  
        Application.launch(MyJavaFXApp.class, args);  
    }  
  
    // More code goes here  
}
```

The class reference passed to the `launch()` method does not have to be of the same class from which the method is called. For example, the following snippet of code launches the `MyJavaFXApp` application class from the `MyAppLauncher` class, which does not extend the `Application` class:

```
public class MyAppLauncher {  
    public static void main(String[] args) {  
        Application.launch(MyJavaFXApp.class, args);  
    }  
  
    // More code goes here  
}
```

The second variant of the `launch()` method takes only one argument, which is the command-line argument passed to the application. Which JavaFX application class does it use to launch the application? It attempts to find the application class name based on the caller. It checks the class name of the code that calls it. If the method is called as part of the code for a class that inherits from the `Application` class, directly or indirectly, that class is used to launch the JavaFX application. Otherwise, a runtime exception is thrown. Let's look at some examples to make this rule clear.

In the following snippet of code, the `launch()` method detects that it is called from the `main()` method of the `MyJavaFXApp` class.

The `MyJavaFXApp` class inherits from the `Application` class.

Therefore, the `MyJavaFXApp` class is used as the application class:

```
public class MyJavaFXApp extends Application {  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
  
    // More code goes here  
}
```

In the following snippet of code, the `launch()` method is called from the `main()` method of the `Test` class. The `Test` does not inherit from the `Application` class. Therefore, a runtime exception is thrown, as shown in the output below the code:

```
public class Test {
    public static void main(String[] args) {
        Application.launch(args);
    }

    // More code goes here
}
Exception in thread "main" java.lang.RuntimeException: Error:
class Test is not a subclass of javafx.application.Application
at
javafx.application.Application.launch(Application.java:211)
at Test.main(Test.java)
```

In the following snippet of code, the `launch()` method detects that it is called from the `run()` method of the `MyJavaFXApp$1` class. Note that `MyJavaFXApp$1` class is an anonymous inner class generated by the compiler, which is a subclass of the `Object` class, not the `Application` class, and it implements the `Runnable` interface. Because the call to the `launch()` method is contained within the `MyJavaFXApp$1` class, which is not a subclass of the `Application` class, a runtime exception is thrown, as shown in the output that follows the code:

```
public class MyJavaFXApp extends Application {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                Application.launch(args);
            }
        });
        t.start();
    }

    // More code goes here
}
Exception in thread "Thread-0" java.lang.RuntimeException: Error:
class MyJavaFXApp$1 is not a subclass of
javafx.application.Application
at
javafx.application.Application.launch(Application.java:211)
at MyJavaFXApp$1.run(MyJavaFXApp.java)
at java.lang.Thread.run(Thread.java:722)
```

Now that you know how to launch a JavaFX application, it's time to learn the best practice in launching a JavaFX application: limit the code

in the `main()` method to only one statement that launches the application, as shown in the following code:

```
public class MyJavaFXApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);

        // Do not add any more code in this method
    }

    // More code goes here
}
```

**Tip** The `launch()` method of the `Application` class must be called only once, otherwise, a runtime exception is thrown. The call to the `launch()` method blocks until the application is terminated. It is not always necessary to have a `main()` method to launch a JavaFX application. A JavaFX packager synthesizes one for you. For example, when you use the NetBeans IDE, you do not need to have a `main()` method, and if you have one, NetBeans ignores it.

## The Life Cycle of a JavaFX Application

JavaFX runtime creates several threads. At different stages in the application, threads are used to perform different tasks. In this section, I will only explain those threads that are used to call methods of the `Application` class during its life cycle. The JavaFX runtime creates, among other threads, two threads:

- JavaFX-Launcher
- JavaFX Application Thread

The `launch()` method of the `Application` class creates these threads. During the lifetime of a JavaFX application, the JavaFX runtime calls the following methods of the specified JavaFX Application class in order:

- The no-args constructor
- The `init()` method
- The `start()` method
- The `stop()` method

The JavaFX runtime creates an object of the specified `Application` class on the JavaFX Application Thread. The JavaFX Launcher Thread calls the `init()` method of the specified `Application` class. The `init()` method implementation in the `Application` class is empty. You can override this method in your

application class. It is not allowed to create a Stage or a Scene on the JavaFX Launcher Thread. They must be created on the JavaFX Application Thread. Therefore, you cannot create a Stage or a Scene inside the `init()` method. Attempting to do so throws a runtime exception. It is fine to create UI controls, for example, buttons or shapes.

The JavaFX Application Thread calls the `start(Stage stage)` method of the specified Application class. Note that the `start()` method in the Application class is declared abstract, and you must override this method in your application class.

At this point, the `launch()` method waits for the JavaFX application to finish. When the application finishes, the JavaFX Application Thread calls the `stop()` method of the specified Application class. The default implementation of the `stop()` method is empty in the Application class. You will have to override this method in your application class to perform your logic when your application stops.

The code in Listing 1-8 illustrates the life cycle of a JavaFX application. It displays an empty stage. You will see the first three lines of the output when the stage is shown. You will need to close the stage to see the last line of the output.

### ***Listing 1-8.*** The Life Cycle of a JavaFX Application

```
// FXLifeCycleApp.java
package com.jdojo.intro;

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

public class FXLifeCycleApp extends Application {
    public FXLifeCycleApp() {
        String name = Thread.currentThread().getName();
        System.out.println("FXLifeCycleApp() constructor: " +
+ name);
    }

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

    @Override
    public void init() {
        String name = Thread.currentThread().getName();
        System.out.println("init() method: " + name);
    }
}
```

```

@Override
public void start(Stage stage) {
    String name = Thread.currentThread().getName();
    System.out.println("start() method: " + name);

    Scene scene = new Scene(new Group(), 200, 200);
    stage.setScene(scene);
    stage.setTitle("JavaFX Application Life Cycle");
    stage.show();
}

@Override
public void stop() {
    String name = Thread.currentThread().getName();
    System.out.println("stop() method: " + name);
}
}

FXLifeCycleApp() constructor: JavaFX Application Thread
init() method: JavaFX-Launcher
start() method: JavaFX Application Thread
stop() method: JavaFX Application Thread

```

## Terminating a JavaFX Application

A JavaFX application may be terminated explicitly or implicitly. You can terminate a JavaFX application explicitly by calling the `Platform.exit()` method. When this method is called, after or from within the `start()` method, the `stop()` method of the `Application` class is called, and then the JavaFX Application Thread is terminated. At this point, if there are only daemon threads running, the JVM will exit. If this method is called from the constructor or the `init()` method of the `Application` class, the `stop()` method may not be called.

**Tip** A JavaFX application may be run in web browsers. Calling the `Platform.exit()` method in web environments may not have any effect.

A JavaFX application may be terminated implicitly, when the last window is closed. This behavior can be turned on and turned off using the static `setImplicitExit(boolean implicitExit)` method of the `Platform` class. Passing `true` to this method turns this behavior on. Passing `false` to this method turns this behavior off. By default, this behavior is turned on. This is the reason that in most of the examples so far, applications were terminated when you closed the windows. When this behavior is turned on, the `stop()` method of the `Application` class is called before terminating the JavaFX Application Thread. Terminating the JavaFX Application Thread does

not always terminate the JVM. The JVM terminates if all running nondaemon threads terminate. If the implicit terminating behavior of the JavaFX application is turned off, you must call the `exit()` method of the `Platform` class to terminate the application.

## Summary

JavaFX is an open source Java-based GUI framework that is used to develop rich client applications. It is the successor of Swing in the arena of GUI development technology on the Java platform.

The GUI in JavaFX is shown in a stage. A stage is an instance of the `Stage` class. A stage is a window in a desktop application and an area in the browser in a web application. A stage contains a scene. A scene contains a group of nodes (graphics) arranged in a tree-like structure.

A JavaFX application inherits from the `Application` class. The JavaFX runtime creates the first stage called the primary stage and calls the `start()` method of the application class passing the reference of the primary stage. The developer needs to add a scene to the stage and make the stage visible inside the `start()` method.

You can launch a JavaFX application using the `launch()` method of the `Application` class. If you run a Java class that inherits from the `application` class, which would be a JavaFX application class, the `java` command automatically launches the JavaFX application for you.

During the lifetime of a JavaFX application, the JavaFX runtime calls predefined methods of the JavaFX `Application` class in a specific order. First, the `no-args` constructor of the class is called, followed by calls to the `init()` and `start()` methods. When the application terminates, the `stop()` method is called.

You can terminate a JavaFX application by calling the `Platform.exit()` method. Calling the `Platform.exit()` method when the application is running in a web browser as an applet may not have any effects.

The next chapter will introduce you to properties and binding in JavaFX.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

## CHAPTER 2



### Properties and Bindings

In this chapter, you will learn:

- What a property is in JavaFX
- How to create a property object and use it
- The class hierarchy of properties in JavaFX
- How to handle the invalidation and change events in a property object
- What a binding is in JavaFX and how to use unidirectional and bidirectional bindings
- About the high-level and low-level binding API in JavaFX

This chapter discusses the properties and binding support in Java and JavaFX. If you have experience using the JavaBeans API for properties and binding, you can skip the first few sections, which discuss the properties and binding support in Java, and start with the section “Understanding Properties in JavaFX.”

#### What Is a Property?

A Java class can contain two types of members: *fields* and *methods*. Fields represent the state of objects and they are declared private. Public methods, known as *accessors*, or *getters* and *setters*, are used to read and modify private fields. In simple terms, a Java class that has public accessors, for all or part of its private fields, is known as a Java *bean*, and the accessors define the properties of the bean. Properties of a Java bean allow users to customize its state, behavior, or both.

Java beans are observable. They support property change notification. When a public property of a Java bean changes, a notification is sent to all interested listeners.

In essence, Java beans define reusable components that can be assembled by a builder tool to create a Java application. This opens the door for third parties to develop Java beans and make them available to others for reuse.

A property can be read-only, write-only, or read/write. A ready-only property has a getter but no setter. A write-only property has a setter but no getter. A read/write property has a getter and a setter.

Java IDEs and other builder tools (e.g., a GUI layout builder), use introspection to get the list of properties of a bean and let you manipulate those properties at design time. A Java bean can be visual or nonvisual. Properties of a bean can be used in a builder tool or programmatically.

The JavaBeans API provides a class library, through the `java.beans` package, and naming conventions to create and use Java beans. The following is an example of a `Person` bean with a `read/write` `name` property. The `getName()` method (the getter) returns the value of the `name` field. The `setName()` method (the setter) sets the value of the `name` field:

```
// Person.java
package com.jdojo.binding;

public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

By convention, the names of the getter and setter methods are constructed by appending the name of the property, with the first letter in uppercase, to the words *get* and *set*, respectively. The getter method should not take any parameters, and its return type should be the same as the type of the field. The setter method should take a parameter whose type should be the same as the type of the field, and its returns type should be `void`.

The following snippet of code manipulates the `name` property of a `Person` bean programmatically:

```
Person p = new Person();
p.setName("John Jacobs");
String name = p.getName();
```

Some object-oriented programming languages, for example, C#, provide a third type of class member known as a *property*. A property is used to read, write, and compute the value of a private field from outside the class. C# lets you declare a `Person` class with a `Name` property as follows:

```
// C# version of the Person class
public class Person {
    private string name;

    public string Name {
```

```

        get { return name; }
        set { name = value; }
    }
}

```

In C#, the following snippet of code manipulates the `name` private field using the `Name` property; it is equivalent to the previously shown Java version of the code:

```

Person p = new Person();
p.Name = "John Jacobs";
string name = p.Name;

```

If the accessors of a property perform the routine work of returning and setting the value of a field, C# offers a compact format to define such a property. You do not even need to declare a private field in this case. You can rewrite the `Person` class in C# as shown here:

```

// C# version of the Person class using the compact format
public class Person {
    public string Name { get; set; }
}

```

So, what is a property? A *property* is a publicly accessible attribute of a class that affects its state, behavior, or both. Even though a property is publicly accessible, its use (read/write) invokes methods that hide the actual implementation to access the data. Properties are observable, so interested parties are notified when its value changes.

**Tip** In essence, properties define the public state of an object that can be read, written, and observed for changes. Unlike other programming languages, such as C#, properties in Java are not supported at the language level. Java support for properties comes through the JavaBeans API and design patterns. For more details on properties in Java, please refer to the JavaBeans specification, which can be downloaded from <http://www.oracle.com/technetwork/java/javase/overview/spec-136004.html>.

Apart from simple properties, such as the `name` property of the `Person` bean, Java also supports *indexed*, *bound*, and *constrained* properties. An indexed property is an array of values that are accessed using indexes. An indexed property is implemented using an array data type. A bound property sends a notification to all listeners when it is changed. A constrained property is a bound property in which a listener can veto a change.

## What Is a Binding?

In programming, the term *binding* is used in many different contexts. Here I want to define it in the context of *data binding*. Data binding defines a relation between data elements (usually variables) in a program

to keep them synchronized. In a GUI application, data binding is frequently used to synchronize the elements in the data model with the corresponding UI elements.

Consider the following statement, assuming that x, y, and z are numeric variables:

```
x = y + z;
```

The above statement defines a binding between x, y, and z. When it is executed, the value of x is synchronized with the sum of y and z. A binding also has a time factor. In the above statement, the value of x is bound to the sum of y and z and is valid at the time the statement is executed. The value of x may not be the sum of y and z before and after the above statement is executed.

Sometimes it is desired for a binding to hold over a period. Consider the following statement that defines a binding

using `listPrice`, `discounts`, and `taxes`:

```
soldPrice = listPrice - discounts + taxes;
```

For this case, you would like to keep the binding valid forever, so the sold price is computed correctly, whenever `listPrice`, `discounts`, or `taxes` change.

In the above binding, `listPrice`, `discounts`, and `taxes` are known as *dependencies*, and it is said that `soldPrice` is bound to `listPrice`, `discounts`, and `taxes`.

For a binding to work correctly, it is necessary that the binding is notified whenever its dependencies change. Programming languages that support binding provide a mechanism to register listeners with the dependencies. When dependencies become invalid or they change, all listeners are notified. A binding may synchronize itself with its dependencies when it receives such notifications.

A binding can be an *eager binding* or a *lazy binding*. In an eager binding, the bound variable is recomputed immediately after its dependencies change. In a lazy binding, the bound variable is not recomputed when its dependencies change. Rather, it is recomputed when it is read the next time. A lazy binding performs better compared to an eager binding.

A binding may be *unidirectional* or *bidirectional*. A unidirectional binding works only in one direction; changes in the dependencies are propagated to the bound variable. A bidirectional binding works in both directions. In a bidirectional binding, the bound variable and the dependency keep their values synchronized with each other. Typically, a bidirectional binding is defined only between two variables. For example, a bidirectional binding,  $x = y$  and  $y = x$ , declares that the values of x and y are always the same.

Mathematically, it is not possible to define a bidirectional binding between multiple variables uniquely. In the above example, the sold price binding is a unidirectional binding. If you want to make it a bidirectional binding, it is not uniquely possible to compute the values of the list price, discounts, and taxes when the sold price is changed. There are an infinite number of possibilities in the other direction.

Applications with GUIs provide users with UI widgets, for example, text fields, check boxes, and buttons, to manipulate data. The data displayed in UI widgets have to be synchronized with the underlying data model and vice versa. In this case, a bidirectional binding is needed to keep the UI and the data model synchronized.

## Understanding Bindings Support in JavaBeans

Before I discuss Java FX properties and binding, let's take a short tour of binding support in the JavaBeans API. You may skip this section if you have used the JavaBeans API before.

Java has supported binding of bean properties since its early releases. Listing 2-1 shows an Employee bean with two properties, name and salary.

***Listing 2-1.*** An Employee Java Bean with Two Properties Named name and salary

```
// Employee.java
package com.jdojo.binding;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Employee {
    private String name;
    private double salary;
    private PropertyChangeSupport pcs = new
PropertyChangeSupport(this);

    public Employee() {
        this.name = "John Doe";
        this.salary = 1000.0;
    }

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }
}
```

```

public void setName(String name) {
    this.name = name;
}

public double getSalary() {
    return salary;
}

public void setSalary(double newSalary) {
    double oldSalary = this.salary;
    this.salary = newSalary;

    // Notify the registered listeners about the change
    pcs.firePropertyChange("salary", oldSalary,
newSalary);
}

public void
addPropertyChangeListener(PropertyChangeListener listener) {
    pcs.addPropertyChangeListener(listener);
}

public void
removePropertyChangeListener(PropertyChangeListener listener) {
    pcs.removePropertyChangeListener(listener);
}

@Override
public String toString() {
    return "name = " + name + ", salary = " + salary;
}
}

```

Both properties of the Employee bean are read/write. The salary property is also a bound property. Its setter generates property change notifications when the salary changes.

Interested listeners can register or deregister for the change notifications using

the addPropertyChangeListener() and removePropertyChangeListener() methods. The PropertyChangeSupport class is part of the JavaBeans API that facilitates the registration and removal of property change listeners and firing of the property change notifications.

Any party interested in synchronizing values based on the salary change will need to register with the Employee bean and take necessary actions when it is notified of the change.

**Listing 2-2** shows how to register for salary change notifications for an Employee bean. The output below it shows that salary change notification is fired only twice, whereas the setSalary() method is called three times. This is true because the second call to the setSalary() method uses the same salary amount as the first call

and the `PropertyChangeSupport` class is smart enough to detect that. The example also shows how you would bind variables using the JavaBeans API. The tax for an employee is computed based on a tax percentage. In the JavaBeans API, property change notifications are used to bind the variables.

***Listing 2-2.*** An `EmployeeTest` Class that Tests the Employee Bean for Salary Changes

```
// EmployeeTest.java
package com.jdojo.binding;

import java.beans.PropertyChangeEvent;

public class EmployeeTest {
    public static void main(String[] args) {
        final Employee e1 = new Employee("John Jacobs",
2000.0);

        // Compute the tax
        computeTax(e1.getSalary());

        // Add a property change listener to e1
        e1.addPropertyChangeListener(EmployeeTest::handlePro
pertyChange);

        // Change the salary
        e1.setSalary(3000.00);
        e1.setSalary(3000.00); // No change notification is
sent.
        e1.setSalary(6000.00);
    }

    public static void handlePropertyChange(PropertyChangeEvent
e) {
        String propertyName = e.getPropertyName();

        if ("salary".equals(propertyName)) {
            System.out.print("Salary has changed. ");
            System.out.print("Old:" + e.getOldValue());
            System.out.println(", New:"
+ e.getNewValue());
            computeTax((Double)e.getNewValue());
        }
    }

    public static void computeTax(double salary) {
        final double TAX_PERCENT = 20.0;
        double tax = salary * TAX_PERCENT/100.0;
        System.out.println("Salary:" + salary + ", Tax:"
+ tax);
    }
}
```

```
Salary:2000.0, Tax:400.0
Salary has changed. Old:2000.0, New:3000.0
Salary:3000.0, Tax:600.0
Salary has changed. Old:3000.0, New:6000.0
Salary:6000.0, Tax:1200.0
```

## Understanding Properties in JavaFX

JavaFX supports properties, events, and binding through *properties* and *binding* APIs. Properties support in JavaFX is a huge leap forward from the JavaBeans properties.

All properties in JavaFX are observable. They can be observed for invalidation and value changes. There can be read/write or read-only properties. All read/write properties support binding.

In JavaFX, a property can represent a value or a collection of values. This chapter covers properties that represent a single value. I will cover properties representing a collection of values in Chapter 3.

In JavaFX, properties are objects. There is a property class hierarchy for each type of property. For example, the `IntegerProperty`, `DoubleProperty`, and `StringProperty` classes represent properties of `int`, `double`, and `String` types, respectively. These classes are abstract. There are two types of implementation classes for them: one to represent a read/write property and one to represent a wrapper for a read-only property. For example, the `SimpleDoubleProperty` and `ReadOnlyDoubleWrapper` classes are concrete classes whose objects are used as read/write and read-only double properties, respectively.

Below is an example of how to create an `IntegerProperty` with an initial value of 100:

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

Property classes provide two pairs of getter and setter methods: `get()`/`set()` and `getValue()`/`setValue()`. The `get()` and `set()` methods get and set the value of the property, respectively. For primitive type properties, they work with primitive type values. For example, for `IntegerProperty`, the return type of the `get()` method and the parameter type of the `set()` method are `int`. The `getValue()` and `setValue()` methods work with an object type; for example, their return type and parameter type are `Integer` for `IntegerProperty`.

**Tip** For reference type properties, such as `StringProperty` and `ObjectProperty<T>`, both pairs of getter and setter work with an object type. That is,

both `get()` and `getValue()` methods of `StringProperty` return a `String`, and `set()` and `setValue()` methods take a `String` parameter. With autoboxing for primitive types, it does not matter which version of getter and setter is used. The `getValue()` and `setValue()` methods exist to help you write generic code in terms of object types.

The following snippet of code uses an `IntegerProperty` and its `get()` and `set()` methods. The counter property is a read/write property as it is an object of the `SimpleIntegerProperty` class:

```
IntegerProperty counter = new SimpleIntegerProperty(1);
int counterValue = counter.get();
System.out.println("Counter:" + counterValue);

counter.set(2);
counterValue = counter.get();
System.out.println("Counter:" + counterValue);
Counter:1
Counter:2
```

Working with read-only properties is a bit tricky.

A `ReadOnlyXXXWrapper` class wraps two properties of `XXX` type: one read-only and one read/write. Both properties are synchronized. Its `getReadOnlyProperty()` method returns a `ReadOnlyXXXProperty` object.

The following snippet of code shows how to create a read-only `Integer` property. The `idWrapper` property is read/write, whereas the `id` property is read-only. When the value in `idWrapper` is changed, the value in `id` is changed automatically:

```
ReadOnlyIntegerWrapper idWrapper = new
ReadOnlyIntegerWrapper(100);
ReadOnlyIntegerProperty id = idWrapper.getReadOnlyProperty();

System.out.println("idWrapper:" + idWrapper.get());
System.out.println("id:" + id.get());

// Change the value
idWrapper.set(101);

System.out.println("idWrapper:" + idWrapper.get());
System.out.println("id:" + id.get());
idWrapper:100
id:100
idWrapper:101
id:101
```

**Tip** Typically, a wrapper property is used as a private instance variable of a class. The class can change the property internally. One of its methods returns the read-only property object of the wrapper class, so the same property is read-only for the outside world.

You can use seven types of properties that represent a single value. The base classes for those properties are named as `XXXProperty`, read-only base classes are named as `ReadOnlyXXXProperty`, and wrapper classes are named as `ReadOnlyXXXWrapper`. The values for `XXX` for each type are listed in Table 2-1.

**Table 2-1.** List of Property Classes that Wrap a Single Value

Type	XXX Value
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>String</code>	<code>String</code>
<code>Object</code>	<code>Object</code>

A property object wraps three pieces of information:

- The reference of the bean that contains it
- A name
- A value

When you create a property object, you can supply all or none of the above three pieces of information. Concrete property classes, named like `SimpleXXXProperty` and `ReadOnlyXXXWrapper`, provide four constructors that let you supply combinations of the three pieces of information. The following are the constructors for the `SimpleIntegerProperty` class:

```
SimpleIntegerProperty()
SimpleIntegerProperty(int initialValue)
SimpleIntegerProperty(Object bean, String name)
SimpleIntegerProperty(Object bean, String name, int initialValue)
```

The default value for the initial value depends on the type of the property. It is zero for numeric types, `false` for boolean types, and `null` for reference types.

A property object can be part of a bean or it can be a standalone object. The specified `bean` is the reference to the bean object that contains the property. For a standalone property object, it can be `null`. Its default value is `null`.

The name of the property is its name. If not supplied, it defaults to an empty string.

The following snippet of code creates a property object as part of a bean and sets all three values. The first argument to the constructor of the `SimpleStringProperty` class is `this`, which is the reference of the `Person` bean, the second argument—"name"—is the name of the property, and the third argument—"Li"—is the value of the property:

```
public class Person {
    private StringProperty name = new SimpleStringProperty(this, "name", "Li"); // More code goes here...
}
```

Every property class has `getBean()` and `getName()` methods that return the bean reference and the property name, respectively.

## Using Properties in JavaFX Beans

In the previous section, you saw the use of JavaFX properties as standalone objects. In this section, you will use them in classes to define properties. Let's create a `Book` class with three properties: `ISBN`, `title`, and `price`, which will be modeled using JavaFX properties classes.

In JavaFX, you do not declare the property of a class as one of the primitive types. Rather, you use one of the JavaFX property classes. The `title` property of the `Book` class will be declared as follows. It is declared `private` as usual:

```
public class Book {
    private StringProperty title = new SimpleStringProperty(this, "title", "Unknown");
}
```

You declare a public getter for the property, which is named, by convention, as `XXXProperty`, where `XXX` is the name of the property. This getter returns the reference of the property. For our `title` property, the getter will be named `titleProperty` as shown below:

```
public class Book {
    private StringProperty title = new SimpleStringProperty(this, "title", "Unknown");

    public final StringProperty titleProperty() {
```

```

        return title;
    }
}

```

The above declaration of the `Book` class is fine to work with the `title` property, as shown in the following snippet of code that sets and gets the title of a book:

```

Book b = new Book();
b.titleProperty().set("Harnessing JavaFX 8.0");
String title = b.titleProperty().get();

```

According to the JavaFX design patterns, and not for any technical requirements, a JavaFX property has a getter and a setter that are similar to the getters and setters in JavaBeans. The return type of the getter and the parameter type of the setter are the same as the type of the property value. For example,

for `StringProperty` and `IntegerProperty`, they will be `String` and `int`, respectively.

The `getTitle()` and `setTitle()` methods for the `title` property are declared as follows:

```

public class Book {
    private StringProperty title = new
SimpleStringProperty(this, "title", "Unknown");

    public final StringProperty titleProperty() {
        return title;
    }

    public final String getTitle() {
        return title.get();
    }

    public final void setTitle(String title) {
        this.title.set(title);
    }
}

```

Note that the `getTitle()` and `setTitle()` methods use the `title` property object internally to get and set the title value.

**Tip** By convention, getters and setters for a property of a class are declared `final`. Additional getters and setters, using JavaBeans naming convention, are added to make the class interoperable with the older tools and frameworks that use the old JavaBeans naming conventions to identify the properties of a class.

The following snippet of code shows the declaration of a read-only `ISBN` property for the `Book` class:

```

public class Book {
    private ReadOnlyStringWrapper ISBN = new
ReadOnlyStringWrapper(this, "ISBN", "Unknown");

```

```

        public final String getISBN() {
            return ISBN.get();
        }

        public final ReadOnlyStringProperty ISBNProperty() {
            return ISBN.getReadOnlyProperty();
        }

        // More code goes here...
    }
}

```

Notice the following points about the declaration of the read-only ISBN property:

- It uses the `ReadOnlyStringWrapper` class instead of the `SimpleStringProperty` class.
- There is no setter for the property value. You may declare one; however, it must be private.
- The getter for the property value works the same as for a read/write property.
- The `ISBNProperty()` method uses `ReadOnlyStringProperty` as the return type, not `ReadOnlyStringWrapper`. It obtains a read-only version of the property object from the wrapper object and returns the same.

For the users of the `Book` class, its `ISBN` property is read-only. However, it can be changed internally, and the change will be reflected in the read-only version of the property object automatically.

**Listing 2-3** shows the complete code for the `Book` class.

### **Listing 2-3.** A Book Class with Two Read/Write and a Read-Only Properties

```

// Book.java
package com.jdojo.binding;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.ReadOnlyStringProperty;
import javafx.beans.property.ReadOnlyStringWrapper;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Book {
    private StringProperty title = new
SimpleStringProperty(this, "title", "Unknown");
    private DoubleProperty price = new
SimpleDoubleProperty(this, "price", 0.0);
}

```

```
private ReadOnlyStringWrapper ISBN = new
ReadOnlyStringWrapper(this, "ISBN", "Unknown");

public Book() {
}

public Book(String title, double price, String ISBN) {
    this.title.set(title);
    this.price.set(price);
    this.ISBN.set(ISBN);
}

public final String getTitle() {
    return title.get();
}

public final void setTitle(String title) {
    this.title.set(title);
}

public final StringProperty titleProperty() {
    return title;
}

public final double getPrice() {
    return price.get();
}

public final void setPrice(double price) {
    this.price.set(price);
}

public final DoubleProperty priceProperty() {
    return price;
}

public final String getISBN() {
    return ISBN.get();
}

public final ReadOnlyStringProperty ISBNProperty() {
    return ISBN.getReadOnlyProperty();
}
}
```

**Listing 2-4** tests the properties of the `Book` class. It creates a `Book` object, prints the details, changes some properties, and prints the details again. Note the use of the `ReadOnlyProperty` parameter type for the `printDetails()` method. All property classes implement, directly or indirectly, the `ReadOnlyProperty` interface.

The `toString()` methods of the property implementation classes return a well-formatted string that contains all relevant pieces of information for a property. I did not use the `toString()` method of the

property objects because I wanted to show you the use of the different methods of the JavaFX properties.

**Listing 2-4.** A Test Class to Test Properties of the Book Class

```
// BookPropertyTest.java
package com.jdojo.binding;

import javafx.beans.property.ReadOnlyProperty;

public class BookPropertyTest {
    public static void main(String[] args) {
        Book book = new Book("Harnessing JavaFX", 9.99,
"0123456789");

        System.out.println("After creating the Book
object...");

        // Print Property details
        printDetails(book.titleProperty());
        printDetails(book.priceProperty());
        printDetails(book.ISBNProperty());

        // Change the book's properties
        book.setTitle("Harnessing JavaFX 8.0");
        book.setPrice(9.49);

        System.out.println("\nAfter changing the Book
properties...");

        // Print Property details
        printDetails(book.titleProperty());
        printDetails(book.priceProperty());
        printDetails(book.ISBNProperty());
    }

    public static void printDetails(ReadOnlyProperty<?> p) {
        String name = p.getName();
        Object value = p.getValue();
        Object bean = p.getBean();
        String beanClassName = (bean == null) ?
"null":bean.getClass().getSimpleName();
        String propClassName = p.getClass().getSimpleName();

        System.out.print(propClassName);
        System.out.print("[Name:" + name);
        System.out.print(", Bean Class:" + beanClassName);
        System.out.println(", Value:" + value + "]");
    }
}

After creating the Book object...
SimpleStringProperty[Name:title, Bean Class:Book,
Value:Harnessing JavaFX]
SimpleDoubleProperty[Name:price, Bean Class:Book, Value:9.99]
```

```
ReadOnlyPropertyImpl [Name:ISBN, Bean Class:Book,  
Value:0123456789]
```

```
After changing the Book properties...  
SimpleStringProperty [Name:title, Bean Class:Book,  
Value:Harnessing JavaFX 8.0]  
SimpleDoubleProperty [Name:price, Bean Class:Book, Value:9.49]  
ReadOnlyPropertyImpl [Name:ISBN, Bean Class:Book,  
Value:0123456789]
```

## Lazily Instantiating Property Objects

Compared to simple JavaBeans properties, JavaFX properties are more powerful. Their power comes from their observable and binding features at a price that every JavaFX property is an object. If you consider ten instances of a JavaFX class with 50 properties, you will have 500 objects in memory. However, not all properties use their advanced features. Most of them will be used as JavaBeans properties, using only getters and setters, or they will just use their default values. When it is likely that a JavaFX property will rarely use its advanced features, the property object may be instantiated lazily to optimize memory usage. The optimization comes at a price of adding a few extra lines of code.

The following are the two use cases where you can lazily instantiate a property:

- When the property will use its default value in most of the cases
- When the property will not use its observable and binding features in most cases

Consider a `Monitor` class with a `screenType` property whose default value is "flat". This falls into the first category, because most of the monitors are flat and will use the default value for the `screenType` property. Listing 2-5 shows the declaration of the `Monitor` class.

### ***Listing 2-5.*** A Monitor Class that Uses the Default Value for Its `screenType` Property Most of the Time

```
// Monitor.java  
package com.jdojo.binding;  
  
import javafx.beans.property.StringProperty;  
import javafx.beans.property.SimpleStringProperty;  
  
public class Monitor {  
    public static final String DEFAULT_SCREEN_TYPE = "flat";  
    private StringProperty screenType;
```

```

        public String getScreenType() {
            return (screenType == null) ? DEFAULT_SCREEN_TYPE
: screenType.get();
        }

        public void setScreenType(String newScreenType) {
            if (screenType != null || !DEFAULT_SCREEN_TYPE.equals(newScreenType)) {
                screenTypeProperty().set(newScreenType);
            }
        }

        public StringProperty screenTypeProperty() {
            if (screenType == null) {
                screenType = new SimpleStringProperty(this,
"screenType",
                               DEFAULT_SCREEN_TYPE);
            }
            return screenType;
        }
    }
}

```

The `Monitor` class declares a static variable `DEFAULT_SCREEN_TYPE`, which is initialized to the default value of the screen type. It declares a `StringProperty`, which is not instantiated at the time of declaration. It is instantiated later, when needed.

The `getScreenType()` method checks if the `screenType` property has been instantiated. If not, it returns the default value. Otherwise, it returns the value stored in the property object.

The `setScreenType()` method checks if the property object has already been instantiated or the property being set is other than the default value. If either one is `true`, it gets the property object using the `screenTypeProperty()` method, which will instantiate the property object, if needed, and sets the new property value.

The `screenTypeProperty()` method instantiates the property object the first time it is called.

This design for the `Monitor` class will work as intended only if its users do not call the `screenTypeProperty()` method until they really need the advanced features of the property. Consider the following snippet of code:

```

Monitor m = new Monitor();
String st = m.screenTypeProperty().get(); // Instantiates the
property object

```

The above snippet of code instantiates a `screenType` property object, even though the user only wants to get the value of the property.

The code should be rewritten as follows to delay the instantiation of the property object:

```
Monitor m = new Monitor();
String st = m.getScreenType(); // Does not instantiate the
property object
```

Properties in the second category are used without advanced features in most of the cases. Listing 2-6 shows the declaration of an `Item` class that instantiates the property object when it is needed.

### ***Listing 2-6.*** An Item Class that Rarely Uses Advanced Features of Its weight Property

```
// Item.java
package com.jdojo.binding;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class Item {
    private DoubleProperty weight;
    private double _weight = 150;

    public double getWeight() {
        return (weight == null) ? _weight : weight.get();
    }

    public void setWeight(double newWeight) {
        if (weight == null) {
            _weight = newWeight;
        } else {
            weight.set(newWeight);
        }
    }

    public DoubleProperty weightProperty() {
        if (weight == null) {
            weight = new SimpleDoubleProperty(this,
"weight", _weight);
        }
        return weight;
    }
}
```

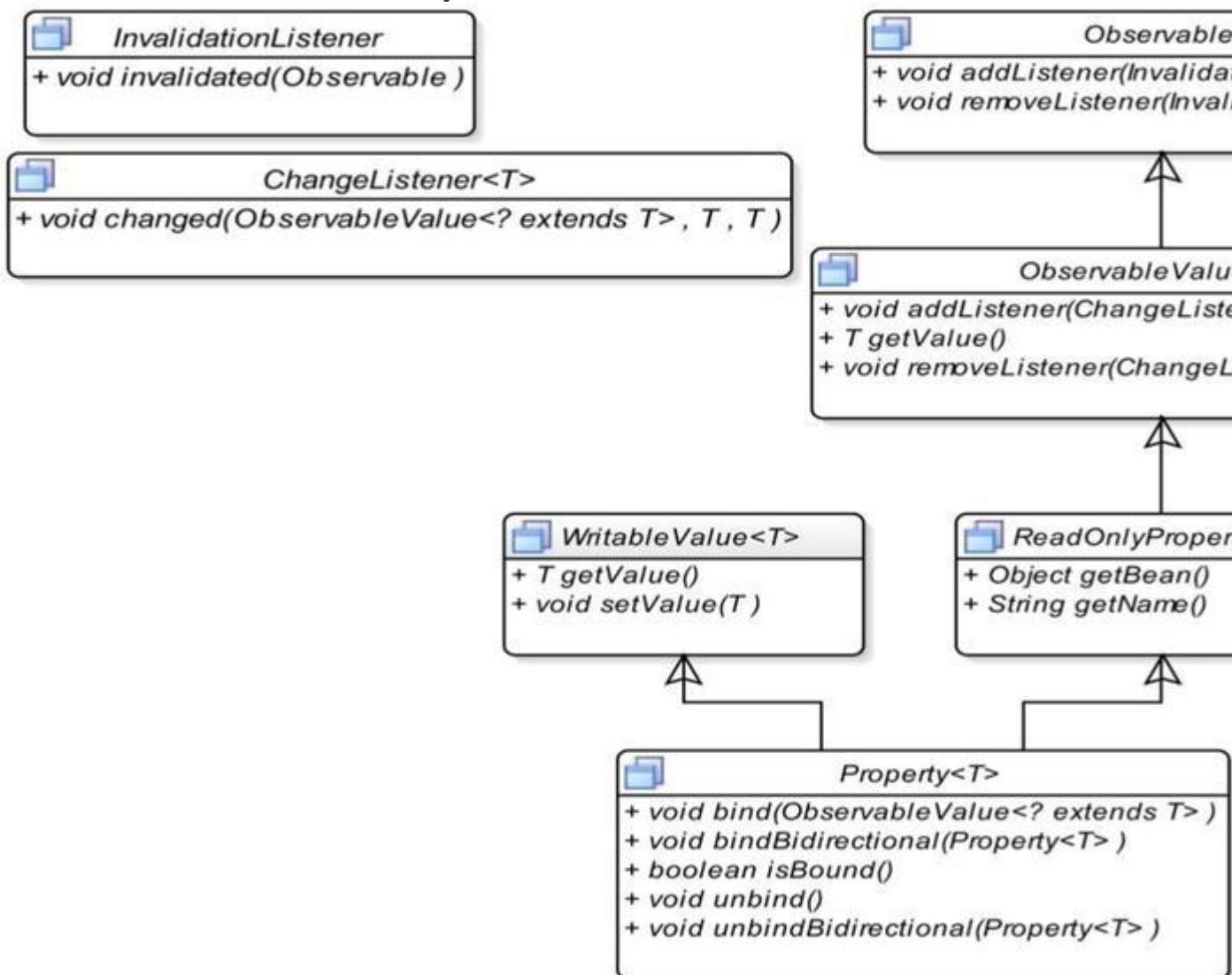
The `Item` class declares an extra variable, `_weight`, which is used to hold the value of the `weight` property until the property object is instantiated. Unlike the `Monitor` class, changing the `weightProperty` does not instantiate the property object. It is instantiated when the `weightProperty()` method is called.

**Tip** The approach used for instantiating a property object, eager or lazy, depends on the situation at hand. The fewer the number of properties in a class,

the more likely all of them will be used, and you should be fine with eager instantiation. The more the number of properties in a class, the more likely fewer of them will be used, and you should go for lazy instantiation if the performance of your application matters.

## Understanding the Property Class Hierarchy

It is important to understand a few core classes and interfaces of the JavaFX properties and binding APIs before you start using them. Figure 2-1 shows the class diagram for core interfaces of the properties API. You will not need to use these interfaces directly in your programs. Specialized versions of these interfaces and the classes that implement them exist and are used directly.



**Figure 2-1.** A class diagram for core interfaces in the JavaFX property API

Classes and interfaces in the JavaFX properties API are spread across different packages. Those packages

are `javafx.beans`, `javafx.beans.binding`, `javafx.beans.property`, and `javafx.beans.value`.

The `Observable` interface is at the top of the properties API. An `Observable` wraps content, and it can be observed for invalidations of its content. The `Observable` interface has two methods to support this. Its `addListener()` method lets you add an `InvalidationListener`. The `invalidated()` method of the `InvalidationListener` is called when the content of the `Observable` becomes invalid. An `InvalidationListener` can be removed using its `removeListener()` method.

**Tip** All JavaFX properties are observable.

An `Observable` should generate an invalidation event only when the status of its content changes from valid to invalid. That is, multiple invalidations in a row should generate only one invalidation event. Property classes in the JavaFX follow this guideline.

**Tip** The generation of an invalidation event by an `Observable` does not necessarily mean that its content has changed. All it means is that its content is invalid for some reason. For example, sorting an `ObservableList` may generate an invalidation event. Sorting does not change the contents of the list; it only reorders the contents.

The `ObservableValue` interface inherits from the `Observable` interface. An `ObservableValue` wraps a value, which can be observed for changes. It has a `getValue()` method that returns the value it wraps. It generates invalidation events and change events. Invalidation events are generated when the value in the `ObservableValue` is no longer valid. Change events are generated when the value changes. You can register a `ChangeListener` to an `ObservableValue`. The `changed()` method of the `ChangeListener` is called every time the value of its value changes. The `changed()` method receives three arguments: the reference of the `ObservableValue`, the old value, and the new value.

An `ObservableValue` can recompute its value lazily or eagerly. In a lazy strategy, when its value becomes invalid, it does not know if the value has changed until the value is recomputed; the value is recomputed the next time it is read. For example, using the `getValue()` method of an `ObservableValue` would make it recompute its value if the value was invalid and if it uses a lazy strategy. In an eager strategy, the value is recomputed as soon as it becomes invalid.

To generate invalidation events, an `ObservableValue` can use lazy or eager evaluation. A lazy evaluation is more efficient. However,

generating change events forces an `ObservableValue` to recompute its value immediately (an eager evaluation) as it has to pass the new value to the registered change listeners.

The `ReadOnlyProperty` interface adds `getBean()` and `getName()` methods. Their use was illustrated in Listing 2-4. The `getBean()` method returns the reference of the bean that contains the property object. The `getName()` method returns the name of the property. A read-only property implements this interface.

A `WritableValue` wraps a value that can be read and set using its `getValue()` and `setValue()` methods, respectively. A read/write property implements this interface.

The `Property` interface inherits from `ReadOnlyProperty` and `WritableValue` interfaces. It adds the following five methods to support binding:

- `void bind(ObservableValue<? extends T> observable)`
- `void unbind()`
- `void bindBidirectional(Property<T> other)`
- `void unbindBidirectional(Property<T> other)`
- `boolean isBound()`

The `bind()` method adds a unidirectional binding between this `Property` and the specified `ObservableValue`.

The `unbind()` method removes the unidirectional binding for this `Property`, if one exists.

The `bindBidirectional()` method creates a bidirectional binding between this `Property` and the specified `Property`. The `unbindBidirectional()` method removes a bidirectional binding.

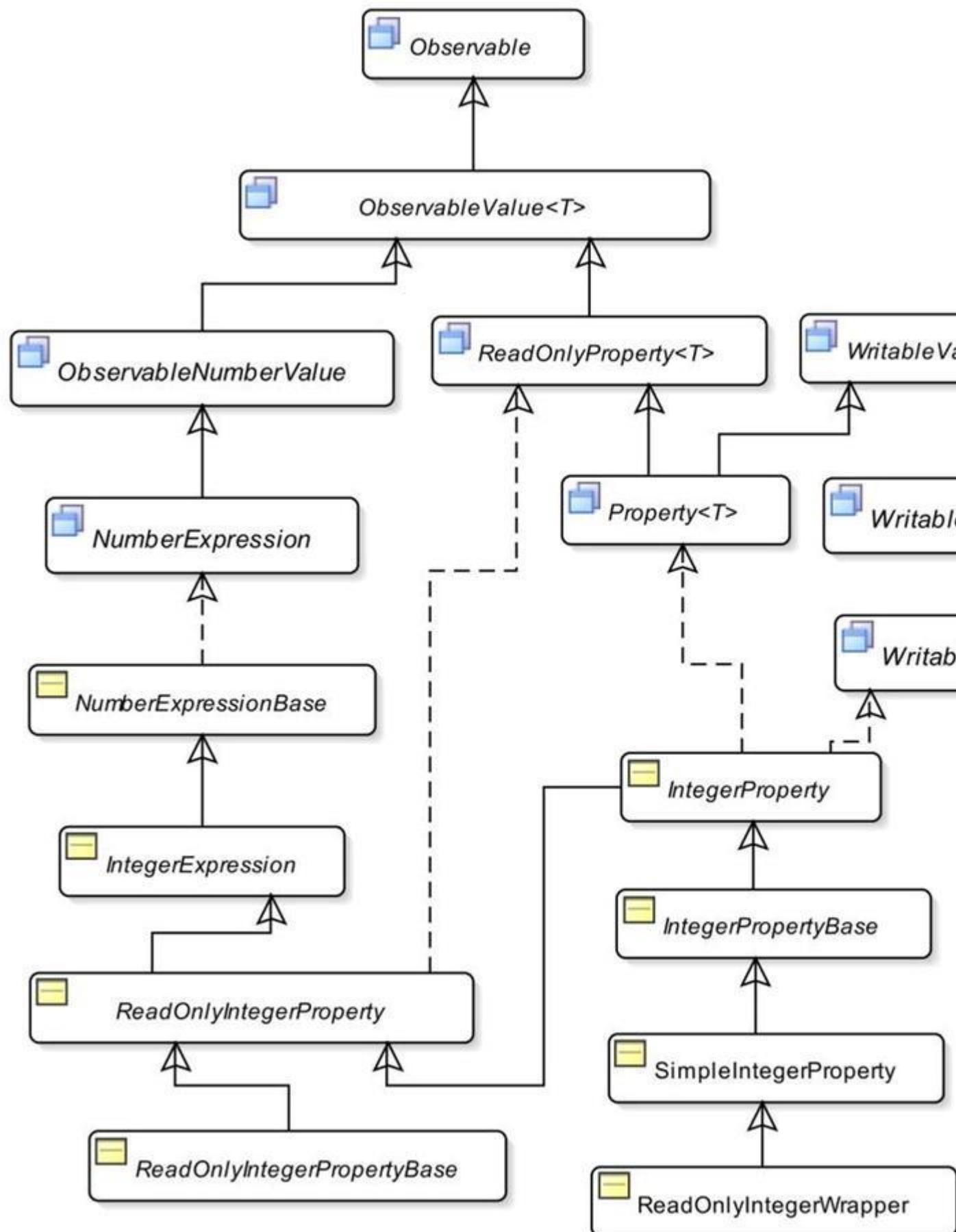
Note the difference in the parameter types for the `bind()` and `bindBidirectional()` methods. A unidirectional binding can be created between a `Property` and an `ObservableValue` of the same type as long as they are related through inheritance. However, a bidirectional binding can only be created between two properties of the same type.

The `isBound()` method returns `true` if the `Property` is bound. Otherwise, it returns `false`.

**Tip** All read/write JavaFX properties support binding.

Figure 2-2 shows a partial class diagram for the integer property in JavaFX. The diagram gives you an idea about the complexity of the

JavaFX properties API. You do not need to learn all of the classes in the properties API. You will use only a few of them in your applications.



**Figure 2-2.** A class diagram for the integer property

## Handling Property Invalidation Events

A property generates an invalidation event when the status of its value changes from valid to invalid for the first time. Properties in JavaFX use lazy evaluation. When an invalid property becomes invalid again, an invalidation event is not generated. An invalid property becomes valid when it is recomputed, for example, by calling its `get()` or `getValue()` method.

Listing 2-7 provides the program to demonstrate when invalidation events are generated for properties. The program includes enough comments to help you understand its logic. In the beginning, it creates an `IntegerProperty` named `counter`:

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

An `InvalidationListener` is added to the `counter` property:

```
counter.addListener(InvalidationTest::invalidated);
```

Note that the above statement uses a lambda expression and a method reference, which are features of Java 8. If you are not familiar with lambda expressions, you can compare the above statement to the following snippet of code, which uses an anonymous inner class:

```
import javafx.beans.InvalidationListener;
...
counter.addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable prop) {
        InvalidationTest.invalidated(prop);
    }
});
```

When you create a property object, it is valid. When you change the `counter` property to 101, it fires an invalidation event. At this point, the `counter` property becomes invalid. When you change its value to 102, it does not fire an invalidation event, because it is already invalid. When you use the `get()` method to read the `counter` value, it becomes valid again. Now you set the same value, 102, to the `counter`, which does not fire an invalidation event, as the value did not really change. The `counter` property is still valid. At the end, you change its value to a different value, and sure enough, an invalidation event is fired.

**Tip** You are not limited to adding only one invalidation listener to a property. You can add as many invalidation listeners as you need. Once you are done with an invalidation listener, make sure to remove it by calling the `removeListener()` method of the `Observable` interface; otherwise, it may lead to memory leaks. Please refer to the section “[Avoiding Memory Leaks in Listeners](#)” for more details on how to avoid memory leaks.

## ***Listing 2-7.*** Testing Invalidation Events for Properties

```
// InvalidationTest.java
package com.jdojo.binding;

import javafx.beans.Observable;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class InvalidationTest {
    public static void main(String[] args) {
        IntegerProperty counter = new SimpleIntegerProperty(100);

        // Add an invalidation listener to the counter
        property
            counter.addListener(InvalidationTest::invalidated);

        System.out.println("Before changing the counter
value-1");
        counter.set(101);
        System.out.println("After changing the counter
value-1");

        /* At this point counter property is invalid and
        further changes
            to its value will not generate invalidation
        events.
        */
        System.out.println("\nBefore changing the counter
value-2");
        counter.set(102);
        System.out.println("After changing the counter
value-2");

        // Make the counter property valid by calling its
        get() method
        int value = counter.get();
        System.out.println("Counter value = " + value);

        /* At this point counter property is valid and
        further changes
            to its value will generate invalidation events.
        */
        // Try to set the same value
        System.out.println("\nBefore changing the counter
value-3");
        counter.set(102);
        System.out.println("After changing the counter
value-3");

        // Try to set a different value
        System.out.println("\nBefore changing the counter
value-4");
        counter.set(103);
```

```

        System.out.println("After changing the counter
value-4");
    }

    public static void invalidated(Observable prop) {
        System.out.println("Counter is invalid.");
    }
}

Before changing the counter value-1
Counter is invalid.
After changing the counter value-1

Before changing the counter value-2
After changing the counter value-2
Counter value = 102

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter is invalid.
After changing the counter value-4

```

## Handling Property Change Events

You can register a `ChangeListener` to receive notifications about property change events. A property change event is fired every time the value of a property changes. The `changed()` method of a `ChangeListener` receives three values: the reference of the property object, the old value, and the new value.

Let's run a similar test case for testing property change events as was done for invalidation events in the previous section. Listing 2-8 has the program to demonstrate change events that are generated for properties.

### ***Listing 2-8.*** Testing Change Events for Properties

```

// ChangeTest.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ObservableValue;

public class ChangeTest {
    public static void main(String[] args) {
        IntegerProperty counter = new
SimpleIntegerProperty(100);

        // Add a change listener to the counter property
        counter.addListener(ChangeTest::changed);

        System.out.println("\nBefore changing the counter

```

```

value-1");
        counter.set(101);
        System.out.println("After changing the counter
value-1");

        System.out.println("\nBefore changing the counter
value-2");
        counter.set(102);
        System.out.println("After changing the counter
value-2");

        // Try to set the same value
        System.out.println("\nBefore changing the counter
value-3");
        counter.set(102); // No change event is fired.
        System.out.println("After changing the counter
value-3");

        // Try to set a different value
        System.out.println("\nBefore changing the counter
value-4");
        counter.set(103);
        System.out.println("After changing the counter
value-4");
    }

    public static void changed(ObservableValue<? extends
Number> prop,
                           Number oldValue,
                           Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = "
+ newValue);
    }
}

Befor changing the counter value-1
Counter changed: Old = 100, new = 101
After changing the counter value-1

Before changing the counter value-2
Counter changed: Old = 101, new = 102
After changing the counter value-2

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter changed: Old = 102, new = 103
After changing the counter value-4

```

**In the beginning, the program creates  
an IntegerProperty named counter:**

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

There is a little trick in adding a ChangeListener.  
The `addListener()` method in the `IntegerPropertyBase` class is declared as follows:

```
void addListener(ChangeListener<? super Number> listener)
```

This means that if you are using generics, the ChangeListener for an `IntegerProperty` must be written in terms of the `Number` class or a superclass of the `Number` class. Three ways to add a ChangeListener to the counter property are shown below:

```
// Method-1: Using generics and the Number class
counter.addListener(new ChangeListener<Number>() {
    @Override
    public void changed(ObservableValue<? extends Number> prop,
                        Number oldValue,
                        Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = "
+ newValue);
    }
});

// Method-2: Using generics and the Object class
counter.addListener( new ChangeListener<Object>() {
    @Override
    public void changed(ObservableValue<? extends Object> prop,
                        Object oldValue,
                        Object newValue) {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = "
+ newValue);
    }
});

// Method-3: Not using generics. It may generate compile-time
// warnings.
counter.addListener(new ChangeListener() {
    @Override
    public void changed(ObservableValue prop,
                        Object oldValue,
                        Object newValue) {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = "
+ newValue);
    }
});
```

Listing 2-8 uses the first method, which makes use of generics; as you can see, the signature of the `changed()` method in the `ChangeTest` class matches with the `changed()` method signature in method-1. I have used a lambda expression with a method reference to add a ChangeListener as shown:

```
counter.addListener(ChangeTest::changed);
```

The output above shows that a property change event is fired when the property value is changed. Calling the `set()` method with the same value does not fire a property change event.

Unlike generating invalidation events, a property uses an eager evaluation for its value to generate change events, because it has to pass the new value to the property change listeners. The next section discusses how a property object evaluates its value, if it has both invalidation and change listeners.

## Avoiding Memory Leaks in Listeners

When you add an invalidation listener to an `Observable`, the `Observable` stores a strong reference to the listener. Like an `Observable`, an `ObservableValue` also keeps a strong reference to the registered change listeners. In a short-lived small application, you may not notice any difference. However, in a long-running big application, you may encounter memory leaks. The cause of the memory leaks is the strong reference to the listeners being stored in the observed objects, even though you do not need those listeners anymore.

**Tip** Memory leaks happens when the property object holding the strong reference to the listeners outlives the need to use the listeners. If you do not need listeners, and at the same time the property object holding their strong references becomes eligible for garbage collection, memory leaks may not occur.

The solution is to remove the listeners using `removeListener()` method when you do not need them. Implementing this solution may not always be easy. The main problem in implementing this is in deciding when to remove the listener. Sometimes multiple paths may exist, adding complexity to the solution, when listeners may need to be removed.

Listing 2-9 shows a simple use case, where a change listener is added, used, and removed. It creates an `IntegerProperty` named `counter` as a static variable. In the `main()` method, it calls the `addListener()` method that adds a change listener to the `counter` property, changes the value of `counter` to fire a change event, as shown in the output, and finally, it removes the change listener. The `main()` method changes the value of `counter` again, which does not fire any change events, because the change listener has already been removed. This is a use case where everything worked as expected.

**Listing 2-9.** Removing Listeners When They Are Not Needed

```

// CleanupListener.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;

public class CleanupListener {
    public static IntegerProperty counter = new
SimpleIntegerProperty(100);

    public static void main(String[] args) {
        // Add a change listener to the property
        ChangeListener<Number> listener
= CleanupListener::changed;
        counter.addListener(listener);

        // Change the counter value
        counter.set(200);

        // Remove the listener
        counter.removeListener(listener);

        // Will not fire change event as change listener has
        // already been removed.
        counter.set(300);
    }

    public static void changed(ObservableValue<? extends
Number> prop,
                               Number oldValue,
                               Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("old = " + oldValue + ", new = "
+ newValue);
    }
}
Counter changed: old = 100, new = 200

```

The program in Listing 2-10 shows a variation of the program in Listing 2-9. In the `addStrongListener()` method, you have added a change listener to the `counter` property but did not remove it. The second line in the output proves that even after the `addStrongListener()` method finishes executing, the `counter` property is still holding the reference to the change listener you had added. After the `addStrongListener()` method is finished, you do not have a reference to the change listener variable, because it was declared as a local variable. Therefore, you do not even have a way to remove the listener. This use case shows, though trivially, the intrinsic nature of memory leaks while using invalidation and change listeners with properties.

## ***Listing 2-10.*** Simulating Memory Leaks Because Listeners Were Not Removed

```
// StrongListener.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;

public class StrongListener {
    public static IntegerProperty counter = new
SimpleIntegerProperty(100);

    public static void main(String[] args) {
        // Add a change listener to the property
        addStrongListener();

        // Change counter value. It will fire a change
event.
        counter.set(300);
    }

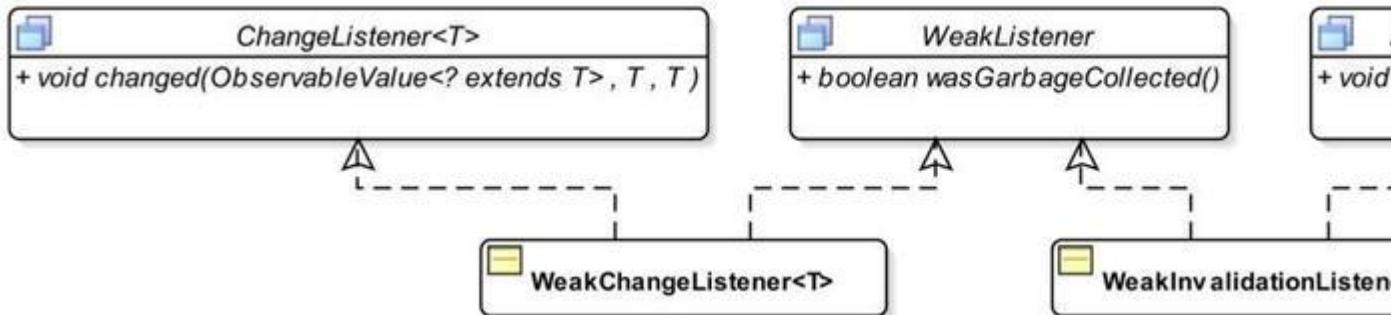
    public static void addStrongListener() {
        ChangeListener<Number> listener
= StrongListener::changed;
        counter.addListener(listener);

        // Change the counter value
        counter.set(200);
    }

    public static void changed(ObservableValue<? extends
Number> prop,
                               Number oldValue,
                               Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("old = " + oldValue + ", new = "
+ newValue);
    }
}
Counter changed: old = 100, new = 200
Counter changed: old = 200, new = 300
```

The solution is to use weak listeners, which are garbage collected automatically. A weak listener is an instance of the `WeakListener` interface. JavaFX provides two implementation classes of the `WeakListener` interface that can be used as invalidation and change listeners: `WeakInvalidationListener` and `WeakChangeListener` classes. Figure 2-3 shows a class diagram for these classes. Note that

a `WeakListener` interface has one method that tells whether the listener has been garbage collected. I will discuss change listeners in the rest of this section. However, this discussion applies to the invalidation listener as well.



**Figure 2-3.** A class diagram  
for `WeakChangeListener` and `WeakInvalidationListener`

A `WeakChangeListener` is a wrapper for a `ChangeListener`. It has only one constructor that accepts an instance of a `ChangeListener`. The following snippet of code shows how to create and use a `WeakChangeListener`:

```

ChangeListener<Number> cListener = create a change listener...
WeakChangeListener<Number> wListener = new
WeakChangeListener(cListener);

// Add a weak change listener, assuming that counter is
// a property
counter.addListener(wListener);
  
```

You might be happy to see the above snippet of code in the hope that you have found an easy solution to the big issue of memory leaks. However, this solution is not as elegant as it seems. You need to keep a strong reference of the change listener around as long as you do not want it to be garbage collected. In the above snippet of code, you will need to keep the reference `cListener` around until you know that you no longer need to listen to the change event. Isn't this similar to saying that you need to remove the listener when you do not need it? The answer is yes and no. The answer is yes, because you do need to take an action to clean up the listener. But the answer is also no, because you may design your logic to store the reference of the change listener in an object that is scoped in such a way that the change listener goes out of scope the same time you do not need it.

The program in Listing 2-11 shows, using a trivial use case, how to use a weak change listener. It is a slight variation of the previous two programs. It declares three static variables: a `counter` property, a `WeakChangeListener`, and a `ChangeListener`. The `addWeakListener()` method creates a change listener, stores its

reference to the static variable, wraps it in a weak change listener, and adds it to the `counter` property. The `counter` property is changed at the end.

The `main()` method changes the `counter` property several times. It also tries to invoke garbage collection, using `System.gc()`, and prints a message to check if the change listener has been garbage collected. As long as you keep a strong reference to the change listener in the `changeListener` static variable, the change listener is not garbage collected. After you set it to `null` and then invoke the garbage collection again, the change listener will be garbage collected. The last change in the `counter` property, inside the `main()` method, did not fire a change event as the change listener had already been removed automatically, as it was wrapped in a weak change listener.

### ***Listing 2-11.*** Using a Weak Change Listener

```
// WeakListener.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.beans.value.WeakChangeListener;

public class WeakListener {
    public static IntegerProperty counter = new
SimpleIntegerProperty(100);
    public static WeakChangeListener<Number> weakListener ;
    public static ChangeListener<Number> changeListener;

    public static void main(String[] args) {
        // Add a weak change listener to the property
        addWeakListener();

        // It will fire a change event
        counter.set(300);

        // Try garbage collection
        System.gc();

        // Check if change listener got garbage collected
        System.out.println("Garbage collected: " +
                           weakListener.wasGarbageCollected());

        // It will fire a change event
        counter.set(400);

        // You do not need a strong reference of the change
        changeListener = null;
    }
}
```

```

        // Try garbage collection
        System.gc();

        // Check if the change listener got garbage
        collected
        System.out.println("Garbage collected: " +
                           weakListener.wasGarbageCollected());

        // It will not fire a change event, if it was
        garbage collected
        counter.set(500);
    }

    public static void addWeakListener() {
        // Keep a strong reference to the change listener
        changeListener = WeakListener::changed;

        // Wrap the change listener inside a weak change
        listener
        weakListener = new
        WeakChangeListener<>(changeListener);

        // Add weak change listener
        counter.addListener(weakListener);

        // Change the value
        counter.set(200);
    }

    public static void changed(ObservableValue<? extends
Number> prop,
                           Number oldValue,
                           Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("old = " + oldValue + ", new = "
+ newValue);
    }
}
Counter changed: old = 100, new = 200
Counter changed: old = 200, new = 300
Garbage collected: false
Counter changed: old = 300, new = 400
Garbage collected: false
Counter changed: old = 400, new = 500

```

## Handling Invalidation and Change Events

You need to consider performance when you have to decide between using invalidation listeners and change listeners. Generally, invalidation listeners perform better than change listeners. The reason is twofold:

- Invalidiation listeners make it possible to compute the value lazily.

- Multiple invalidations in a row fire only one invalidation event.

However, which listener you use depends on the situation at hand. A rule of thumb is that if you read the value of the property inside the invalidation event handler, you should use a change listener instead. When you read the value of a property inside an invalidation listener, it triggers the recomputation of the value, which is automatically done before firing a change event. If you do not need to read the value of a property, use invalidation listeners.

**Listing 2-12** has a program that adds an invalidation listener and a change listener to an `IntegerProperty`. This program is a combination of Listing 2-7 and Listing 2-8. The output below it shows that when the property value changes, both events, invalidation and change, are always fired. This is because a change event makes a property valid immediately after the change, and the next change in the value fires an invalidation event, and of course, a change event too.

### ***Listing 2-12.*** Testing Invalidation and Change Events for Properties Together

```
// ChangeAndInvalidationTest.java
package com.jdojo.binding;

import javafx.beans.Observable;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ObservableValue;

public class ChangeAndInvalidationTest {
    public static void main(String[] args) {
        IntegerProperty counter = new SimpleIntegerProperty(100);

        // Add an invalidation listener to the counter
        property
        counter.addListener(ChangeAndInvalidationTest::invalidated);

        // Add a change listener to the counter property
        counter.addListener(ChangeAndInvalidationTest::changed);

        System.out.println("Before changing the counter
value-1");
        counter.set(101);
        System.out.println("After changing the counter
value-1");

        System.out.println("\nBefore changing the counter
value-1");
    }
}
```

```

value-2");
        counter.set(102);
        System.out.println("After changing the counter
value-2");

        // Try to set the same value
        System.out.println("\nBefore changing the counter
value-3");
        counter.set(102);
        System.out.println("After changing the counter
value-3");

        // Try to set a different value
        System.out.println("\nBefore changing the counter
value-4");
        counter.set(103);
        System.out.println("After changing the counter
value-4");
    }

    public static void invalidated(Observable prop) {
        System.out.println("Counter is invalid.");
    }

    public static void changed(ObservableValue<? extends
Number> prop,
                           Number oldValue,
                           Number newValue) {
        System.out.print("Counter changed: ");
        System.out.println("old = " + oldValue + ", new = "
+ newValue);
    }
}
Before changing the counter value-1
Counter is invalid.
Counter changed: old = 100, new = 101
After changing the counter value-1

Before changing the counter value-2
Counter is invalid.
Counter changed: old = 101, new = 102
After changing the counter value-2

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter is invalid.
Counter changed: old = 102, new = 103
After changing the counter value-4

```

## Using Bindings in JavaFX

In JavaFX, a binding is an expression that evaluates to a value. It consists of one or more observable values known as its *dependencies*. A binding observes its dependencies for changes and recomputes its value automatically. JavaFX uses lazy evaluation for all bindings. When a binding is initially defined or when its dependencies change, its value is marked as invalid. The value of an invalid binding is computed when it is requested next time, usually using its `get()` or `getValue()` method. All property classes in JavaFX have built-in support for binding.

Let's look at a quick example of binding in JavaFX. Consider the following expression that represents the sum of two integers `x` and `y`:

`x + y`

The expression, `x + y`, represents a binding, which has two dependencies: `x` and `y`. You can give it a name `sum` as:

`sum = x + y`

To implement the above logic in JavaFX, you create two `IntegerProperty` variables: `x` and `y`:

```
IntegerProperty x = new SimpleIntegerProperty(100);
IntegerProperty y = new SimpleIntegerProperty(200);
```

The following statement creates a binding named `sum` that represents the sum of `x` and `y`:

```
NumberBinding sum = x.add(y);
```

A binding has an `isValid()` method that returns `true` if it is valid; otherwise, it returns `false`. You can get the value of a `NumberBinding` using the methods `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` as `int`, `long`, `float`, and `double`, respectively.

The program in Listing 2-13 shows how to create and use a binding based on the above discussion. When the `sum` binding is created, it is invalid and it does not know its value. This is evident from the output. Once you request its value, using the `sum.initValue()` method, it computes its value and marks itself as valid. When you change one of its dependencies, it becomes invalid until you request its value again.

### ***Listing 2-13.*** Using a Simple Binding

```
// BindingTest.java
package com.jdojo.binding;

import javafx.beans.binding.NumberBinding;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class BindingTest {
    public static void main(String[] args) {
        IntegerProperty x = new SimpleIntegerProperty(100);
    }
}
```

```

IntegerProperty y = new SimpleIntegerProperty(200);

// Create a binding: sum = x + y
NumberBinding sum = x.add(y);

System.out.println("After creating sum");
System.out.println("sum.isValid(): " +
+ sum.isValid());

// Let us get the value of sum, so it computes its
value and
// becomes valid
int value = sum.intValue();

System.out.println("\nAfter requesting value");
System.out.println("sum.isValid(): " +
+ sum.isValid());
System.out.println("sum = " + value);

// Change the value of x
x.set(250);

System.out.println("\nAfter changing x");
System.out.println("sum.isValid(): " +
+ sum.isValid());

// Get the value of sum again
value = sum.intValue();

System.out.println("\nAfter requesting value");
System.out.println("sum.isValid(): " +
+ sum.isValid());
System.out.println("sum = " + value);
}

}

After creating sum
sum.isValid(): false

After requesting value
sum.isValid(): true
sum = 300

After changing x
sum.isValid(): false

After requesting value
sum.isValid(): true
sum = 450

```

A binding, internally, adds invalidation listeners to all of its dependencies (Listing 2-14). When any of its dependencies become invalid, it marks itself as invalid. An invalid binding does not mean that its value has changed. All it means is that it needs to recompute its value when the value is requested next time.

In JavaFX, you can also bind a property to a binding. Recall that a binding is an expression that is synchronized with its dependencies automatically. Using this definition, a bound property is a property whose value is computed based on an expression, which is automatically synchronized when the dependencies change. Suppose you have three properties, `x`, `y`, and `z`, as follows:

```
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(20);
IntegerProperty z = new SimpleIntegerProperty(60);
```

You can bind the property `z` to an expression, `x + y`, using the `bind()` method of the `Property` interface as follows:

```
z.bind(x.add(y));
```

Note that you cannot write `z.bind(x + y)` as the `+` operator does not know how to add the values of two `IntegerProperty` objects. You need to use the binding API, as you did in the above statement, to create a binding expression. I will cover the details of the binding API shortly.

Now, when `x`, `y`, or both change, the `z` property becomes invalid. The next time you request the value of `z`, it recomputes the expression `x.add(y)` to get its value.

You can use the `unbind()` method of the `Property` interface to unbind a bound property. Calling the `unbind()` method on an unbound or never bound property has no effect. You can unbind the `z` property as follows:

```
z.unbind();
```

After unbinding, a property behaves as a normal property, maintaining its value independently. Unbinding a property breaks the link between the property and its dependencies.

### ***Listing 2-14.*** Binding a Property

```
// BoundProperty.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class BoundProperty {
    public static void main(String[] args) {
        IntegerProperty x = new SimpleIntegerProperty(10);
        IntegerProperty y = new SimpleIntegerProperty(20);
        IntegerProperty z = new SimpleIntegerProperty(60);
        z.bind(x.add(y));
        System.out.println("After binding z: Bound = "
+ z.isBound() +
                ", z = " + z.get());
        // Change x and y
```

```

        x.set(15);
        y.set(19);
        System.out.println("After changing x and y: Bound
= " + z.isBound() +
                           ", z = " + z.get());
        // Unbind z
        z.unbind();

        // Will not affect the value of z as it is not bound
        to x and y anymore
        x.set(100);
        y.set(200);
        System.out.println("After unbinding z: Bound = "
+ z.isBound() +
                           ", z = " + z.get());
    }
}
After binding z: Bound = true, z = 30
After changing x and y: Bound = true, z = 34
After unbinding z: Bound = false, z = 34

```

## Unidirectional and Bidirectional Bindings

A binding has a direction, which is the direction in which changes are propagated. JavaFX supports two types of binding for properties: *unidirectional binding* and *bidirectional binding*. A unidirectional binding works only in one direction; changes in dependencies are propagated to the bound property and not vice versa. A bidirectional binding works in both directions; changes in dependencies are reflected in the property and vice versa.

The `bind()` method of the `Property` interface creates a unidirectional binding between a property and an `ObservableValue`, which could be a complex expression.

The `bindBidirectional()` method creates a bidirectional binding between a property and another property of the same type.

Suppose that `x`, `y`, and `z` are three instances of `IntegerProperty`. Consider the following bindings:

`z = x + y`

In JavaFX, the above binding can only be expressed as a unidirectional binding as follows:

`z.bind(x.add(y));`

Suppose you were able to use bidirectional binding in the above case. If you were able to change the value of `z` to 100, how would you compute the values of `x` and `y` in the reverse direction? For `z` being 100, there are an infinite number of possible combinations for `x` and `y`, for example, (99, 1), (98, 2), (101, -1), (200, -100), and so on. Propagating changes from a bound property to its dependencies is not possible with

predictable results. This is the reason that binding a property to an expression is allowed only as a unidirectional binding.

Unidirectional binding has a restriction. Once a property has a unidirectional binding, you cannot change the value of the property directly; its value must be computed automatically based on the binding. You must unbind it before changing its value directly. The following snippet of code shows this case:

```
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(20);
IntegerProperty z = new SimpleIntegerProperty(60);
z.bind(x.add(y));

z.set(7878); // Will throw a RuntimeException
```

To change the value of `z` directly, you can type the following:

```
z.unbind(); // Unbind z first
z.set(7878); // OK
```

Unidirectional binding has another restriction. A property can have only one unidirectional binding at a time. Consider the following two unidirectional bindings for a property `z`. Assume that `x`, `y`, `z`, `a`, and `b` are five instances of `IntegerProperty`:

```
z = x + y
z = a + b
```

If `x`, `y`, `a`, and `b` are four different properties, the bindings shown above for `z` are not possible. Think about `x = 1`, `y = 2`, `a = 3`, and `b = 4`. Can you define the value of `z`? Will it be `3` or `7`? This is the reason that a property can have only one unidirectional binding at a time.

Rebinding a property that already has a unidirectional binding unbinds the previous binding. For example, the following snippet of code works fine:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
IntegerProperty a = new SimpleIntegerProperty(3);
IntegerProperty b = new SimpleIntegerProperty(4);
IntegerProperty z = new SimpleIntegerProperty(0);

z.bind(x.add(y));
System.out.println("z = " + z.get());

z.bind(a.add(b)); // Will unbind the previous binding
System.out.println("z = " + z.get());
z = 3
z = 7
```

A bidirectional binding works in both directions. It has some restrictions. It can only be created between properties of the same type. That is, a bidirectional binding can only be of the type `x = y` and `y = x`, where `x` and `y` are of the same type.

Bidirectional binding removes some restrictions that are present for unidirectional binding. A property can have multiple bidirectional bindings at the same time. A bidirectional bound property can also be changed independently; the change is reflected in all properties that are bound to this property. That is, the following bindings are possible, using the bidirectional bindings:

```
x = y
x = z
```

In the above case, the values of `x`, `y`, and `z` will always be synchronized. That is, all three properties will have the same value, after the bindings are established. You can also establish bidirectional bindings between `x`, `y`, and `z` as follows:

```
x = z
z = y
```

Now a question arises. Will both of the above bidirectional bindings end up having the same values in `x`, `y`, and `z`? The answer is no. The value of the right-hand operand (see the above expressions for example) in the last bidirectional binding is the value that is contained by all participating properties. Let me elaborate this point. Suppose `x` is 1, `y` is 2, and `z` is 3, and you have the following bidirectional bindings:

```
x = y
x = z
```

The first binding, `x = y`, will set the value of `x` equal to the value of `y`. At this point, `x` and `y` will be 2. The second binding, `x = z`, will set the value of `x` to be equal to the value of `z`. That is, `x` and `z` will be 3. However, `x` already has a bidirectional binding to `y`, which will propagate the new value 3 of `x` to `y` as well. Therefore, all three properties will have the same value as that of `z`. The program in Listing 2-15 shows how to use bidirectional bindings.

### ***Listing 2-15.*** Using Bidirectional Bindings

```
// BidirectionalBinding.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class BidirectionalBinding {
    public static void main(String[] args) {
        IntegerProperty x = new SimpleIntegerProperty(1);
        IntegerProperty y = new SimpleIntegerProperty(2);
        IntegerProperty z = new SimpleIntegerProperty(3);

        System.out.println("Before binding:");
        System.out.println("x=" + x.get() + ", y=" + y.get()
+ ", z=" + z.get());
    }
}
```

```

        x.bindBidirectional(y);
        System.out.println("After binding-1:");
        System.out.println("x=" + x.get() + ", y=" + y.get()
+ ", z=" + z.get());

        x.bindBidirectional(z);
        System.out.println("After binding-2:");
        System.out.println("x=" + x.get() + ", y=" + y.get()
+ ", z=" + z.get());

        System.out.println("After changing z:");
        z.set(19);
        System.out.println("x=" + x.get() + ", y=" + y.get()
+ ", z=" + z.get());

        // Remove bindings
        x.unbindBidirectional(y);
        x.unbindBidirectional(z);
        System.out.println("After unbinding and changing
them separately:");
        x.set(100);
        y.set(200);
        z.set(300);
        System.out.println("x=" + x.get() + ", y=" + y.get()
+ ", z=" + z.get());
    }
}

Before binding:
x=1, y=2, z=3
After binding-1:
x=2, y=2, z=3
After binding-2:
x=3, y=3, z=3
After changing z:
x=19, y=19, z=19
After unbinding and changing them separately:
x=100, y=200, z=300

```

Unlike a unidirectional binding, when you create a bidirectional binding, the previous bindings are not removed because a property can have multiple bidirectional bindings. You must remove all bidirectional bindings using the `unbindBidirectional()` method, calling it once for each bidirectional binding for a property, as shown here:

```

// Create bidirectional bindings
x.bindBidirectional(y);
x.bindBidirectional(z);

// Remove bidirectional bindings
x.unbindBidirectional(y);
x.unbindBidirectional(z);

```

## Understanding the Binding API

Previous sections gave you a quick and simple introduction to bindings in JavaFX. Now it's time to dig deeper and understand the binding API in detail. The binding API is divided into two categories:

- High-level binding API
- Low-level binding API

The high-level binding API lets you define binding using the JavaFX class library. For most use cases, you can use the high-level binding API.

Sometimes the existing API is not sufficient to define a binding. In those cases, the low-level binding API is used. In low-level binding API, you derive a binding class from an existing binding class and write your own logic to define a binding.

## The High-Level Binding API

The high-level binding API consists of two parts: the Fluent API and the Bindings class. You can define bindings using only the Fluent API, only the Bindings class, or by combining the two. Let's look at both parts, first separately and then together.

### Using the Fluent API

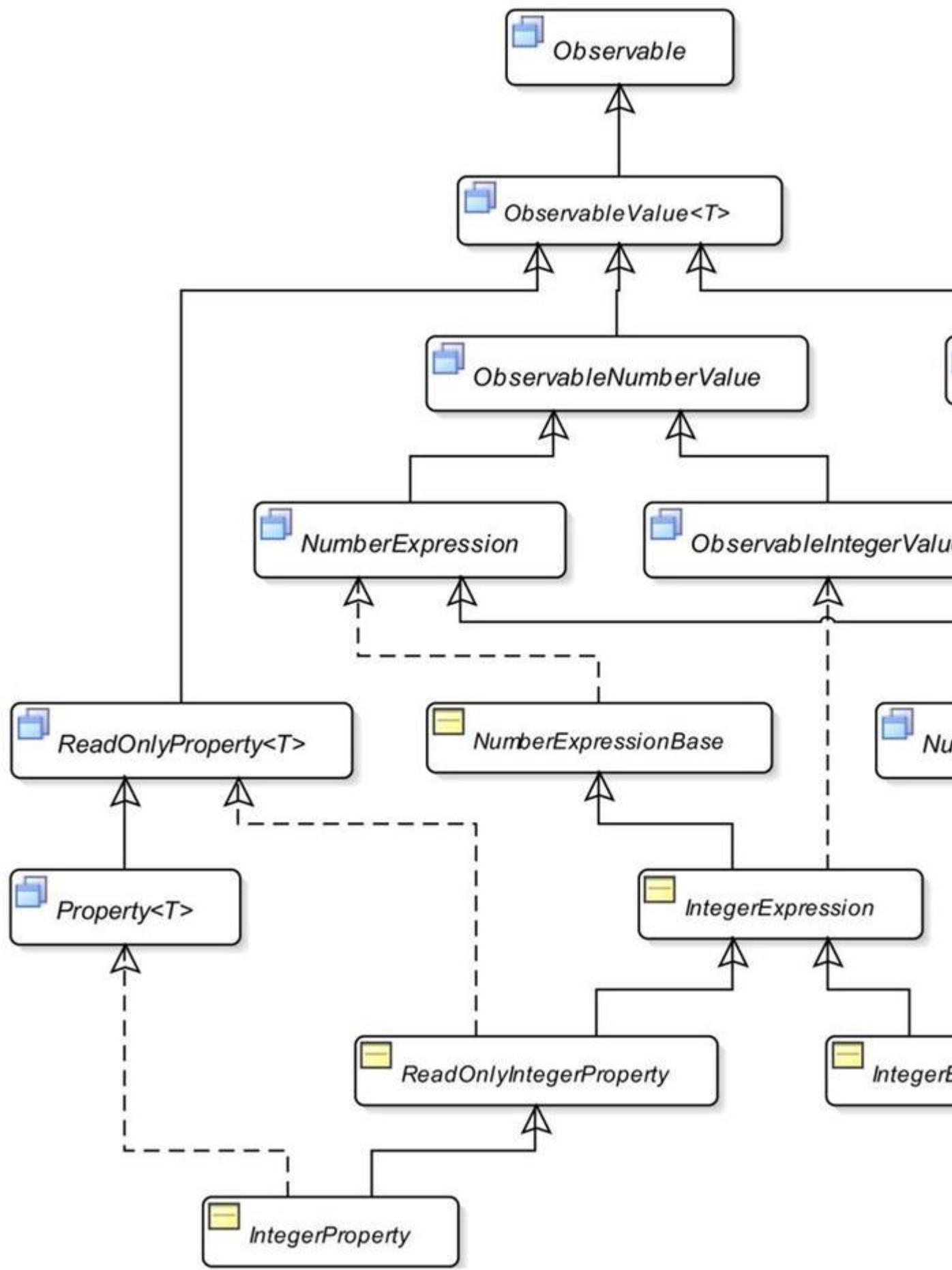
The Fluent API consists of several methods in different interfaces and classes. The API is called *Fluent* because the method names, their parameters, and return types have been designed in such a way that they allow writing the code fluently. The code written using the Fluent API is more readable as compared to code written using nonfluent APIs. Designing a fluent API takes more time. A fluent API is more developer friendly and less designer friendly. One of the features of a fluent API is *method chaining*; you can combine separate method calls into one statement. Consider the following snippet of code to add three properties x, y, and z. The code using a nonfluent API might look as follows:

```
x.add(y);  
x.add(z);
```

Using a Fluent API, the above code may look as shown below, which gives readers a better understanding of the intention of the writer:

```
x.add(y).add(z);
```

Figure 2-4 shows a class diagram for the IntegerBinding and IntegerProperty classes. The diagram has omitted some of the interfaces and classes that fall into the IntegerProperty class hierarchy. Class diagrams for long, float, and double types are similar.



**Figure 2-4.** A partial class diagram for `IntegerBinding` and `IntegerProperty`

Classes and interfaces from the `ObservableNumberValue` and `Binding` interfaces down to the `IntegerBinding` class are part of the fluent binding API for the `int` data type. At first it may seem as if there were many classes to learn. Most of the classes and interfaces exist in properties and binding APIs to avoid boxing and unboxing of primitive values. To learn the fluent binding API, you need to focus on `XXXExpression` and `XXxBinding` classes and interfaces. The `XXXExpression` classes have the methods that are used to create binding expressions.

### The *Binding* Interface

An instance of the `Binding` interface represents a value that is derived from one or more sources known as dependencies. It has the following four methods:

- `public void dispose()`
- `public ObservableList<?> getDependencies()`
- `public void invalidate()`
- `public boolean isValid()`

The `dispose()` method, whose implementation is optional, indicates to a `Binding` that it will no longer be used, so it can remove references to other objects. The binding API uses weak invalidation listeners internally, making the call to this method unnecessary.

The `getDependencies()` method, whose implementation is optional, returns an unmodifiable `ObservableList` of dependencies. It exists only for debugging purposes. This method should not be used in production code.

A call to the `invalidate()` method invalidates a `Binding`. The `isValid()` method returns `true` if a `Binding` is valid. Otherwise, it returns `false`.

### The *NumberBinding* Interface

The `NumberBinding` interface is a marker interface whose instance wraps a numeric value of `int`, `long`, `float`, or `double` type. It is implemented by `DoubleBinding`, `FloatBinding`, `IntegerBinding`, and `LongBinding` classes.

## The *ObservableNumberValue* Interface

An instance of the `ObservableNumberValue` interface wraps a numeric value of `int`, `long`, `float`, or `double` type. It provides the following four methods to get the value:

- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`

You used the `intValue()` method provided in Listing 2-13 to get the `int` value from a `NumberBinding` instance. The code you use would be:

```
IntegerProperty x = new SimpleIntegerProperty(100);
IntegerProperty y = new SimpleIntegerProperty(200);

// Create a binding: sum = x + y
NumberBinding sum = x.add(y);
int value = sum.intValue(); // Get the int value
```

## The *ObservableIntegerValue* Interface

The `ObservableIntegerValue` interface defines a `get()` method that returns the type specific `intvalue`.

## The *NumberExpression* Interface

The `NumberExpression` interface contains several convenience methods to create bindings using a fluent style. It has over 50 methods, and most of them are overloaded. These methods return a `Binding` type such as `NumberBinding`, `BooleanBinding`, and so on. Table 2-2 lists the methods in the `NumberExpression` interface. Most of the methods are overloaded. The table does not show the method arguments.

**Table 2-2.** Summary of the Methods in the `NumberExpression` Interface

Method Name	Return Type	Description
<code>add()</code>	<code>NumberBinding</code>	These methods create a new <code>NumberBinding</code> that is the difference, product, and division of the <code>NumberExpression</code> , and a
<code>subtract()</code>		
<code>multiply()</code>		
<code>divide()</code>		

Method Name	Return Type	Description
greaterThan()	BooleanBinding	value or an ObservableNumber
greaterThanOrEqualTo()	BooleanBinding	These methods create a new BooleanBinding that stores the result of the comparison of the NumberExpression and a value or an ObservableNumber.
isEqualTo()	BooleanBinding	Method names are clear enough to indicate the kind of comparisons they perform.
isNotEqualTo()	BooleanBinding	
lessThan()	BooleanBinding	
lessThanOrEqualTo()	BooleanBinding	
negate()	NumberBinding	It creates a new NumberBinding that represents the negation of the NumberExpression.
asString()	StringBinding	It creates a StringBinding that represents the value of the NumberExpression as a String object. This method also supports locale-based string formatting.

The methods in the NumberExpression interface allow for mixing types (`int`, `long`, `float`, and `double`) while defining a binding, using an arithmetic expression. When the return type of a method in this interface is `NumberBinding`, the actual returned type would be `IntegerBinding`, `LongBinding`, `FloatBinding`, or `DoubleBinding`. The binding type of an arithmetic expression is determined by the same rules as the Java programming language. The results of an expression depend on the types of the operands. The rules are as follows:

- If one of the operands is a `double`, the result is a `double`.
- If none of the operands is a `double` and one of them is a `float`, the result is a `float`.
- If none of the operands is a `double` or a `float` and one of them is a `long`, the result is a `long`.
- Otherwise, the result is an `int`.

Consider the following snippet of code:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
```

```
NumberBinding sum = x.add(y);
int value = sum.intValue();
```

The number expression `x.add(y)` involves only `int` operands (`x` and `y` are of `int` type). Therefore, according to the above rules, its result is an `int` value and it returns an `IntegerBinding` object. Because the `add()` method in the `NumberExpression` specifies the return type as `NumberBinding`, a `NumberBinding` type is used to store the result. You have to use the `intValue()` method from the `ObservableNumberValue` interface. You can rewrite the above snippet of code as follows:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);

// Casting to IntegerBinding is safe
IntegerBinding sum = (IntegerBinding)x.add(y);
int value = sum.get();
```

The `NumberExpressionBase` class is an implementation of the `NumberExpression` interface. The `IntegerExpression` class extends the `NumberExpressionBase` class. It overrides methods in its superclass to provide a type-specific return type.

The program in Listing 2-16 creates a `DoubleBinding` that computes the area of a circle. It also creates a `DoubleProperty` and binds it to the same expression to compute the area. It is your choice whether you want to work with `Binding` objects or bound property objects. The program shows you both approaches.

### ***Listing 2-16.*** Computing the Area of a Circle from Its Radius Using Fluent Binding API

```
// CircleArea.java
package com.jdojo.binding;

import javafx.beans.binding.DoubleBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class CircleArea {
    public static void main(String[] args) {
        DoubleProperty radius = new
SimpleDoubleProperty(7.0);

        // Create a binding for computing area of the circle
        DoubleBinding area
= radius.multiply(radius).multiply(Math.PI);

        System.out.println("Radius = " + radius.get() +
", Area = " + area.get());
    }
}
```

```
// Change the radius
radius.set(14.0);
System.out.println("Radius = " + radius.get() +",
Area = " + area.get());

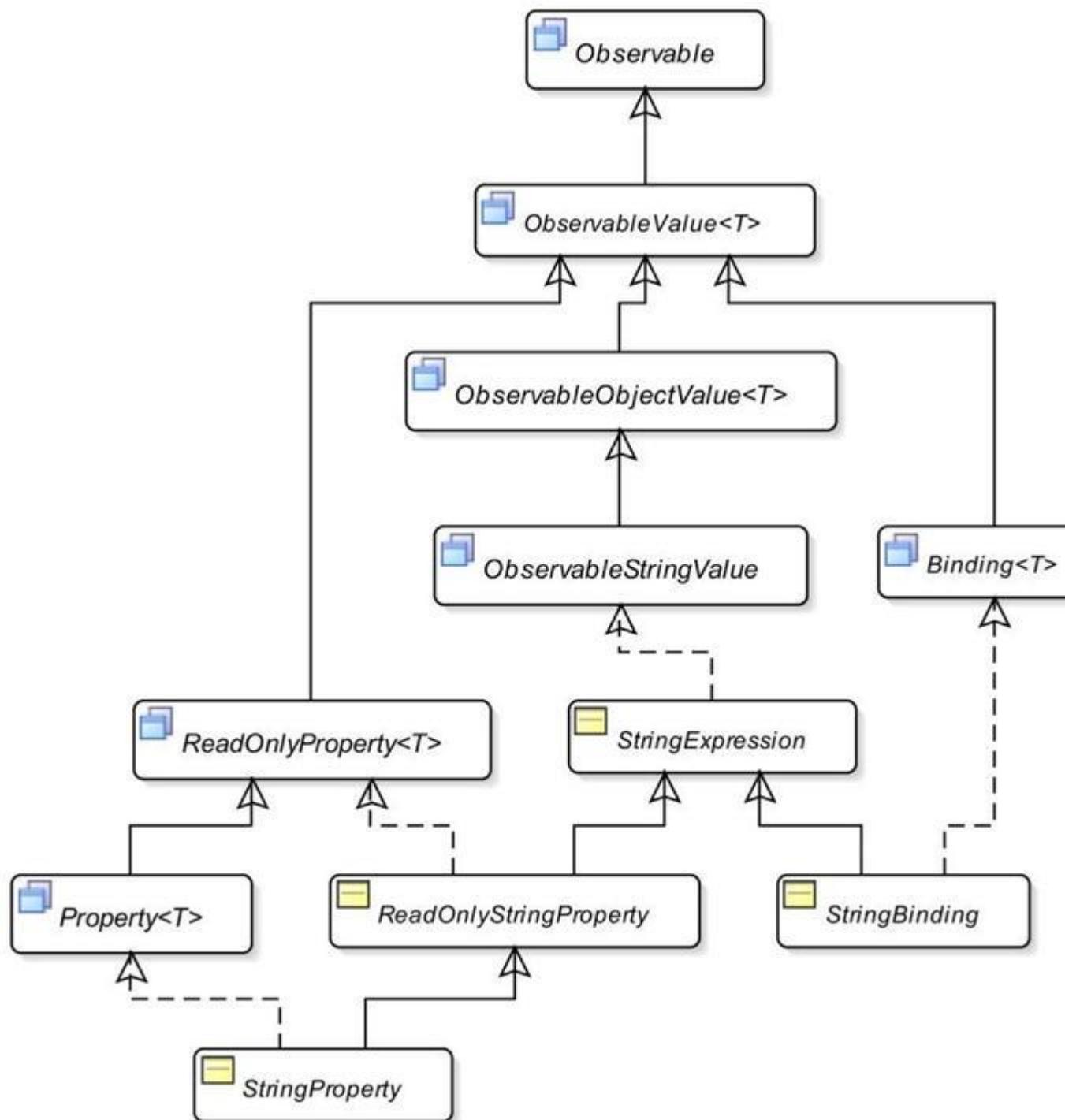
// Create a DoubleProperty and bind it to an
expression
// that computes the area of the circle
DoubleProperty area2 = new SimpleDoubleProperty();
area2.bind(radius.multiply(radius).multiply(Math.PI)
);

System.out.println("Radius = " + radius.get() +
", Area2 = " + area2.get());
}

Radius = 7.0, Area = 153.93804002589985
Radius = 14.0, Area = 615.7521601035994
Radius = 14.0, Area2 = 615.7521601035994
```

### The *StringBinding* Class

The class diagram containing classes in the binding API that supports binding of *String* type is depicted in Figure 2-5.



**Figure 2-5.** A partial class diagram for `StringBinding`

The `ObservableStringValue` interface declares a `get()` method whose return type is `String`. The methods in the `StringExpression` class let you create binding using a fluent style. Methods are provided to concatenate an object to the `StringExpression`, compare two strings, check for `null`, among others. It has two methods to get its

`value: getValue()` and `getValueSafe()`. Both return the current value. However, the latter returns an empty `String` when the current value is `null`.

The program in Listing 2-17 shows how to use `StringBinding` and `StringExpression` classes. The `concat()` method in the `StringExpression` class takes an `Object` type as an argument. If the argument is an `ObservableValue`, the `StringExpression` is updated automatically when the argument changes. Note the use of the `asString()` method on the `radius` and `area` properties. The `asString()` method on a `NumberExpression` returns a `StringBinding`.

### ***Listing 2-17.*** Using `StringBinding` and `StringExpression`

```
// StringExpressionTest.java
package com.jdojo.binding;

import java.util.Locale;
import javafx.beans.binding.StringExpression;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class StringExpressionTest {
    public static void main(String[] args) {
        DoubleProperty radius = new SimpleDoubleProperty(7.0);
        DoubleProperty area = new SimpleDoubleProperty(0);
        StringProperty initStr = new SimpleStringProperty("Radius = ");

        // Bind area to an expression that computes the area
        // of the circle
        area.bind(radius.multiply(radius).multiply(Math.PI));
    }

        // Create a string expression to describe the circle
        StringExpression desc
= initStr.concat(radius.asString())
            .concat(", Area = ")
            .concat(area.asString(Locale.US, "%.2f"));

        System.out.println(desc.getValue());

        // Change the radius
        radius.set(14.0);
        System.out.println(desc.getValue());
    }
}
```

```
    }
}
Radius = 7.0, Area = 153.94
Radius = 14.0, Area = 615.75
```

## The *ObjectExpression* and *ObjectBinding* Classes

Now it's time for *ObjectExpression* and *ObjectBinding* classes to create bindings of any type of objects. Their class diagram is very similar to that of the *StringExpression* and *StringBinding* classes.

The *ObjectExpression* class has methods to compare objects for equality and to check for null values. The program in Listing 2-18 shows how to use the *ObjectBinding* class.

### **Listing 2-18.** Using the *ObjectBinding* Class

```
// ObjectBindingTest.java
package com.jdojo.binding;

import javafx.beans.binding.BooleanBinding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;

public class ObjectBindingTest {
    public static void main(String[] args) {
        Book b1 = new Book("J1", 90, "1234567890");
        Book b2 = new Book("J2", 80, "0123456789");
        ObjectProperty<Book> book1 = new
SimpleObjectProperty<>(b1);
        ObjectProperty<Book> book2 = new
SimpleObjectProperty<>(b2);

        // Create a binding that computes if book1 and book2
are equal
        BooleanBinding isEqual = book1.isEqualTo(book2);
        System.out.println(isEqual.get());

        book2.set(b1);
        System.out.println(isEqual.get());
    }
}
false
true
```

## The *BooleanExpression* and *BooleanBinding* Classes

The *BooleanExpression* class contains methods such as *and()*, *or()*, and *not()* that let you use boolean logical operators in an expression. Its *isEqualTo()* and *isNotEqualTo()* methods let you compare a *BooleanExpression* with

another `ObservableBooleanValue`. The result of a `BooleanExpression` is true or false.

The program in Listing 2-19 shows how to use the `BooleanExpression` class. It creates a boolean expression, `x > y && y <> z`, using a fluent style. Note that the `greaterThan()` and `isNotEqualTo()` methods are defined in the `NumberExpression` interface. The program only uses the `and()` method from the `BooleanExpression` class.

### *****Listing 2-19.***** Using `BooleanExpression` and `BooleanBinding`

```
// BooleanExpressionTest.java
package com.jdojo.binding;

import javafx.beans.binding.BooleanExpression;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class BooleanExpressionTest {
    public static void main(String[] args) {
        IntegerProperty x = new SimpleIntegerProperty(1);
        IntegerProperty y = new SimpleIntegerProperty(2);
        IntegerProperty z = new SimpleIntegerProperty(3);

        // Create a boolean expression for x > y && y <> z
        BooleanExpression condition
= x.greaterThan(y).and(y.isNotEqualTo(z));

        System.out.println(condition.get());

        // Make the condition true by setting x to 3
        x.set(3);
        System.out.println(condition.get());
    }
}
false
true
```

## **Using Ternary Operation in Expressions**

The Java programming language offers a ternary operator, `(condition?value1:value2)`, to perform a ternary operation of the form *when-then-otherwise*. The JavaFX binding API has a `When` class for this purpose. The general syntax of using the `When` class is shown here:

```
new When(condition).then(value1).otherwise(value2)
```

The condition must be an `ObservableBooleanValue`. When the condition evaluates to true, it returns value1. Otherwise, it

returns `value2`. The types of `value1` and `value2` must be the same. Values may be constants or instances of `ObservableValue`.

Let's use a ternary operation that returns a `String` even or odd depending on whether the value of an `IntegerProperty` is even or odd, respectively. The Fluent API does not have a method to compute modulus. You will have to do this yourself. Perform an integer division by 2 on an integer and multiply the result by 2. If you get the same number back, the number is even. Otherwise, the number is odd. For example, using an integer division,  $(7/2)*2$ , results in 6, and not 7. Listing 2-20 provides the complete program.

### ***Listing 2-20.*** Using the `When` Class to Perform a Ternary Operation

```
// TernaryTest.java
package com.jdojo.binding;

import javafx.beans.binding.When;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.StringBinding;

public class TernaryTest {
    public static void main(String[] args) {
        IntegerProperty num = new SimpleIntegerProperty(10);
        StringBinding desc = new
When(num.divide(2).multiply(2).isEqualTo(num))
            .then("even")
            .otherwise("odd"));

        System.out.println(num.get() + " is " + desc.get());
        num.set(19);
        System.out.println(num.get() + " is " + desc.get());
    }
}
10 is even
19 is odd
```

## **Using the *Bindings* Utility Class**

The `Bindings` class is a helper class to create simple bindings. It consists of more than 150 static methods. Most of them are overloaded with several variants. I will not list or discuss all of them. Please refer to the online JavaFX API documentation to get the complete list of methods. Table 2-3 lists the methods of the `Bindings` class and their descriptions. It has excluded methods belonging to collections binding.

**Table 2-3.** Summary of Methods in the `Bindings` Class

Method Name	Description
add() subtract() multiple() divide()	They create a binding by applying an arithmetic operation, their names, on two of its arguments. At least one of the arguments must be an ObservableNumberValue. If one of the arguments is a double, its return type is DoubleBinding; otherwise, its return type is NumberBinding.
and()	It creates a BooleanBinding by applying the boolean and operation on its arguments.
bindBidirectional() unbindBidirectional()	They create and delete a bidirectional binding between two objects.
concat()	It returns a StringExpression that holds the value of the concatenation of its arguments. It takes a varargs argument.
convert()	It returns a StringExpression that wraps its argument.
createXXXBinding()	It lets you create a custom binding of XXX type, where XXX can be Boolean, Double, Float, Integer, String, and so on.
equal() notEqual() equalIgnoreCase() notEqualIgnoreCase()	They create a BooleanBinding that wraps the result of comparing two of its arguments being equal or not equal. Some variants of these methods allow passing a tolerance value. If two arguments are within the tolerance, they are considered equal. Generally, a tolerance is used to compare floating-point numbers. The ignore case variants of these methods work only on String type.
format()	It creates a StringExpression that holds the value of multiple objects formatted according to a specified format String.
greaterThan() greaterThanOrEqual() lessThan() lessThanOrEqual()	They create a BooleanBinding that wraps the result of comparing two of its arguments.

Method Name	Description
isNotNull isNull	They create a BooleanBinding that wraps the result of the argument with null.
max() min()	They create a binding that holds the maximum and minimum arguments of the method. One of the arguments must be an ObservableNumberValue.
negate()	It creates a NumberBinding that holds the negation of an ObservableNumberValue.
not()	It creates a BooleanBinding that holds the inverse of an ObservableBooleanValue.
or()	It creates a BooleanBinding that holds the result of applying conditional or operation on its two ObservableBooleanValue arguments.
selectXXX()	It creates a binding to select a nested property. The nested properties involved in the expression like a.b.c must be of the type a.b.c. The value of the binding will be c. The properties involved in the expression like a.b.c must be of the type a.b.c. The value of the binding will be c. If any part of the expression is not accessible, because they are not present in the object or they do not exist, the default value for the type, for example, null for Object type, an empty String for String type, 0 for numeric type, and false for boolean type, is the binding. (Later I will discuss an example of using the select() method.)
when()	It creates an instance of the When class taking a condition argument.

Most of our examples using the Fluent API can also be written using the Bindings class. The program in Listing 2-21 is similar to the one in Listing 2-17. It uses the Bindings class instead of the Fluent API. It uses the multiply() method to compute the area and the format() method to format the results. There may be several ways

of doing the same thing. For formatting the result, you can also use the `Bindings.concat()` method, as shown here:

```
StringExpression desc = Bindings.concat("Radius = ",
    radius.asString(Locale.US, "%.2f"),
    ", Area = ",
    area.asString(Locale.US, "%.2f"));
```

### ***Listing 2-21.*** Using the Bindings Class

```
// BindingsClassTest.java
package com.jdojo.binding;

import java.util.Locale;
import javafx.beans.binding.Bindings;
import javafx.beans.binding.StringExpression;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class BindingsClassTest {
    public static void main(String[] args) {
        DoubleProperty radius = new SimpleDoubleProperty(7.0);
        DoubleProperty area = new SimpleDoubleProperty(0.0);

        // Bind area to an expression that computes the area
        // of the circle
        area.bind(Bindings.multiply(Bindings.multiply(radius,
            radius), Math.PI));

        // Create a string expression to describe the circle
        StringExpression desc = Bindings.format(Locale.US,
            "Radius = %.2f, Area = %.2f",
            radius, area);

        System.out.println(desc.get());

        // Change the radius
        radius.set(14.0);
        System.out.println(desc.getValue());
    }
}
Radius = 7.00, Area = 153.94
Radius = 14.00, Area = 615.75
```

Let's look at an example of using the `selectXXX()` method of the `Bindings` class. It is used to create a binding for a nested property. In the nested hierarchy, all classes and properties must be public. Suppose you have an `Address` class that has a `zip` property and a `Person` class that has an `addr` property. The classes are shown in Listing 2-22 and Listing 2-23, respectively.

### ***Listing 2-22.*** An Address Class

```
// Address.java
package com.jdojo.binding;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Address {
    private StringProperty zip = new SimpleStringProperty("36106");

    public StringProperty zipProperty() {
        return zip;
    }
}
```

### ***Listing 2-23.*** An Person Class

```
// Person.java
package com.jdojo.binding;

import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;

public class Person {
    private ObjectProperty<Address> addr = new SimpleObjectProperty(new Address());

    public ObjectProperty<Address> addrProperty() {
        return addr;
    }
}
```

Suppose you create an `ObjectProperty` of the `Person` class as follows:

```
ObjectProperty<Person> p = new SimpleObjectProperty(new Person());
```

Using the `Bindings.selectString()` method, you can create a `StringBinding` for the `zip` property of the `addr` property of the `Person` object as shown here:

```
// Bind p.addr.zip
StringBinding zipBinding = Bindings.selectString(p, "addr",
"zip");
```

The above statement gets a binding for the `StringProperty` `zip`, which is a nested property of the `addr` property of the object `p`. A property in the `selectXXX()` method may have multiple levels of nesting. You can have a `selectXXX()` call like:

```
StringBinding xyzBinding = Bindings.selectString(x, "a", "b",
"c", "d");
```

**Note** JavaFX 2.2 API documentation states that `Bindings.selectString()` returns an empty `String` if any of its property arguments is inaccessible. However, the runtime returns `null`.

**Listing 2-24** shows the use of the `selectString()` method. The program prints the values of the `zip` property twice: once for its default value and once for its changed value. At the end, it tries to bind a nonexistent property `p.addr.state`. Binding to a nonexistent property is not a runtime error. When I ran the program in the latest Java Development Kit 8 release, accessing the property `p.addr.state` resulted in a runtime `NoSuchMethodException` that seems to be a bug; earlier it returned `null` without throwing the exception.

### ***Listing 2-24.*** Using the `selectXXX()` Method of the `Bindings` Class

```
// BindNestedProperty.java
package com.jdojo.binding;

import javafx.beans.binding.Bindings;
import javafx.beans.binding.StringBinding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class BindNestedProperty {
    public static class Address {
        private StringProperty zip = new SimpleStringProperty("36106");

        public StringProperty zipProperty() {
            return zip;
        }

        public String getZip() {
            return zip.get();
        }

        public void setZip(String newZip) {
            zip.set(newZip);
        }
    }

    public static class Person {
        private ObjectProperty<Address> addr =
            new SimpleObjectProperty(new Address());

        public ObjectProperty<Address> addrProperty() {
            return addr;
        }
    }
}
```

```

        public Address getAddr() {
            return addr.get();
        }

        public void setZip(Address newAddr) {
            addr.set(newAddr);
        }
    }

    public static void main(String[] args) {
        ObjectProperty<Person> p = new
SimpleObjectProperty(new Person());
        // Bind p.addr.zip
        StringBinding zipBinding = Bindings.selectString(p,
"addr", "zip");
        System.out.println(zipBinding.get());

        // Change the zip
        p.get().addrProperty().get().setZip("35217");
        System.out.println(zipBinding.get());

        // Bind p.addr.state, which does not exist
        StringBinding stateBinding
= Bindings.selectString(p, "addr", "state");
        System.out.println(stateBinding.get());
    }
}
36106
35217
null

```

## Combining the Fluent API and the *Bindings* Class

While using the high-level binding API, you can use the fluent and *Bindings* class APIs in the same binding expression. The following snippet of code shows this approach:

```

DoubleProperty radius = new SimpleDoubleProperty(7.0);
DoubleProperty area = new SimpleDoubleProperty(0);

// Combine the Fluent API and Bindings class API
area.bind(Bindings.multiply(Math.PI, radius.multiply(radius)));

```

## Using the Low-Level Binding API

The high-level binding API is not sufficient in all cases. For example, it does not provide a method to compute the square root of an *Observable* number. If the high-level binding API becomes too cumbersome to use or it does not provide what you need, you can use the low-level binding API. It gives you power and flexibility at the cost of a

few extra lines of code. The low-level API allows you to use the full potential of the Java programming language to define bindings.

Using the low-level binding API involves the following three steps:

1. Create a class that extends one of the binding classes. For example, if you want to create a `DoubleBinding`, you need to extend the `DoubleBinding` class.
2. Call the `bind()` method of the superclass to bind all dependencies. Note that all binding classes have a `bind()` method implementation. You need to call this method passing all dependencies as arguments. Its argument type is a `varargs` of `Observable` type.
3. Override the `computeValue()` method of the superclass to write the logic for your binding. It calculates the current value of the binding. Its return type is the same as the type of the binding, for example, it is `double` for a `DoubleBinding`, `String` for a `StringBinding`, and so forth.

Additionally, you can override some methods of the binding classes to provide more functionality to your binding. You can override the `dispose()` method to perform additional actions when a binding is disposed. The `getDependencies()` method may be overridden to return the list of dependencies for the binding. Overriding the `onInvalidating()` method is needed if you want to perform additional actions when the binding becomes invalid.

Consider the problem of computing the area of a circle. The following snippet of code uses the low-level API to do this:

```
final DoubleProperty radius = new SimpleDoubleProperty(7.0);
DoubleProperty area = new SimpleDoubleProperty(0);

DoubleBinding areaBinding = new DoubleBinding() {
    {
        this.bind(radius);
    }

    @Override
    protected double computeValue() {
        double r = radius.get();
        double area = Math.PI * r * r;
        return area;
    }
};

area.bind(areaBinding); // Bind the area property to the
areaBinding
```

The above snippet of code creates an anonymous class, which extends the `DoubleBinding` class. It calls the `bind()` method, passing the reference of the `radius` property. An anonymous class does not have a constructor, so you have to use an instance initializer to call the `bind()` method. The `computeValue()` method computes and returns the area of the circle. The `radius` property has been declared `final`, because it is being used inside the anonymous class.

The program in Listing 2-25 shows how to use the low-level binding API. It overrides the `computeValue()` method for the area binding. For the description binding, it overrides the `dispose()`, `getDependencies()`, and `onInvalidate()` methods as well.

***Listing 2-25.*** Using the Low-Level Binding API to Compute the Area of a Circle

```
// LowLevelBinding.java
package com.jdojo.binding;

import java.util.Formatter;
import java.util.Locale;
import javafx.beans.binding.DoubleBinding;
import javafx.beans.binding.StringBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class LowLevelBinding {
    public static void main(String[] args) {
        final DoubleProperty radius = new SimpleDoubleProperty(7.0);
        final DoubleProperty area = new SimpleDoubleProperty(0);

        DoubleBinding areaBinding = new DoubleBinding() {
            {
                this.bind(radius);
            }

            @Override
            protected double computeValue() {
                double r = radius.get();
                double area = Math.PI * r * r;
                return area;
            }
        };

        // Bind area to areaBinding
        area.bind(areaBinding);
    }
}
```

```
// Create a StringBinding
StringBinding desc = new StringBinding() {
    {
        this.bind(radius, area);
    }

    @Override
    protected String computeValue() {
        Formatter f = new Formatter();
        f.format(Locale.US, "Radius = %.2f, Area
= %.2f",
                radius.get(), area.get());
        String desc = f.toString();
        return desc;
    }

    @Override
    public ObservableList<?> getDependencies() {
        return
FXCollections.unmodifiableObservableList(
            FXCollections.observableArrayList(
radius, area));
    }

    @Override
    public void dispose() {
        System.out.println("Description binding
is disposed.");
    }

    @Override
    protected void onInvalidating() {
        System.out.println("Description is
invalid.");
    }
};

System.out.println(desc.getValue());

// Change the radius
radius.set(14.0);
System.out.println(desc.getValue());
}

}
Radius = 7.00, Area = 153.94
Description is invalid.
Radius = 14.00, Area = 615.75
```

## Using Bindings to Center a Circle

Let's look at an example of a JavaFX GUI application that uses bindings. You will create a screen with a circle, which will be centered on the screen, even after the screen is resized. The circumference of the circle

will touch the closer sides of the screen. If the width and height of the screen is the same, the circumference of the circle will touch all four sides of the screen.

Attempting to develop the screen, with a centered circle, without bindings is a tedious task. The `Circle` class in the `javafx.scene.shape` package represents a circle. It has three properties—`centerX`, `centerY`, and `radius`—of the `DoubleProperty` type. The `centerX` and `centerY` properties define the  $(x, y)$  coordinates of the center of the circle. The `radius` property defines the radius of the circle. By default, a circle is filled with black color.

You create a circle with `centerX`, `centerY`, and `radius` set to the default value of `0.0` as follows:

```
Circle c = new Circle();
```

Next, add the circle to a group and create a scene with the group as its root node as shown here:

```
Group root = new Group(c);
Scene scene = new Scene(root, 150, 150);
```

The following bindings will position and size the circle according to the size of the scene:

```
c.centerXProperty().bind(scene.widthProperty().divide(2));
c.centerYProperty().bind(scene.heightProperty().divide(2));
c.radiusProperty().bind(Bindings.min(scene.widthProperty(),
scene.heightProperty())
.divide(2));
```

The first two bindings bind the `centerX` and `centerY` of the circle to the middle of the width and height of the scene, respectively. The third binding binds the `radius` of the circle to the half (see `divide(2)`) of the minimum of the width and the height of the scene. That's it! The binding API does the magic of keeping the circle centered when the application is run.

**Listing 2-26** has the complete program. Figure 2-6 shows the screen when the program is initially run. Figure 2-7 shows the screen when the screen is stretched horizontally. Try stretching the screen vertically and you will notice that the circumference of the circle touches only the left and right sides of the screen.

### ***Listing 2-26.*** Using the Binding API to Keep a Circle Centered in a Scene

```
// CenteredCircle.java
package com.jdojo.binding;

import javafx.application.Application;
import javafx.beans.binding.Bindings;
```

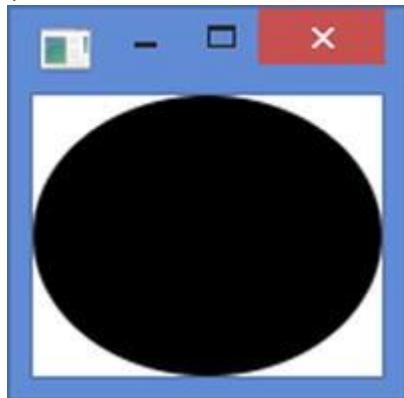
```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class CenteredCircle extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

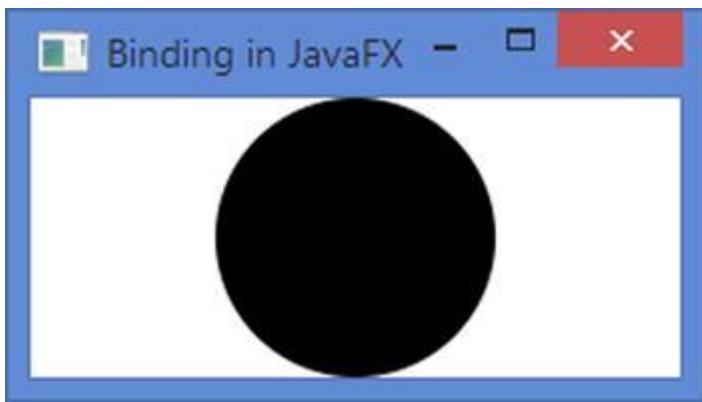
    @Override
    public void start(Stage stage) {
        Circle c = new Circle();
        Group root = new Group(c);
        Scene scene = new Scene(root, 100, 100);

        // Bind the centerX, centerY, and radius to the scene
        // width and height
        c.centerXProperty().bind(scene.widthProperty().divide(2));
        c.centerYProperty().bind(scene.heightProperty().divide(2));
        c.radiusProperty().bind(Bindings.min(scene.widthProperty(),
                                              scene.heightProperty())
                                  .divide(2));

        // Set the stage properties and make it visible
        stage.setTitle("Binding in JavaFX");
        stage.setScene(scene);
        stage.sizeToScene();
        stage.show();
    }
}
```



**Figure 2-6.** The screen when the `CenteredCircle` program is initially run



**Figure 2-7.** The screen when the screen for the *CenteredCircle* program is stretched horizontally

## Summary

A Java class may contain two types of members: fields and methods. Fields represent the state of its objects and they are declared private. Public methods, known as accessors, or getters and setters, are used to read and modify private fields. A Java class having public accessors for all or part of its private fields is known as a Java bean, and the accessors define the properties of the bean. Properties of a Java bean allow users to customize its state, behavior, or both.

JavaFX supports properties, events, and binding through properties and binding APIs. Properties support in JavaFX is a huge leap forward from the JavaBeans properties. All properties in JavaFX are observable. They can be observed for invalidation and value changes. You can have read/write or read-only properties. All read/write properties support binding. In JavaFX, a property can represent a value or a collection of values.

A property generates an invalidation event when the status of its value changes from valid to invalid for the first time. Properties in JavaFX use lazy evaluation. When an invalid property becomes invalid again, an invalidation event is not generated. An invalid property becomes valid when it is recomputed.

In JavaFX, a binding is an expression that evaluates to a value. It consists of one or more observable values known as its dependencies. A binding observes its dependencies for changes and recomputes its value automatically. JavaFX uses lazy evaluation for all bindings. When a binding is initially defined or when its dependencies change, its value is marked as invalid. The value of an invalid binding is computed when it is requested next time. All property classes in JavaFX have built-in support for binding.

A binding has a direction, which is the direction in which changes are propagated. JavaFX supports two types of binding for properties: unidirectional binding and bidirectional binding. A unidirectional binding works only in one direction; changes in dependencies are propagated to the bound property, not vice versa. A bidirectional binding works in both directions; changes in dependencies are reflected in the property and vice versa.

The binding API in JavaFX is divided into two categories: high-level binding API and low-level binding API. The high-level binding API lets you define binding using the JavaFX class library. For most use cases, you can use the high-level binding API. Sometimes, the existing API is not sufficient to define a binding. In those cases, the low-level binding API is used. In low-level binding API, you derive a binding class from an existing binding class and write your own logic to define the binding.

The next chapter will introduce you to observable collections in JavaFX.

## CHAPTER 3



### Observable Collections

In this chapter, you will learn:

- What observable collections in JavaFX are
- How to observe observable collections for invalidations and changes
- How to use observable collections as properties

#### What Are Observable Collections?

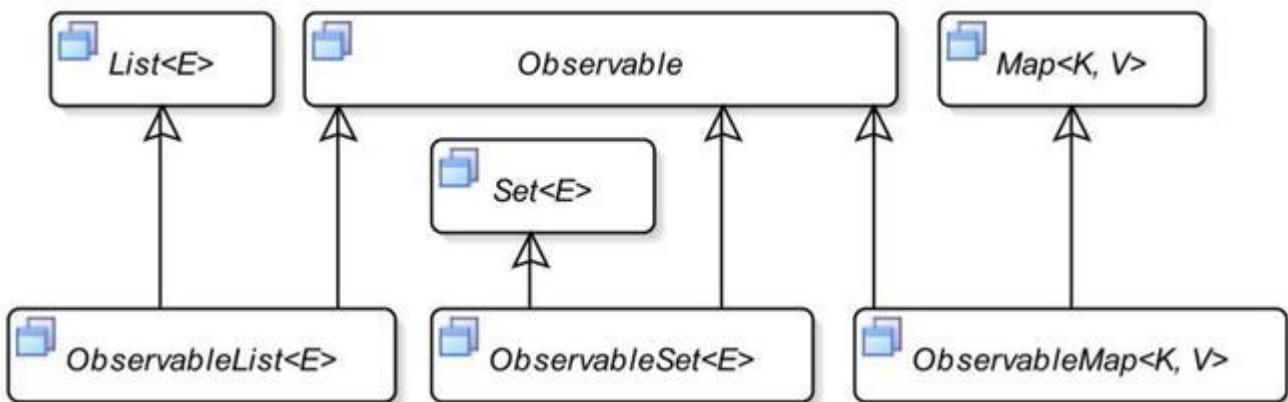
Observable collections in JavaFX are extensions to collections in Java. The *collections* framework in Java has the `List`, `Set`, and `Map` interfaces. JavaFX adds the following three types of observable collections that may be observed for changes in their contents:

- An observable list
- An observable set
- An observable map

JavaFX supports these types of collections through three new interfaces:

- `ObservableList`
- `ObservableSet`
- `ObservableMap`

These interfaces inherit from `List`, `Set`, and `Map` from the `java.util` package. In addition to inheriting from the Java collection interfaces, JavaFX collection interfaces also inherit the `Observable` interface. All JavaFX observable collection interfaces and classes are in the `javafx.collections` package. Figure 3-1 shows a partial class diagram for the `ObservableList`, `ObservableSet`, and `ObservableMap` interfaces.



**Figure 3-1.** A partial class diagram for observable collection interfaces in JavaFX

The observable collections in JavaFX have two additional features:

- They support invalidation notifications as they are inherited from the `Observable` interface.
- They support change notifications. You can register change listeners to them, which are notified when their contents change.

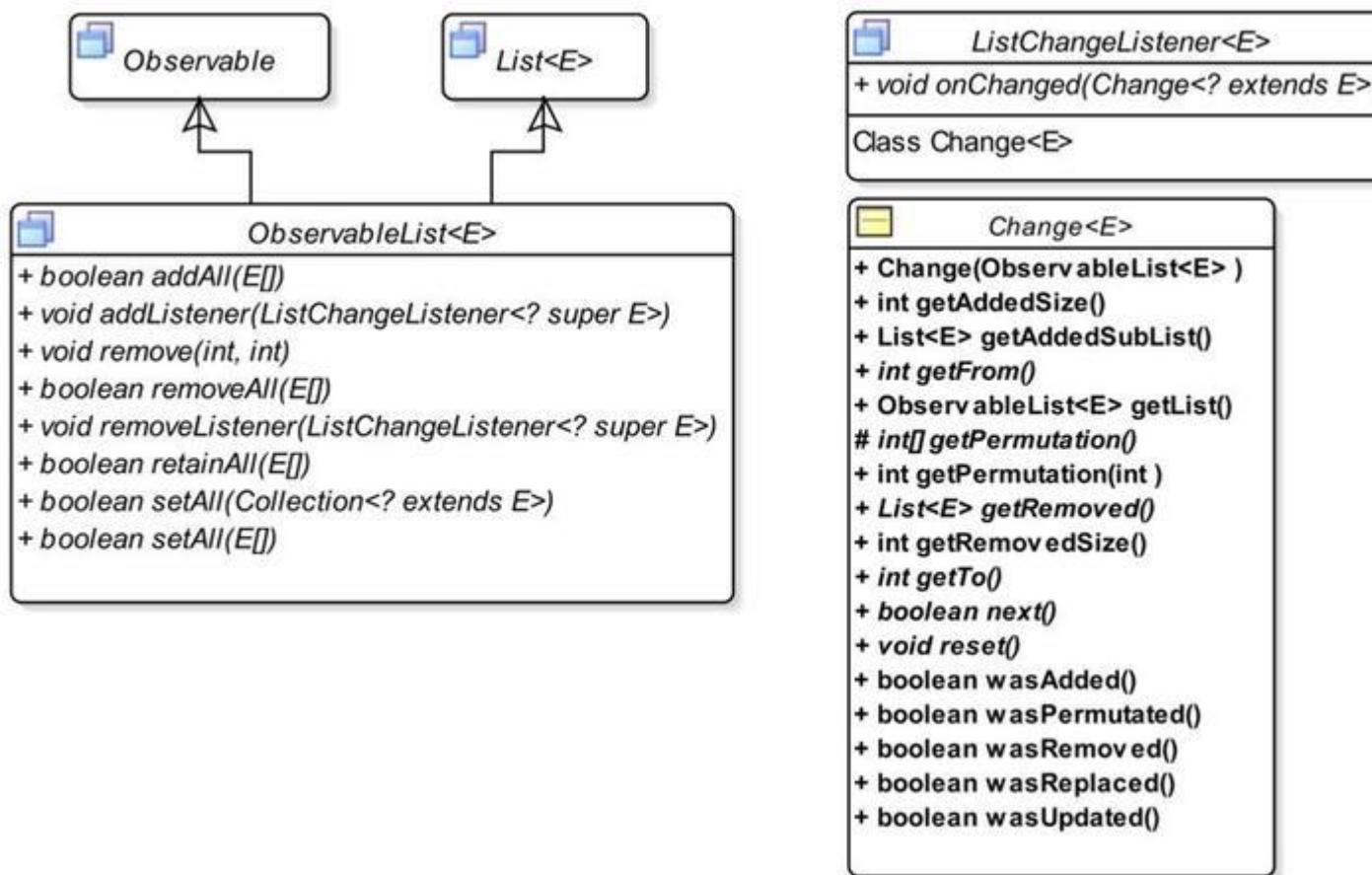
The `javafx.collections.FXCollections` class is a utility class to work with JavaFX collections. It consists of all static methods.

JavaFX does not expose the implementation classes of observable lists, sets, and maps. You need to use one of the factory methods in the `FXCollections` class to create objects of the `ObservableList`, `ObservableSet`, and `ObservableMap` interfaces.

**Tip** In simple terms, an observable collection in JavaFX is a list, set, or map that may be observed for invalidation and content changes.

## Understanding `ObservableList`

An `ObservableList` is a `java.util.List` and an `Observable` with change notification features. Figure 3-2 shows the class diagram for the `ObservableList` interface.



**Figure 3-2.** A class diagram for the `ObservableList` interface

The `addListener()` and `removeListener()` methods in the `ObservableList` interface allow you to add and remove `ListChangeListener`s, respectively. Other methods perform operations on the list, which affect multiple elements.

If you want to receive notifications when changes occur in an `ObservableList`, you need to add a `ListChangeListener` interface whose `onChanged()` method is called when a change occurs in the list. The `Change` class is a static inner class of the `ListChangeListener` interface. A `Change` object contains a report of the changes in an `ObservableList`. It is passed to the `onChanged()` method of the `ListChangeListener`. I will discuss list change listeners in detail later in this section.

You can add or remove invalidation listeners to or from an `ObservableList` using the following two methods that it inherits from the `Observable` interface:

- `void addListener(InvalidationListener listener)`

- void removeListener(InvalidationListener listener)

Note that an `ObservableList` contains all of the methods of the `List` interface as it inherits them from the `List` interface.

**Tip** JavaFX library provides two classes named `FilteredList` and `SortedList` that are in the `javafx.collections.transformation` package. A `FilteredList` is an `ObservableList` that filters its contents using a specified `Predicate`. A `SortedList` sorts its contents. I will not discuss these classes in this chapter. All discussions of observable lists apply to the objects of these classes as well.

### Creating an `ObservableList`

You need to use one of the following factory methods of the `FXCollections` class to create an `ObservableList`:

- <E> `ObservableList<E>` `emptyObservableList()`
- <E> `ObservableList<E>` `observableArrayList()`
- <E> `ObservableList<E>`  
`observableArrayList(Collection<? extends E> col)`
- <E> `ObservableList<E>`  
`observableArrayList(E... items)`
- <E> `ObservableList<E>` `observableList(List<E> list)`
- <E> `ObservableList<E>`  
`observableArrayList(Callback<E, Observable[]> extractor)`
- <E> `ObservableList<E>` `observableList(List<E> list, Callback<E, Observable[]> extractor)`

The `emptyObservableList()` method creates an empty, unmodifiable `ObservableList`. Often, this method is used when you need an `ObservableList` to pass to a method as an argument and you do not have any elements to pass to that list. You can create an empty `ObservableList` of `String` as follows:

```
ObservableList<String> emptyList
= FXCollections.emptyObservableList();
```

**The `observableArrayList()` method creates an `ObservableList` backed by an `ArrayList`.** Other variants of this

method `create` an `ObservableList` whose initial elements can be specified in a `Collection` as a list of items or as a `List`.

The last two methods in the above list create an `ObservableList` whose elements can be observed for updates. They take an extractor, which is an instance of the `Callback<E, Observable[]>` interface. An extractor is used to get the list of `Observable` values to observe for updates. I will cover the use of these two methods in the “Observing an `ObservableList` for Updates” section.

**Listing 3-1** shows how to create observable lists and how to use some of the methods of the `ObservableList` interface to manipulate the lists. At the end, it shows how to use the `concat()` method of the `FXCollections` class to concatenate elements of two observable lists.

### ***Listing 3-1.*** Creating and Manipulating Observable Lists

```
// ObservableListTest.java
package com.jdojo.collections;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class ObservableListTest {
    public static void main(String[] args) {
        // Create a list with some elements
        ObservableList<String> list
= FXCollections.observableArrayList("one", "two");
        System.out.println("After creating list: " + list);

        // Add some more elements to the list
        list.addAll("three", "four");
        System.out.println("After adding elements: "
+ list);

        // You have four elements. Remove the middle two
        // from index 1 (inclusive) to index 3 (exclusive)
        list.remove(1, 3);
        System.out.println("After removing elements: "
+ list);

        // Retain only the element "one"
        list.retainAll("one");
        System.out.println("After retaining \"one\": "
+ list);

        // Create another ObservableList
        ObservableList<String> list2 =
            FXCollections.<String>observableArrayList("1",
"2", "3");
    }
}
```

```

        // Set list2 to list
        list.setAll(list2);
        System.out.println("After setting list2 to list: "
+ list);

        // Create another list
        ObservableList<String> list3 =
            FXCollections.<String>observableArrayList("ten",
", "twenty", "thirty");

        // Concatenate elements of list2 and list3
        ObservableList<String> list4
= FXCollections.concat(list2, list3);
        System.out.println("list2 is " + list2);
        System.out.println("list3 is " + list3);
        System.out.println("After concatenating list2 and
list3:" + list4);
    }
}

After creating list: [one, two]
After adding elements: [one, two, three, four]
After removing elements: [one, four]
After retaining "one": [one]
After setting list2 to list: [1, 2, 3]
list2 is [1, 2, 3]
list3 is [ten, twenty, thirty]
After concatenating list2 and list3:[1, 2, 3, ten, twenty,
thirty]

```

## Observing an *ObservableList* for Invalidations

You can add invalidation listeners to an *ObservableList* as you do to any *Observable*. Listing 3-2 shows how to use an invalidation listener with an *ObservableList*.

**Tip** In the case of the *ObservableList*, the invalidation listeners are notified for every change in the list, irrespective of the type of a change.

### **Listing 3-2.** Testing Invalidations Notifications for an *ObservableList*

```

// ListInvalidationTest.java
package com.jdojo.collections;

import javafx.beans.Observable;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class ListInvalidationTest {
    public static void main(String[] args) {
        // Create a list with some elements
        ObservableList<String> list =

```

```

FXCollections.observableArrayList("one",
"two");

// Add an InvalidationListener to the list
list.addListener(ListInvalidationTest::invalidated);

System.out.println("Before adding three.");
list.add("three");
System.out.println("After adding three.");

System.out.println("Before adding four and five.");
list.addAll("four", "five");
System.out.println("Before adding four and five.");

System.out.println("Before replacing one with
one.");
list.set(0, "one");
System.out.println("After replacing one with one.");
}

public static void invalidated(Observable list) {
    System.out.println("List is invalid.");
}
}

Before adding three.
List is invalid.
After adding three.
Before adding four and five.
List is invalid.
Before adding four and five.
Before replacing one with one.
List is invalid.
After replacing one with one.

```

## Observing an *ObservableList* for Changes

Observing an *ObservableList* for changes is a bit tricky. There could be several kinds of changes to a list. Some of the changes could be exclusive, whereas some can occur along with other changes. Elements of a list can be permuted, updated, replaced, added, and removed. You need to be patient in learning this topic because I will cover it in bits and pieces.

You can add a change listener to an *ObservableList* using its `addListener()` method, which takes an instance of the `ListChangeListener` interface. The `changed()` method of the listeners is called every time a change occurs in the list. The following snippet of code shows how to add a change listener to an *ObservableList* of `String`. The `onChanged()` method is simple; it prints a message on the standard output when it is notified of a change:

```
// Create an observable list
ObservableList<String> list
= FXCollections.observableArrayList();

// Add a change listener to the list
list.addListener(new ListChangeListener<String>() {
    @Override
    public void onChanged(ListChangeListener.Change<? extends String> change) {
        System.out.println("List has changed.");
    }
});
```

**Listing 3-3** contains the complete program showing how to detect changes in an `ObservableList`. It uses a lambda expression with a method reference, which are features of Java 8, to add a change listener. After adding a change listener, it manipulates the list four times, and the listener is notified each time, as is evident from the output that follows.

### ***Listing 3-3.*** Detecting Changes in an `ObservableList`

```
// SimpleListChangeTest.java
package com.jdojo.collections;

import javafx.collections.FXCollections;
import javafx.collections.ListChangeListener;
import javafx.collections.ObservableList;

public class SimpleListChangeTest {
    public static void main(String[] args) {
        // Create an observable list
        ObservableList<String> list
= FXCollections.observableArrayList();

        // Add a change listener to the list
        list.addListener(SimpleListChangeTest::onChanged);

        // Manipulate the elements of the list
        list.add("one");
        list.add("two");
        FXCollections.sort(list);
        list.clear();
    }

    public static void onChanged(ListChangeListener.Change<? extends String> change) {
        System.out.println("List has changed");
    }
}
List has changed.
List has changed.
List has changed.
List has changed.
```

## Understanding the *ListChangeListener.Change* Class

Sometimes you may want to analyze changes to a list in more detail rather than just knowing that the list has changed.

The *ListChangeListener.Change* object that is passed to the *onChanged()* method contains a report to a change performed on the list. You need to use a combination of its methods to know the details of a change. Table 3-1 lists the methods in the *ListChangeListener.Change* class with their categories.

**Table 3-1.** Methods in the *ListChangeListener.Change* Class

Method	Category
<code>ObservableList&lt;E&gt; getList()</code>	General
<code>boolean next()</code> <code>void reset()</code>	Cursor movement
<code>boolean wasAdded()</code> <code>boolean wasRemoved()</code> <code>boolean wasReplaced()</code> <code>boolean wasPermutated()</code> <code>boolean wasUpdated()</code>	Change type
<code>int getFrom()</code> <code>int getTo()</code>	Affected range
<code>int getAddedSize()</code> <code>List&lt;E&gt; getAddedSubList()</code>	Addition
<code>List&lt;E&gt; getRemoved()</code> <code>int getRemovedSize()</code>	Removal
<code>int getPermutation(int oldIndex)</code>	Permutation

The *getList()* method returns the source list after changes have been made. A *ListChangeListener.Change* object may report a

change in multiple chunks. This may not be obvious at first. Consider the following snippet of code:

```
ObservableList<String> list
= FXCollections.observableArrayList();

// Add a change listener here...

list.addAll("one", "two", "three");
list.removeAll("one", "three");
```

In this code, the change listener will be notified twice: once for the `addAll()` method call and once for the `removeAll()` method call. The `ListChangeListener.Change` object reports the affected range of indexes. In the second change, you remove two elements that fall into two different ranges of indexes. Note that there is an element "two" between the two removed elements. In the second case, the `Change` object will contain a report of two changes. The first change will contain the information that, at index 0, the element "one" has been removed. Now, the list contains only two elements with the index 0 for the element "two" and index 1 for the element "three". The second change will contain the information that, at index 1, the element "three" has been removed.

A `Change` object contains a cursor that points to a specific change in the report. The `next()` and `reset()` methods are used to control the cursor. When the `onChanged()` method is called, the cursor points before the first change in the report. Calling the `next()` method the first time moves the cursor to the first change in the report. Before attempting to read the details for a change, you must point the cursor to the change by calling the `next()` method. The `next()` method returns `true` if it moves the cursor to a valid change. Otherwise, it returns `false`. The `reset()` method moves the cursor before the first change. Typically, the `next()` method is called in a while-loop, as shown in the following snippet of code:

```
ObservableList<String> list
= FXCollections.observableArrayList();
...

// Add a change listener to the list
list.addListener(new ListChangeListener<String>() {
    @Override
    public void onChanged(ListChangeListener.Change<? extends String> change) {
        while(change.next()) {
            // Process the current change here...
        }
    }
});
```

In the change type category, methods report whether a specific type of change has occurred. The `wasAdded()` method returns `true` if elements were added. The `wasRemoved()` method returns `true` if elements were removed. The `wasReplaced()` method returns `true` if elements were replaced. You can think of a replacement as a removal followed by an addition at the same index.

If `wasReplaced()` returns `true`, both `wasRemoved()` and `wasAdded()` return `true` as well. The `wasPermutated()` method returns `true` if elements of a list were permuted (i.e., reordered) but not removed, added, or updated. The `wasUpdated()` method returns `true` if elements of a list were updated.

Not all five types of changes to a list are exclusive. Some changes may occur simultaneously in the same change notification. The two types of changes, permutations and updates, are exclusive. If you are interested in working with all types of changes, your code in the `onChanged()` method should look as follows:

```
public void onChanged(ListChangeListener.Change change) {
    while (change.next()) {
        if (change.wasPermutated()) {
            // Handle permutations
        }
        else if (change.wasUpdated()) {
            // Handle updates
        }
        else if (change.wasReplaced()) {
            // Handle replacements
        }
        else {
            if (change.wasRemoved()) {
                // Handle removals
            }
            else if (change.wasAdded()) {
                // Handle additions
            }
        }
    }
}
```

In the affected range type category, the `getFrom()` and `getTo()` methods report the range of indexes affected by a change. The `getFrom()` method returns the beginning index and the `getTo()` method returns the ending index plus one. If the `wasPermutated()` method returns `true`, the range includes the elements that were permuted. If the `wasUpdated()` method returns `true`, the range includes the elements that were updated. If the `wasAdded()` method returns `true`, the range includes the elements

that were added. If the `wasRemoved()` method returns `true` and the `wasAdded()` method returns `false`, the `getFrom()` and `getTo()` methods return the same number—the index where the removed elements were placed in the list.

The `getAddedSize()` method returns the number of elements added. The `getAddedSubList()` method returns a list that contains the elements added. The `getRemovedSize()` method returns the number of elements removed. The `getRemoved()` method returns an immutable list of removed or replaced elements.

The `getPermutation(int oldIndex)` method returns the new index of an element after permutation. For example, if an element at index 2 moves to index 5 during a permutation, the `getPermutation(2)` will return 5.

This completes the discussion about the methods of the `ListChangeListener.Change` class. However, you are not done with this class yet! I still need to discuss how to use these methods in actual situations, for example, when elements of a list are updated. I will cover handling updates to elements of a list in the next section. I will finish this topic with an example that covers everything that was discussed.

## Observing an *ObservableList* for Updates

In the “Creating an *ObservableList*” section, I had listed the following two methods of the `FXCollections` class that create an `ObservableList`:

- <E> `ObservableList<E> observableArrayList(Callback<E, Observable[]> extractor)`
- <E> `ObservableList<E> observableList(List<E> list, Callback<E, Observable[]> extractor)`

If you want to be notified when elements of a list are updated, you need to create the list using one of these methods. Both methods have one thing in common: They take a `Callback<E, Observable[]>` object as an argument.

The `Callback<P, R>` interface is in the `javafx.util` package. It is defined as follows:

```
public interface Callback<P, R> {  
    R call(P param)  
}
```

The `Callback<P, R>` interface is used in situations where further action is required by APIs at a later suitable time. The first generic type parameter specifies the type of the parameter passed to the `call()` method and the second one specifies the returns type of the `call()` method.

If you notice the declaration of the type parameters in `Callback<E, Observable[]>`, the first type parameter is `E`, which is the type of the elements of the list. The second parameter is an array of `Observable`. When you add an element to the list, the `call()` method of the `Callback` object is called. The added element is passed to the `call()` method as an argument. You are supposed to return an array of `Observable` from the `call()` method. If any of the elements in the returned `Observable` array changes, listeners will be notified of an “update” change for the element of the list for which the `call()` method had returned the `Observable` array.

Let's examine why you need a `Callback` object and an `Observable` array to detect updates to elements of a list. A list stores references of its elements. Its elements can be updated using their references from anywhere in the program. A list does not know that its elements are being updated from somewhere else. It needs to know the list of `Observable` objects, where a change to any of them may be considered an update to its elements. The `call()` method of the `Callback` object fulfills this requirement. The list passes every element to the `call()` method. The `call()` method returns an array of `Observable`. The list watches for any changes to the elements of the `Observable` array. When it detects a change, it notifies its change listeners that its element associated with the `Observable` array has been updated. The reason this parameter is named *extractor* is that it extracts an array of `Observable` for an element of a list.

**Listing 3-4** shows how to create an `ObservableList` that can notify its change listeners when its elements are updated.

#### **Listing 3-4.** Observing a List for Updates of Its Elements

```
// ListUpdateTest.java
package com.jdojo.collections;

import java.util.List;
import javafx.beans.Observable;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.collections.FXCollections;
import javafx.collections.ListChangeListener;
import javafx.collections.ObservableList;
```

```

import javafx.util.Callback;

public class ListUpdateTest {
    public static void main(String[] args) {
        // Create an extractor for IntegerProperty.
        Callback<IntegerProperty, Observable[]> extractor
= (IntegerProperty p) -> {
            // Print a message to know when it
is called
            System.out.println("The extractor
is called for " + p);

            // Wrap the parameter in an
Observable[] and return it
            return new Observable[]{p};
        };

        // Create an empty observable list with a callback
to extract the
        // observable values for each element of the list
        ObservableList<IntegerProperty> list =
            FXCollections.observableArrayList(extractor);

        // Add two elements to the list
        System.out.println("Before adding two elements...");
        IntegerProperty p1 = new SimpleIntegerProperty(10);
        IntegerProperty p2 = new SimpleIntegerProperty(20);
        list.addAll(p1, p2); // Will call the call() method
of the
        // extractor - once for p1 and once
for p2.
        System.out.println("After adding two elements...");

        // Add a change listener to the list
        list.addListener(ListUpdateTest::onChanged);

        // Update p1 from 10 to 100, which will trigger
        // an update change for the list
        p1.set(100);
    }

    public static void onChanged(
        ListChangeListener.Change<? extends IntegerProperty>
change) {
        System.out.println("List is " + change.getList());

        // Work on only updates to the list
        while (change.next()) {
            if (change.wasUpdated()) {
                // Print the details of the update
                System.out.println("An update is
detected.");

                int start = change.getFrom();
                int end = change.getTo();
            }
        }
    }
}

```

```

        System.out.println("Updated range: ["
+ start + ", " + end + "]");

        List<? extends IntegerProperty>
updatedElementsList;
        updatedElementsList
= change.getList().subList(start, end);

        System.out.println("Updated elements: "
+ updatedElementsList);
    }
}
}

Before adding two elements...
The extractor is called for IntegerProperty [value: 10]
The extractor is called for IntegerProperty [value: 20]
After adding two elements...
List is [IntegerProperty [value: 100], IntegerProperty [value:
20]]
An update is detected.
Updated range: [0, 1]
Updated elements: [IntegerProperty [value: 100]]

```

The main() method of the ListUpdateTest class creates an extractor that is an object of the Callback<IntegerProperty, Observable[]> interface. The call() method takes an IntegerProperty argument and returns the same by wrapping it in an Observable array. It also prints the object that is passed to it.

The extractor is used to create an ObservableList.

Two IntegerProperty objects are added to the list. When the objects are being added, the call() method of the extractor is called with the object being added as its argument. This is evident from the output. The call() method returns the object being added. This means that the list will watch for any changes to the object (the IntegerProperty) and notify its change listeners of the same.

A change listener is added to the list. It handles only updates to the list. At the end, you change the value for the first element of the list from 10 to 100 to trigger an update change notification.

## A Complete Example of Observing an *ObservableList* for Changes

This section provides a complete example that shows how to handle the different kinds of changes to an ObservableList.

Our starting point is a Person class as shown in Listing 3-5. Here you will work with an ObservableList of Person objects.

The Person class has two properties: firstName and lastName. Both properties are of the StringProperty type. Its compareTo() method is implemented to sort Person objects in ascending order by the first name then by the last name. Its toString() method prints the first name, a space, and the last name.

***Listing 3-5.*** A Person Class with Two Properties  
Named firstName and lastName

```
// Person.java
package com.jdojo.collections;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Person implements Comparable<Person> {
    private StringProperty firstName = new
SimpleStringProperty();
    private StringProperty lastName = new
SimpleStringProperty();

    public Person() {
        this.setFirstName("Unknown");
        this.setLastName("Unknown");
    }

    public Person(String firstName, String lastName) {
        this.setFirstName(firstName);
        this.setLastName(lastName);
    }

    public final String getFirstName() {
        return firstName.get();
    }

    public final void setFirstName(String newFirstName) {
        firstName.set(newFirstName);
    }

    public StringProperty firstNameProperty() {
        return firstName;
    }

    public final String getLastName() {
        return lastName.get();
    }

    public final void setLastName(String newLastName) {
        lastName.set(newLastName);
    }

    public StringProperty lastNameProperty() {
        return lastName;
```

```

    }

    @Override
    public int compareTo(Person p) {
        // Assume that the first and last names are always
not null
        int diff
= this.getFirstName().compareTo(p.getFirstName());
        if (diff == 0) {
            diff
= this.getLastName().compareTo(p.getLastName());
        }

        return diff;
    }

    @Override
    public String toString() {
        return getFirstName() + " " + getLastName();
    }
}

```

The PersonListChangeListener class, as shown in Listing 3-6, is a change listener class. It implements the onChanged() method of the ListChangeListener interface to handle all types of change notifications for an ObservableList of Person objects.

### ***Listing 3-6.*** A Change Listener for an ObservableList of Person Objects

```

// PersonListChangeListener.java
package com.jdojo.collections;

import java.util.List;
import javafx.collections.ListChangeListener;

public class PersonListChangeListener implements
ListChangeListener<Person> {
    @Override
    public void onChanged(ListChangeListener.Change<? extends
Person> change) {
        while (change.next()) {
            if (change.wasPermutated()) {
                handlePermutated(change);
            }
            else if (change.wasUpdated()) {
                handleUpdated(change);
            }
            else if (change.wasReplaced()) {
                handleReplaced(change);
            }
            else {
                if (change.wasRemoved()) {

```

```

        handleRemoved(change);
    }
    else if (change.wasAdded()) {
        handleAdded(change);
    }
}
}

public void handlePermutated(ListChangeListener.Change<?
extends Person> change) {
    System.out.println("Change Type: Permutated");
    System.out.println("Permutated Range: "
+ getRangeText(change));
    int start = change.getFrom();
    int end = change.getTo();
    for(int oldIndex = start; oldIndex < end;
oldIndex++) {
        int newIndex
= change.getPermutation(oldIndex);
        System.out.println("index[" + oldIndex + "]"
moved to " +
                           "index[" + newIndex + "]");
    }
}

public void handleUpdated(ListChangeListener.Change<?
extends Person> change) {
    System.out.println("Change Type: Updated");
    System.out.println("Updated Range : "
+ getRangeText(change));
    System.out.println("Updated elements are: " +
change.getList().subList(change.getFrom(),
change.getTo())));
}

public void handleReplaced(ListChangeListener.Change<?
extends Person> change) {
    System.out.println("Change Type: Replaced");

    // A "replace" is the same as a "remove" followed
with an "add"
    handleRemoved(change);
    handleAdded(change);
}

public void handleRemoved(ListChangeListener.Change<?
extends Person> change) {
    System.out.println("Change Type: Removed");

    int removedSize = change.getRemovedSize();
    List<? extends Person> subList
= change.getRemoved();

    System.out.println("Removed Size: " + removedSize);
}

```

```

        System.out.println("Removed Range: "
+ getRangeText(change));
        System.out.println("Removed List: " + subList);
    }

    public void handleAdded(ListChangeListener.Change<? extends
Person> change) {
        System.out.println("Change Type: Added");

        int addedSize = change.getAddedSize();
        List<? extends Person> subList
= change.getAddedSubList();

        System.out.println("Added Size: " + addedSize);
        System.out.println("Added Range: "
+ getRangeText(change));
        System.out.println("Added List: " + subList);
    }

    public String getRangeText(ListChangeListener.Change<?
extends Person> change) {
        return "[" + change.getFrom() + ", "
+ change.getTo() + "]";
    }
}

```

The `ListChangeTest` class, as shown in Listing 3-7, is a test class. It creates an `ObservableList` with an extractor. The extractor returns an array of `firstName` and `lastName` properties of a `Person` object. That means when one of these properties is changed, a `Person` object as an element of the list is considered updated and an update notification will be sent to all change listeners. It adds a change listener to the list. Finally, it makes several kinds of changes to the list to trigger change notifications. The details of a change notification are printed on the standard output.

This completes one of the most complex discussions about writing a change listener for an `ObservableList`. Aren't you glad that JavaFX designers didn't make it more complex?

### *****Listing 3-7.***** Testing an `ObservableList` of `Person` Objects for All Types of Changes

```

// ListChangeTest.java
package com.jdojo.collections;

import javafx.beans.Observable;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.util.Callback;

public class ListChangeTest {

```

```

public static void main(String[] args) {
    Callback<Person, Observable[]> cb =
        (Person p) -> new Observable[] {
            p.firstNameProperty(),
            p.lastNameProperty()
        };

    // Create a list
    ObservableList<Person> list
    = FXCollections.observableArrayList(cb);

    // Add a change listener to the list
    list.addListener(new PersonListChangeListener());

    Person p1 = new Person("Li", "Na");
    System.out.println("Before adding " + p1 + ": "
+ list);
    list.add(p1);
    System.out.println("After adding " + p1 + ": "
+ list);

    Person p2 = new Person("Vivi", "Gin");
    Person p3 = new Person("Li", "He");
    System.out.println("\nBefore adding " + p2 + " and "
+ p3 + ": " + list);
    list.addAll(p2, p3);
    System.out.println("After adding " + p2 + " and "
+ p3 + ": " + list);

    System.out.println("\nBefore sorting the list:"
+ list);
    FXCollections.sort(list);
    System.out.println("After sorting the list:"
+ list);

    System.out.println("\nBefore updating " + p1 + ": "
+ list);
    p1.setLastName("Smith");
    System.out.println("After updating " + p1 + ": "
+ list);

    Person p = list.get(0);
    Person p4 = new Person("Simon", "Ng");
    System.out.println("\nBefore replacing " + p +
        " with " + p4 + ": " + list);
    list.set(0, p4);
    System.out.println("After replacing " + p + " with "
+ p4 + ": " + list);

    System.out.println("\nBefore setAll(): " + list);
    Person p5 = new Person("Lia", "Li");
    Person p6 = new Person("Liz", "Na");
    Person p7 = new Person("Li", "Ho");
    list.setAll(p5, p6, p7);
    System.out.println("After setAll(): " + list);

```

```
        System.out.println("\nBefore removeAll(): " + list);
        list.removeAll(p5, p7); // Leave p6 in the list
        System.out.println("After removeAll(): " + list);
    }
}

Before adding Li Na: []
Change Type: Added
Added Size: 1
Added Range: [0, 1]
Added List: [Li Na]
After adding Li Na: [Li Na]

Before adding Vivi Gin and Li He: [Li Na]
Change Type: Added
Added Size: 2
Added Range: [1, 3]
Added List: [Vivi Gin, Li He]
After adding Vivi Gin and Li He: [Li Na, Vivi Gin, Li He]

Before sorting the list:[Li Na, Vivi Gin, Li He]
Change Type: Permutated
Permutated Range: [0, 3]
index[0] moved to index[1]
index[1] moved to index[2]
index[2] moved to index[0]
After sorting the list:[Li He, Li Na, Vivi Gin]

Before updating Li Na: [Li He, Li Na, Vivi Gin]
Change Type: Updated
Updated Range : [1, 2]
Updated elements are: [Li Smith]
After updating Li Smith: [Li He, Li Smith, Vivi Gin]

Before replacing Li He with Simon Ng: [Li He, Li Smith, Vivi Gin]
Change Type: Replaced
Change Type: Removed
Removed Size: 1
Removed Range: [0, 1]
Removed List: [Li He]
Change Type: Added
Added Size: 1
Added Range: [0, 1]
Added List: [Simon Ng]
After replacing Li He with Simon Ng: [Simon Ng, Li Smith, Vivi Gin]

Before setAll(): [Simon Ng, Li Smith, Vivi Gin]
Change Type: Replaced
Change Type: Removed
Removed Size: 3
Removed Range: [0, 3]
Removed List: [Simon Ng, Li Smith, Vivi Gin]
Change Type: Added
```

```

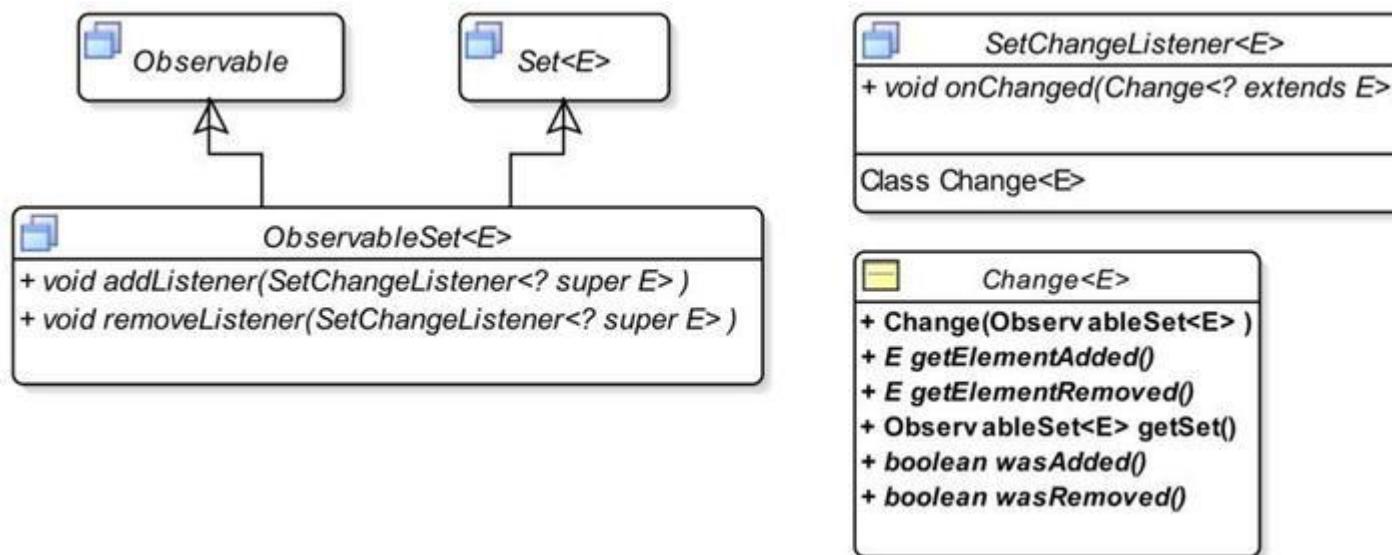
Added Size: 3
Added Range: [0, 3]
Added List: [Lia Li, Liz Na, Li Ho]
After setAll(): [Lia Li, Liz Na, Li Ho]

Before removeAll(): [Lia Li, Liz Na, Li Ho]
Change Type: Removed
Removed Size: 1
Removed Range: [0, 0]
Removed List: [Lia Li]
Change Type: Removed
Removed Size: 1
Removed Range: [1, 1]
Removed List: [Li Ho]
After removeAll(): [Liz Na]

```

## Understanding ObservableSet

If you survived learning the ObservableList and list change listeners, learning about the ObservableSet will be easy! Figure 3-3 shows the class diagram for the ObservableSet interface.



**Figure 3-3.** A class diagram for the ObservableSet interface

It inherits from the Set and Observable interfaces. It supports invalidation and change notifications and it inherits the methods for the invalidation notification support from the Observable interface. It adds the following two methods to support change notifications:

- void addListener(SetChangeListener<? super E> listener)
- void removeListener(SetChangeListener<? super E> listener)

An instance of the `SetChangeListener` interface listens for changes in an `ObservableSet`. It declares a static inner class named `Change`, which represents a report of changes in an `ObservableSet`.

**Note** A set is an unordered collection. This section shows the elements of several sets in outputs. You may get a different output showing the elements of sets in a different order than shown in those examples.

## Creating an `ObservableSet`

You need to use one of the following factory methods of the `FXCollections` class to create an `ObservableSet`:

- <E> `ObservableSet<E> observableSet(E... elements)`
- <E> `ObservableSet<E> observableSet(Set<E> set)`
- <E> `ObservableSet<E> emptyObservableSet()`

The first method lets you specify initial elements for the set. The second method lets you create an `ObservableSet` that is backed by the specified set. Mutations performed on the `ObservableSet` are reported to the listeners. Mutations performed directly on the backing set are not reported to the listeners. The third method creates an empty unmodifiable observable set. Listing 3-8 shows how to create `ObservableSets`.

### ***Listing 3-8.*** Creating ObservableSets

```
// ObservableSetTest.java
package com.jdojo.collections;

import java.util.HashSet;
import java.util.Set;
import javafx.collections.FXCollections;
import javafx.collections.ObservableSet;

public class ObservableSetTest {
    public static void main(String[] args) {
        // Create an ObservableSet with three initial
elements
        ObservableSet<String> s1
= FXCollections.observableSet("one", "two", "three");
        System.out.println("s1: " + s1);

        // Create a Set, and not an ObservableSet
        Set<String> s2 = new HashSet<String>();
        s2.add("one");
        s2.add("two");
```

```

        System.out.println("s2: " + s2);

        // Create an ObservableSet backed by the Set s2
        ObservableSet<String> s3
= FXCollections.observableSet(s2);
        s3.add("three");
        System.out.println("s3: " + s3);
    }
}

s1: [one, two, three]
s2: [one, two]
s3: [one, two, three]

```

## Observing an *ObservableSet* for Invalidations

You can add invalidation listeners to an *ObservableSet*. It fires an invalidation event when elements are added or removed. Adding an already existing element does not fire an invalidation event. Listing 3-9 shows how to use an invalidation listener with an *ObservableSet*.

### ***Listing 3-9.*** Testing Invalidations Notifications for an *ObservableSet*

```

// SetInvalidationTest.java
package com.jdojo.collections;

import javafx.beans.Observable;
import javafx.collections.FXCollections;
import javafx.collections.ObservableSet;

public class SetInvalidationTest {
    public static void main(String[] args) {
        // Create a set with some elements
        ObservableSet<String> set
= FXCollections.observableSet("one", "two");

        // Add an InvalidatorListener to the set
        set.addListener(SetInvalidationTest::invalidated);

        System.out.println("Before adding three.");
        set.add("three");
        System.out.println("After adding three.");

        System.out.println("\nBefore adding four.");
        set.add("four");
        System.out.println("After adding four.");

        System.out.println("\nBefore adding one.");
        set.add("one");
        System.out.println("After adding one.");

        System.out.println("\nBefore removing one.");
        set.remove("one");
        System.out.println("After removing one.");
    }
}

```

```
        System.out.println("\nBefore removing 123.");
        set.remove("123");
        System.out.println("After removing 123.");
    }

    public static void invalidated(Observable set) {
        System.out.println("Set is invalid.");
    }
}

Before adding three.
Set is invalid.
After adding three.

Before adding four.
Set is invalid.
After adding four.

Before adding one.
After adding one.

Before removing one.
Set is invalid.
After removing one.

Before removing 123.
After removing 123.
```

## Observing an *ObservableSet* for Changes

An *ObservableSet* can be observed for changes. You need to add a *SetChangeListener* whose *onChanged()* method is called for every addition or removal of elements. It means if you use methods like *addAll()* or *removeAll()* on an *ObservableSet*, which affects multiple elements, multiple change notifications will be fired—one for each element added or removed.

An object of the *SetChangeListener.Change* class is passed to the *onChanged()* method of the *SetChangeListener* interface. The *SetChangeListener.Change* class is a static inner class of the *SetChangeListener* interface with the following methods:

- *boolean wasAdded()*
- *boolean wasRemoved()*
- *E getElementAdded()*
- *E getElementRemoved()*
- *ObservableSet<E> getSet()*

The *wasAdded()* and *wasRemoved()* methods return *true* if an element was added and removed, respectively. Otherwise, they

`return false.`

The `getElementAdded()` and `getElementRemoved()` methods return the element that was added and removed, respectively.

The `getElementAdded()` method returns `null` if removal of an element triggers a change notification.

The `getElementRemoved()` method returns `null` if addition of an element triggers a change notification. The `getSet()` method returns the source `ObservableSet` on which the changes are performed.

The program in Listing 3-10 shows how to observe an `ObservableSet` for changes.

### ***Listing 3-10.*** Observing an `ObservableSet` for Changes

```
// SetChangeTest.java
package com.jdojo.collections;

import java.util.HashSet;
import java.util.Set;
import javafx.collections.FXCollections;
import javafx.collections.ObservableSet;
import javafx.collections.SetChangeListener;

public class SetChangeTest {
    public static void main(String[] args) {
        // Create an observable set with some elements
        ObservableSet<String> set
= FXCollections.observableSet("one", "two");

        // Add a change lisetener to the set
        set.addListener(SetChangeTest::onChanged);

        set.add("three"); // Fires an add change event

        // Will not fire a change event as "one" already
exists in the set
        set.add("one");

        // Create a Set
        Set<String> s = new HashSet<>();
        s.add("four");
        s.add("five");

        // Add all elements of s to set in one go
        set.addAll(s); // Fires two add change events

        set.remove("one"); // Fires a removal change event
        set.clear(); // Fires four removal change
events
    }

    public static void onChanged(SetChangeListener.Change<?
extends String> change) {
```

```
        if (change.wasAdded()) {
            System.out.print("Added: "
+ change.getElementAdded());
        } else if (change.wasRemoved()) {
            System.out.print("Removed: "
+ change.getElementRemoved());
        }

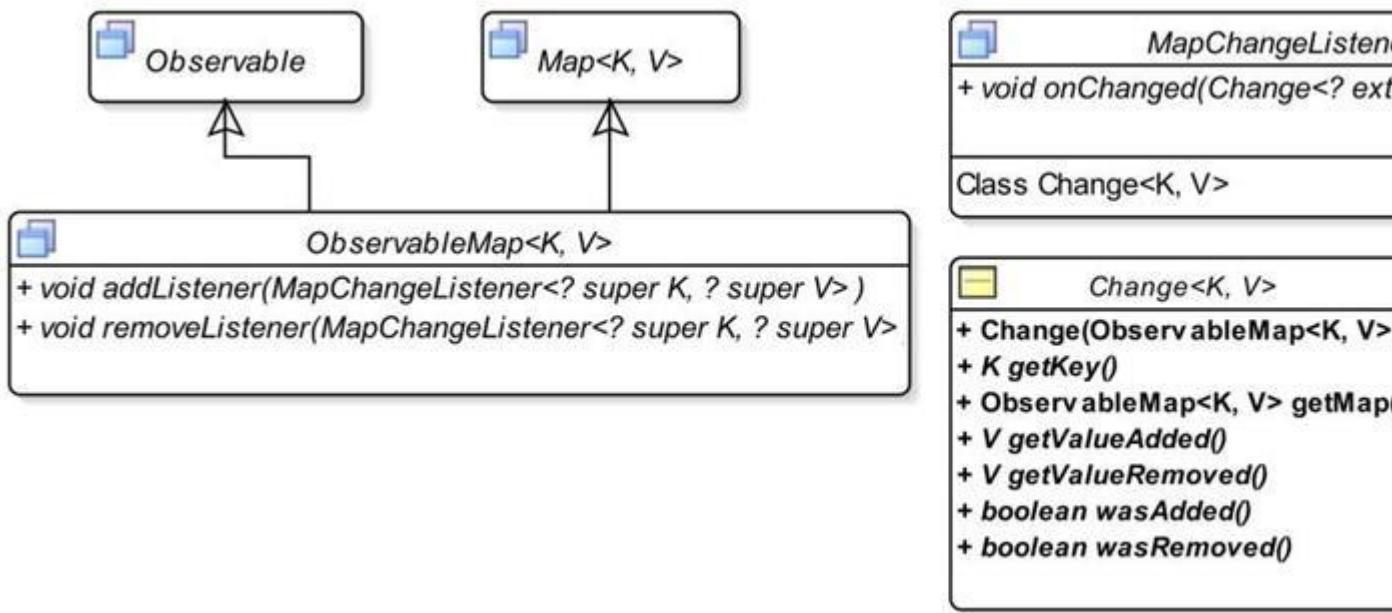
        System.out.println(", Set after the change: "
+ change.getSet());
    }
}

Added: three, Set after the change: [three, two, one]
Added: four, Set after the change: [four, one, two, three]
Added: five, Set after the change: [four, one, five, two, three]
Removed: one, Set after the change: [four, five, two, three]
Removed: four, Set after the change: [five, two, three]
Removed: five, Set after the change: [two, three]
Removed: two, Set after the change: [three]
Removed: three, Set after the change: []
```

## Understanding *ObservableMap*

Figure 3-4 shows the class diagram for the `ObservableMap` interface. It inherits from the `Map` and `Observable` interfaces. It supports invalidation and change notifications. It inherits the methods for the invalidation notification support from the `Observable` interface and it adds the following two methods to support change notifications:

- `void addListener(MapChangeListener<? super K, ? super V> listener)`
- `void removeListener(MapChangeListener<? super K, ? super V> listener)`



**Figure 3-4.** A class diagram for the `ObservableMap` interface

An instance of the `MapChangeListener` interface listens for changes in an `ObservableMap`. It declares a static inner class named `Change`, which represents a report of changes in an `ObservableMap`.

### Creating an `ObservableMap`

You need to use one of the following factory methods of the `FXCollections` class to create an `ObservableMap`:

- `<K, V> ObservableMap<K, V> observableHashMap()`
- `<K, V> ObservableMap<K, V> observableMap(Map<K, V> map)`
- `<K, V> ObservableMap<K, V> emptyObservableMap()`

The first method creates an empty observable map that is backed by a `HashMap`. The second method creates an `ObservableMap` that is backed by the specified map. Mutations performed on the `ObservableMap` are reported to the listeners. Mutations performed directly on the backing map are not reported to the listeners. The third method creates an empty unmodifiable observable map. Listing 3-11 shows how to create `ObservableMaps`.

### ***Listing 3-11.*** Creating `ObservableMaps`

```
// ObservableMapTest.java
package com.jdojo.collections;
```

```

import java.util.HashMap;
import java.util.Map;
import javafx.collections.FXCollections;
import javafx.collections.ObservableMap;

public class ObservableMapTest {
    public static void main(String[] args) {
        ObservableMap<String, Integer> map1
= FXCollections.observableHashMap();

        map1.put("one", 1);
        map1.put("two", 2);
        System.out.println("Map 1: " + map1);

        Map<String, Integer> backingMap = new HashMap<>();
        backingMap.put("ten", 10);
        backingMap.put("twenty", 20);

        ObservableMap<String, Integer> map2
= FXCollections.observableMap(backingMap);
        System.out.println("Map 2: " + map2);
    }
}
Map 1: {two=2, one=1}
Map 2: {ten=10, twenty=20}

```

## Observing an *ObservableMap* for Invalidations

You can add invalidation listeners to an *ObservableMap*. It fires an invalidation event when a new (key, value) pair is added, the value for an existing key is changed, or a (key, value) pair is removed. Invalidation events are fired once for every affected (key, value) pair. For example, if you call the `clear()` method on an observable map that has two entries, two invalidation events are fired. Listing 3-12 shows how to use an invalidation listener with an *ObservableMap*.

### ***Listing 3-12.*** Testing Invalidation Notifications for an *ObservableMap*

```

// MapInvalidationTest.java
package com.jdojo.collections;

import javafx.beans.Observable;
import javafx.collections.FXCollections;
import javafx.collections.ObservableMap;

public class MapInvalidationTest {
    public static void main(String[] args) {
        ObservableMap<String, Integer> map
= FXCollections.observableHashMap();

        // Add an InvalidationToken to the map

```

```

map.addListener(MapInvalidationTest::invalidated);

System.out.println("Before adding (\\"one\\", 1)");
map.put("one", 1);
System.out.println("After adding (\\"one\\", 1)");

System.out.println("\nBefore adding (\\"two\\", 2)");
map.put("two", 2);
System.out.println("After adding (\\"two\\", 2)");

System.out.println("\nBefore adding (\\"one\\", 1)");

// Adding the same (key, value) does not trigger an
invalidation event
map.put("one", 1);
System.out.println("After adding (\\"one\\", 1)");

System.out.println("\nBefore adding (\\"one\\",
100)");

// Adding the same key with different value triggers
invalidation event
map.put("one", 100);
System.out.println("After adding (\\"one\\", 100)");

System.out.println("\nBefore calling clear()");
map.clear();
System.out.println("After calling clear()");
}

public static void invalidated(Observable map) {
    System.out.println("Map is invalid.");
}
}

Before adding ("one", 1)
Map is invalid.
After adding ("one", 1)

Before adding ("two", 2)
Map is invalid.
After adding ("two", 2)

Before adding ("one", 1)
After adding ("one", 1)

Before adding ("one", 100)
Map is invalid.
After adding ("one", 100)

Before calling clear()
Map is invalid.
Map is invalid.
After calling clear()

```

## Observing an *ObservableMap* for Changes

An ObservableMap can be observed for changes by adding a MapChangeListener. The onChanged() method of map change listeners is called for every addition and removal of a (key, value) pair and for a change in the value of an existing key.

An object of the MapChangeListener.Change class is passed to the onChanged() method of the MapChangeListener interface. MapChangeListener.Change is a static inner class of the MapChangeListener interface with the following methods:

- boolean wasAdded()
- boolean wasRemoved()
- K getKey()
- V getValueAdded()
- V getValueRemoved()
- ObservableMap<K, V> getMap()

The wasAdded() method returns true if a (key, value) pair is added. The wasRemoved() method returns true if a (key, value) pair is removed. If the value for an existing key is replaced, both methods return true for the same change event. Replacing the value of a key is treated as a removal of the (key, oldValue) pair followed by an addition of a new (key, newValue) pair.

The getKey method returns the key associated with the change. If it is a removal, the key returned by this method does not exist in the map when the change is reported. The getValueAdded() method returns the new key value for an addition. For a removal, it returns null. The getValueRemoved() method returns the old value of the removed key. This is null if and only if the value was added to the key that was not previously in the map. The getMap() method returns the source ObservableMap on which the changes are performed.

Listing 3-13 shows how to observe an ObservableMap for changes.

### ***Listing 3-13.*** Observing an ObservableMap for Changes

```
// MapChangeTest.java
package com.jdojo.collections;

import javafx.collections.FXCollections;
import javafx.collections.MapChangeListener;
import javafx.collections.ObservableMap;

public class MapChangeTest {
    public static void main(String[] args) {
```

```

ObservableMap<String, Integer> map
= FXCollections.observableHashMap();

    // Add an MapChangeListener to the map
    map.addListener(MapChangeTest::onChanged);

    System.out.println("Before adding (\\"one\\", 1)");
    map.put("one", 1);
    System.out.println("After adding (\\"one\\", 1)");

    System.out.println("\nBefore adding (\\"two\\", 2)");
    map.put("two", 2);
    System.out.println("After adding (\\"two\\", 2)");

    System.out.println("\nBefore adding (\\"one\\", 3)");

    // Will remove ("one", 1) and add("one", 3)
    map.put("one", 3);
    System.out.println("After adding (\\"one\\", 3)");

    System.out.println("\nBefore calling clear()");
    map.clear();
    System.out.println("After calling clear()");
}

public static void onChanged(
    MapChangeListener.Change<? extends String, ? extends
Integer> change) {
    if (change.wasRemoved()) {
        System.out.println("Removed (" +
+ change.getKey() + ", " +
                change.getValueRemoved() +
+ ")");
    }

    if (change.wasAdded()) {
        System.out.println("Added (" + change.getKey() +
", " +
                change.getValueAdded() + ")");
    }
}

Before adding ("one", 1)
Added (one, 1)
After adding ("one", 1)

Before adding ("two", 2)
Added (two, 2)
After adding ("two", 2)

Before adding ("one", 3)
Removed (one, 1)
Added (one, 3)
After adding ("one", 3)

```

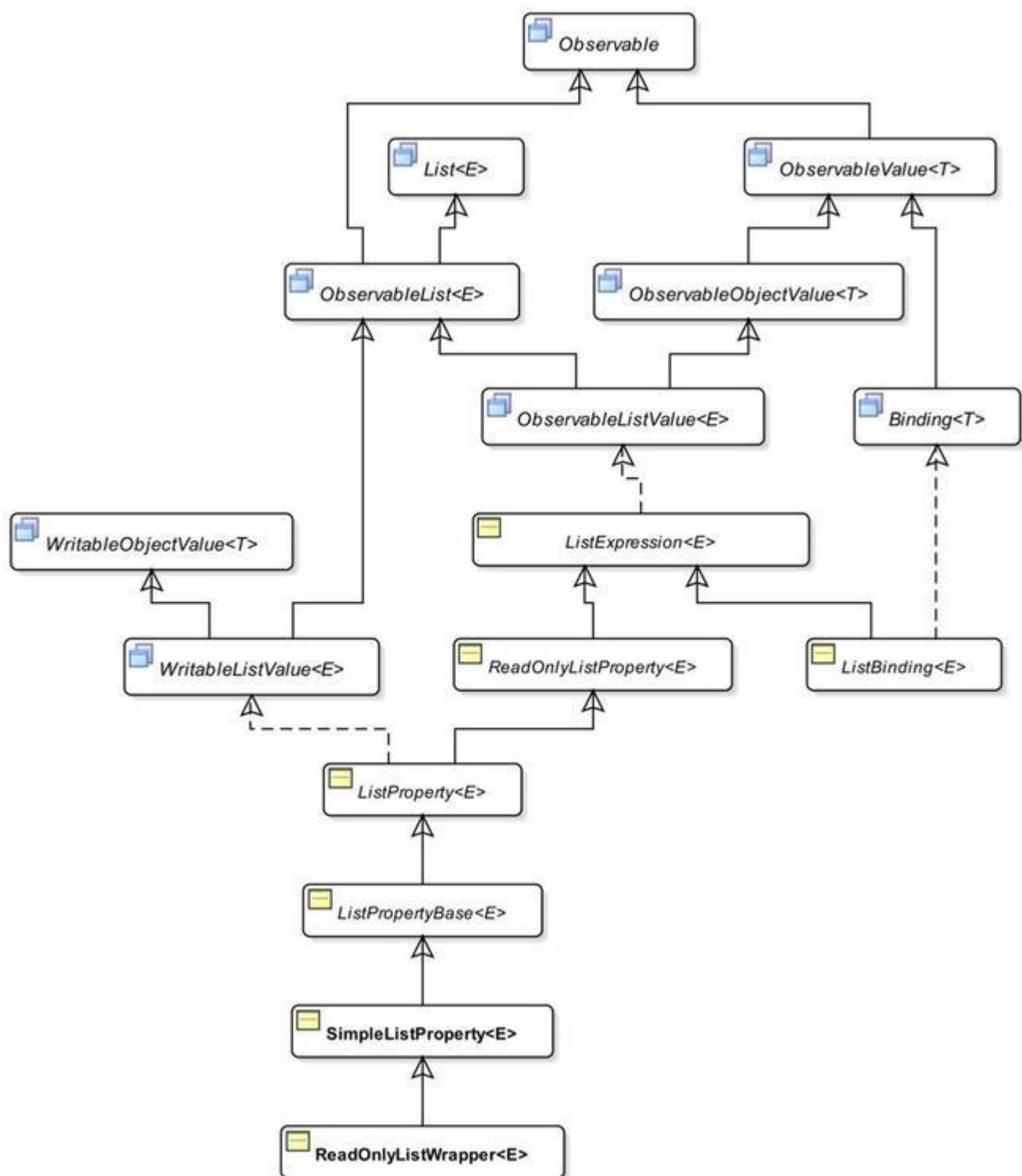
```
Before calling clear()
Removed (one, 3)
Removed (two, 2)
After calling clear()
```

## Properties and Bindings for JavaFX Collections

The `ObservableList`, `ObservableSet`, and `ObservableMap` collections can be exposed as `Property` objects. They also support bindings using high-level and low-level binding APIs. Property objects representing single values were discussed in Chapter 2. Make sure you have read that chapter before proceeding in this section.

### Understanding *ObservableList* Property and Binding

Figure 3-5 shows a partial class diagram for the `ListProperty` class. The `ListProperty` class implements the `ObservableValue` and `ObservableList` interfaces. It is an observable value in the sense that it wraps the reference of an `ObservableList`. Implementing the `ObservableList` interface makes all of its methods available to a `ListProperty` object. Calling methods of the `ObservableList` on a `ListProperty` has the same effect as if they were called on the wrapped `ObservableList`.



**Figure 3-5.** A partial class diagram for the *ListProperty* class

You can use one of the following constructors of the `SimpleListProperty` class to create an instance of the `ListProperty`:

- `SimpleListProperty()`

- SimpleListProperty(ObservableList<E> initialValue)
- SimpleListProperty(Object bean, String name)
- SimpleListProperty(Object bean, String name, ObservableList<E> initialValue)

One of the common mistakes in using the `ListProperty` class is not passing an `ObservableList` to its constructor before using it.

A `ListProperty` must have a reference to an `ObservableList` before you can perform a meaningful operation on it. If you do not use an `ObservableList` to create a `ListProperty` object, you can use its `set()` method to set the reference of an `ObservableList`. The following snippet of code generates an exception:

```
ListProperty<String> lp = new SimpleListProperty<String>();

// No ObservableList to work with. Generates an exception.
lp.add("Hello");
Exception in thread "main"
java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(AbstractList.java:148)
    at java.util.AbstractList.add(AbstractList.java:108)
    at
javafx.beans.binding.ListExpression.add(ListExpression.java:262)
```

**Tip** Operations performed on a `ListProperty` that wraps a null reference are treated as if the operations were performed on an immutable empty `ObservableList`.

The following snippet of code shows how to create and initialize a `ListProperty` before using it:

```
ObservableList<String> list1
= FXCollections.observableArrayList();
ListProperty<String> lp1 = new SimpleListProperty<String>(list1);
lp1.add("Hello");

ListProperty<String> lp2 = new SimpleListProperty<String>();
lp2.set(FXCollections.observableArrayList());
lp2.add("Hello");
```

## Observing a `ListProperty` for Changes

You can attach three types of listeners to a `ListProperty`:

- An `InvalidationListener`
- A `ChangeListener`
- A `ListChangeListener`

All three listeners are notified when the reference of the ObservableList, which is wrapped in the ListProperty, changes or the content of the ObservableList changes. When the content of the list changes, the changed() method of ChangeListeners receives the reference to the same list as the old and new value. If the wrapped reference of the ObservableList is replaced with a new one, this method receives references of the old list and the new list. To handle the list change events, please refer to the “Observing an ObservableList for Changes” section in this chapter.

The program in Listing 3-14 shows how to handle all three types of changes to a ListProperty. The list change listener handles the changes to the content of the list in a brief and generic way. Please refer to the “Observing an ObservableList for Changes” section in this chapter on how to handle the content change events for an ObservableList in detail.

### *****Listing 3-14.***** Adding Validation, Change, and List Change Listeners to a ListProperty

```
// ListPropertyTest.java
package com.jdojo.collections;

import javafx.beans.Observable;
import javafx.beans.property.ListProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ListChangeListener;
import javafx.collections.ObservableList;

public class ListPropertyTest {
    public static void main(String[] args) {
        // Create an observable list property
        ListProperty<String> lp =
            new
        SimpleListProperty<>(FXCollections.observableArrayList());

        // Add invalidation, change, and list change
        listeners
            lp.addListener(ListPropertyTest::invalidated);
            lp.addListener(ListPropertyTest::changed);
            lp.addListener(ListPropertyTest::onChanged);

        System.out.println("Before addAll()");
        lp.addAll("one", "two", "three");
        System.out.println("After addAll()");

        System.out.println("\nBefore set()");
        // Replace the wrapped list with a new one
```

```

        lp.set(FXCollections.observableArrayList("two",
"three"));
        System.out.println("After set()");

        System.out.println("\nBefore remove()");
        lp.remove("two");
        System.out.println("After remove()");
    }

    // An invalidation listener
    public static void invalidated(Observable list) {
        System.out.println("List property is invalid.");
    }

    // A change listener
    public static void changed(ObservableValue<? extends
ObservableList<String>> observable,
                           ObservableList<String> oldList,
                           ObservableList<String> newList) {
        System.out.print("List Property has changed.");
        System.out.print(" Old List: " + oldList);
        System.out.println(", New List: " + newList);
    }

    // A list change listener
    public static void onChanged(ListChangeListener.Change<?
extends String> change) {
        while (change.next()) {
            String action = change.wasPermutated()
? "Permutated"
                : change.wasUpdated() ? "Updated"
                : change.wasRemoved() &&
change.wasAdded() ? "Replaced"
                : change.wasRemoved() ? "Removed"
: "Added";

            System.out.print("Action taken on the list: "
+ action);
            System.out.print(". Removed: "
+ change.getRemoved());
            System.out.println(", Added: "
+ change.getAddedSubList());
        }
    }
}

Before addAll()
List property is invalid.
List Property has changed. Old List: [one, two, three], New List:
[one, two, three]
Action taken on the list: Added. Removed: [], Added: [one, two,
three]
After addAll()

Before set()

```

```

List property is invalid.
List Property has changed. Old List: [one, two, three], New List:
[two, three]
Action taken on the list: Replaced. Removed: [one, two, three],
Added: [two, three]
After set()

Before remove()
List property is invalid.
List Property has changed. Old List: [three], New List: [three]
Action taken on the list: Removed. Removed: [two], Added: []
After remove()

```

## Binding the `size` and `empty` Properties of a `ListProperty`

A `ListProperty` exposes two properties, `size` and `empty`, which are of

`type ReadOnlyIntegerProperty` and `ReadOnlyBooleanProperty`, respectively. You can access them using the `sizeProperty()` and `emptyProperty()` methods.

The `size` and `empty` properties are useful for binding in GUI applications. For example, the model in a GUI application may be backed by a `ListProperty`, and you can bind these properties to the text property of a label on the screen. When the data changes in the model, the label will be updated automatically through binding.

The `size` and `empty` properties are declared in the `ListExpression` class.

The program in Listing 3-15 shows how to use the `size` and `empty` properties. It uses the `asString()` method of the `ListExpression` class to convert the content of the wrapped `ObservableList` to a `String`.

### ***Listing 3-15.*** Using the `size` and `empty` Properties of a `ListProperty` Object

```

// ListBindingTest.java
package com.jdojo.collections;

import javafx.beans.property.ListProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;

public class ListBindingTest {
    public static void main(String[] args) {
        ListProperty<String> lp =
            new

```

```

SimpleListProperty<>(FXCollections.observableArrayList());
    // Bind the size and empty properties of the
ListProperty
    // to create a description of the list
    StringProperty initStr = new
SimpleStringProperty("Size: " );
    StringProperty desc = new SimpleStringProperty();
    desc.bind(initStr.concat(lp.sizeProperty())
        .concat(", Empty: ")
        .concat(lp.emptyProperty())
        .concat(", List: ")
        .concat(lp.asString())));
    System.out.println("Before addAll(): "
+ desc.get());
    lp.addAll("John", "Jacobs");
    System.out.println("After addAll(): " + desc.get());
}
}
Before addAll(): Size: 0, Empty: true, List: []
After addAll(): Size: 2, Empty: false, List: [John, Jacobs]

```

Methods to support high-level binding for a list property are in the `ListExpression` and `Bindings` classes. Low-level binding can be created by subclassing the `ListBinding` class.

A `ListProperty` supports two types of bindings:

- Binding the reference of the `ObservableList` that it wraps
- Binding the content of the `ObservableList` that it wraps

The `bind()` and `bindBidirectional()` methods are used to create the first kind of binding. The program in Listing 3-16 shows how to use these methods. As shown in the output below, notice that both list properties have the reference of the same `ObservableList` after binding.

### ***Listing 3-16.*** Binding the References of List Properties

```

// BindingListReference.java
package com.jdojo.collections;

import javafx.beans.property.ListProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.collections.FXCollections;

public class BindingListReference {
    public static void main(String[] args) {
        ListProperty<String> lp1 =
            new

```

```

SimpleListProperty<>(FXCollections.observableArrayList());
    ListProperty<String> lp2 =
        new
SimpleListProperty<>(FXCollections.observableArrayList());
    lp1.bind(lp2);

    print("Before addAll():", lp1, lp2);
    lp1.addAll("One", "Two");
    print("After addAll():", lp1, lp2);

    // Change the reference of the ObservableList in lp2
    lp2.set(FXCollections.observableArrayList("1",
"2"));
    print("After lp2.set():", lp1, lp2);

    // Cannot do the following as lp1 is a bound
property
    // lp1.set(FXCollections.observableArrayList("1",
"2"));
    // Unbind lp1
    lp1.unbind();
    print("After unbind():", lp1, lp2);

    // Bind lp1 and lp2 bidirectionally
    lp1.bindBidirectional(lp2);
    print("After bindBidirectional():", lp1, lp2);

    lp1.set(FXCollections.observableArrayList("X",
"Y"));
    print("After lp1.set():", lp1, lp2);
}

public static void print(String msg, ListProperty<String>
lp1, ListProperty<String> lp2) {
    System.out.println(msg);
    System.out.println("lp1: " + lp1.get() + ", lp2: "
+ lp2.get() +
                    ", lp1.get() == lp2.get(): "
+ (lp1.get() == lp2.get()));
    System.out.println("-----");
}
}

Before addAll():
lp1: [], lp2: [], lp1.get() == lp2.get(): true
-----
After addAll():
lp1: [One, Two], lp2: [One, Two], lp1.get() == lp2.get(): true
-----
After lp2.set():
lp1: [1, 2], lp2: [1, 2], lp1.get() == lp2.get(): true
-----
After unbind():
lp1: [1, 2], lp2: [1, 2], lp1.get() == lp2.get(): true
-----
```

```

After bindBidirectional():
lp1: [1, 2], lp2: [1, 2], lp1.get() == lp2.get(): true
-----
After lp1.set():
lp1: [X, Y], lp2: [X, Y], lp1.get() == lp2.get(): true
-----
```

The `bindContent()` and `bindContentBidirectional()` methods let you bind the content of the `ObservableList` that is wrapped in a `ListProperty` to the content of another `ObservableList` in one direction and both directions, respectively. Make sure to use the corresponding methods, `unbindContent()` and `unbindContentBidirectional()`, to unbind contents of two observable lists.

**Tip** You can also use methods of the `Bindings` class to create bindings for references and contents of observable lists.

It is allowed, but not advisable, to change the content of a `ListProperty` whose content has been bound to another `ObservableList`. In such cases, the bound `ListProperty` will not be synchronized with its target list. Listing 3-17 shows examples of both types of content binding.

### ***Listing 3-17.*** Binding Contents of List Properties

```

// BindingListContent.java
package com.jdojo.collections;

import javafx.beans.property.ListProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.collections.FXCollections;

public class BindingListContent {

    public static void main(String[] args) {
        ListProperty<String> lp1 =
            new
        SimpleListProperty<>(FXCollections.observableArrayList());
        ListProperty<String> lp2 =
            new
        SimpleListProperty<>(FXCollections.observableArrayList());

        // Bind the content of lp1 to the content of lp2
        lp1.bindContent(lp2);

        /* At this point, you can change the content of lp1.
        However,
            * that will defeat the purpose of content binding,
        because the
            * content of lp1 is no longer in sync with the
```

```

content of lp2.
    * Do not do this:
    * lp1.addAll("X", "Y");
    */
    print("Before lp2.addAll():", lp1, lp2);
    lp2.addAll("1", "2");
    print("After lp2.addAll():", lp1, lp2);

    lp1.unbindContent(lp2);
    print("After lp1.unbindContent(lp2):", lp1, lp2);

    // Bind lp1 and lp2 contents bidirectionally
    lp1.bindContentBidirectional(lp2);

    print("Before lp1.addAll():", lp1, lp2);
    lp1.addAll("3", "4");
    print("After lp1.addAll():", lp1, lp2);

    print("Before lp2.addAll():", lp1, lp2);
    lp2.addAll("5", "6");
    print("After lp2.addAll():", lp1, lp2);
}

public static void print(String msg, ListProperty<String>
lp1, ListProperty<String> lp2) {
    System.out.println(msg + " lp1: " + lp1.get() + ", "
lp2: " + lp2.get());
}
}
Before lp2.addAll(): lp1: [], lp2: []
After lp2.addAll(): lp1: [1, 2], lp2: [1, 2]
After lp1.unbindContent(lp2): lp1: [1, 2], lp2: [1, 2]
Before lp1.addAll(): lp1: [1, 2], lp2: [1, 2]
After lp1.addAll(): lp1: [1, 2, 3, 4], lp2: [1, 2, 3, 4]
Before lp2.addAll(): lp1: [1, 2, 3, 4], lp2: [1, 2, 3, 4]
After lp2.addAll(): lp1: [1, 2, 3, 4, 5, 6], lp2: [1, 2, 3, 4, 5,
6]

```

## Binding to Elements of a List

ListProperty provides so many useful features that I can keep discussing this topic for at least 50 more pages! I will wrap this topic up with one more example.

It is possible to bind to a specific element of the ObservableList wrapped in a ListProperty using one of the following methods of the ListExpression class:

- ObjectBinding<E> valueAt(int index)
- ObjectBinding<E>  
valueAt(ObservableIntegerValue index)

The first version of the method creates an `ObjectBinding` to an element in the list at a specific index. The second version of the method takes an index as an argument, which is an `ObservableIntegerValue` that can change over time. When the bound index in the `valueAt()` method is outside the list range, the `ObjectBinding` contains `null`.

Let's use the second version of the method to create a binding that will bind to the last element of a list. Here you can make use of the `size` property of the `ListProperty` in creating the binding expression. The program in Listing 3-18 shows how to use the `valueAt()` method. Note that this program throws an `ArrayIndexOutOfBoundsException` when run using Java Development Kit 8 Build 25. It did not throw an exception before and it does not throw an exception in Java Development Kit 9's early access build.

### ***Listing 3-18.*** Binding to the Elements of a List

```
// BindingToListElements.java
package com.jdojo.collections;

import javafx.beans.binding.ObjectBinding;
import javafx.beans.property.ListProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.collections.FXCollections;

public class BindingToListElements {
    public static void main(String[] args) {
        ListProperty<String> lp =
            new
SimpleListProperty<>(FXCollections.observableArrayList());

        // Create a binding to the last element of the list
        ObjectBinding<String> last
= lp.valueAt(lp.sizeProperty().subtract(1));
        System.out.println("List:" + lp.get() + ", Last
Value: " + last.get());

        lp.add("John");
        System.out.println("List:" + lp.get() + ", Last
Value: " + last.get());

        lp.addAll("Donna", "Geshan");
        System.out.println("List:" + lp.get() + ", Last
Value: " + last.get());

        lp.remove("Geshan");
        System.out.println("List:" + lp.get() + ", Last
Value: " + last.get());
    }
}
```

```
        lp.clear();
        System.out.println("List:" + lp.get() + ", Last
Value: " + last.get());
    }
}
List:[], Last Value: null
List:[John], Last Value: John
List:[John, Donna, Geshan], Last Value: Geshan
List:[John, Donna], Last Value: Donna
List:[], Last Value: null
```

## Understanding *ObservableSet* Property and Binding

A  `SetProperty` object wraps an  `ObservableSet`. Working with a  `SetProperty` is very similar to working with a  `ListProperty`. I am not going to repeat what has been discussed in the previous sections about properties and bindings of an  `ObservableList`. The same discussions apply to properties and bindings of  `ObservableSet`. The following are the salient points to remember while working with a  `SetProperty`:

- The class diagram for the  `SetProperty` class is similar to the one shown in Figure 3-5 for the  `ListProperty` class. You need to replace the word “List” with the word “Set” in all names.
- The  `SetExpression` and  `Bindings` classes contain methods to support high-level bindings for set properties. You need to subclass the  `SetBinding` class to create low-level bindings.
- Like the  `ListProperty`, the  `SetProperty` exposes the  `size` and  `empty` properties.
- Like the  `ListProperty`, the  `SetProperty` supports bindings of the reference and the content of the  `ObservableSet` that it wraps.
- Like the  `ListProperty`, the  `SetProperty` supports three types of notifications: invalidation notifications, change notifications, and set change notifications.
- Unlike a list, a set is an unordered collection of items. Its elements do not have indexes. It does not support binding to its specific elements. Therefore, the  `SetExpression` class does not contain a method like  `valueAt()` as the  `ListExpression` class does.

You can use one of the following constructors of the `SimpleSetProperty` class to create an instance of the `SetProperty`:

- `SimpleSetProperty()`
- `SimpleSetProperty(ObservableSet<E> initialValue)`
- `SimpleSetProperty(Object bean, String name)`
- `SimpleSetProperty(Object bean, String name, ObservableSet<E> initialValue)`

The following snippet of code creates an instance of the `SetProperty` and adds two elements to the `ObservableSet` that the property wraps. In the end, it gets the reference of the `ObservableSet` from the property object using the `get()` method:

```
// Create a SetProperty object
SetProperty<String> sp = new
SimpleSetProperty<String>(FXCollections.observableSet());

// Add two elements to the wrapped ObservableSet
sp.add("one");
sp.add("two");

// Get the wrapped set from the sp property
ObservableSet<String> set = sp.get();
```

The program in Listing 3-19 demonstrates how to use binding with `SetProperty` objects.

### ***Listing 3-19.*** Using Properties and Bindings for Observable Sets

```
// SetBindingTest.java
package com.jdojo.collections;

import javafx.beans.property SetProperty;
import javafx.beans.property.SimpleSetProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;

public class SetBindingTest {
    public static void main(String[] args) {
        SetProperty<String> sp1 =
            new
        SimpleSetProperty<>(FXCollections.observableSet());

        // Bind the size and empty properties of the
        SetProperty
            // to create a description of the set
            StringProperty initStr = new
        SimpleStringProperty("Size: " );
```

```

        StringProperty desc = new SimpleStringProperty();
        desc.bind(initStr.concat(sp1.sizeProperty())
                  .concat(", Empty: ")
                  .concat(sp1.emptyProperty())
                  .concat(", Set: " )
                  .concat(sp1.asString()))
        );
        System.out.println("Before sp1.add(): "
+ desc.get());
        sp1.add("John");
        sp1.add("Jacobs");
        System.out.println("After sp1.add(): "
+ desc.get());

        SetProperty<String> sp2 =
            new
SimpleSetProperty<>(FXCollections.observableSet());
        // Bind the content of sp1 to the content of sp2
        sp1.bindContent(sp2);
        System.out.println("Called
sp1.bindContent(sp2)...");

        /* At this point, you can change the content of sp1.
However,
           * that will defeat the purpose of content binding,
because the
           * content of sp1 is no longer in sync with the
content of sp2.
           * Do not do this:
           * sp1.add("X");
           */
        print("Before sp2.add():", sp1, sp2);
        sp2.add("1");
        print("After sp2.add():", sp1, sp2);

        sp1.unbindContent(sp2);
        print("After sp1.unbindContent(sp2):", sp1, sp2);

        // Bind sp1 and sp2 contents bidirectionally
        sp1.bindContentBidirectional(sp2);

        print("Before sp2.add():", sp1, sp2);
        sp2.add("2");
        print("After sp2.add():", sp1, sp2);
    }

    public static void print(String msg, SetProperty<String>
sp1, SetProperty<String> sp2) {
        System.out.println(msg + " sp1: " + sp1.get() + ",
sp2: " + sp2.get());
    }
}

```

```
Before sp1.add(): Size: 0, Empty: true, Set: []
After sp1.add(): Size: 2, Empty: false, Set: [Jacobs, John]
Called sp1.bindContent(sp2)...
Before sp2.add(): sp1: [], sp2: []
After sp2.add(): sp1: [1], sp2: [1]
After sp1.unbindContent(sp2): sp1: [1], sp2: [1]
Before sp2.add(): sp1: [1], sp2: [1]
After sp2.add(): sp1: [1, 2], sp2: [2, 1]
```

## Understanding *ObservableMap* Property and Binding

A `MapProperty` object wraps an `ObservableMap`. Working with a `MapProperty` is very similar to working with a `ListProperty`. I am not going to repeat what has been discussed in the previous sections about properties and bindings of an `ObservableList`. The same discussions apply to properties and bindings of `ObservableMap`. The following are the salient points to remember while working with a `MapProperty`:

- The class diagram for the `MapProperty` class is similar to the one shown in Figure 3-5 for the `ListProperty` class. You need to replace the word “List” with the word “Map” in all names and the generic type parameter `<E>` with `<K, V>`, where K and V stand for the key type and value type, respectively, of entries in the map.
- The `MapExpression` and `Bindings` classes contain methods to support high-level bindings for map properties. You need to subclass the `MapBinding` class to create low-level bindings.
- Like the `ListProperty`, the `MapProperty` exposes `size` and `empty` properties.
- Like the `ListProperty`, the `MapProperty` supports bindings of the reference and the content of the `ObservableMap` that it wraps.
- Like the `ListProperty`, the `MapProperty` supports three types of notifications: invalidation notifications, change notifications, and map change notifications.
- The `MapProperty` supports binding to the value of a specific key using its `valueAt()` method.

Use one of the following constructors of the `SimpleMapProperty` class to create an instance of the `MapProperty`:

- SimpleMapProperty()
- SimpleMapProperty(Object bean, String name)
- SimpleMapProperty(Object bean, String name, ObservableMap<K, V> initialValue)
- SimpleMapProperty(ObservableMap<K, V> initialValue)

The following snippet of code creates an instance of the MapProperty and adds two entries. In the end, it gets the reference of the wrapped ObservableMap using the get() method:

```
// Create a MapProperty object
MapProperty<String, Double> mp =
    new SimpleMapProperty<String,
Double>(FXCollections.observableHashMap());

// Add two entries to the wrapped ObservableMap
mp.put("Ken", 8190.20);
mp.put("Jim", 8990.90);

// Get the wrapped map from the mp property
ObservableMap<String, Double> map = mp.get();
```

The program in Listing 3-20 shows how to use binding with MapProperty objects. It shows the content binding between two maps. You can also use unidirectional and bidirectional simple binding between two map properties to bind the references of the maps they wrap.

### ***Listing 3-20.*** Using Properties and Bindings for Observable Maps

```
// MapBindingTest.java
package com.jdojo.collections;

import javafx.beans.binding.ObjectBinding;
import javafx.beans.property.MapProperty;
import javafx.beans.property.SimpleMapProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;

public class MapBindingTest {
    public static void main(String[] args) {
        MapProperty<String, Double> mp1 =
            new
SimpleMapProperty<>(FXCollections.observableHashMap());

        // Create an object binding to bind mp1 to the value
        // of the key "Ken"
        ObjectBinding<Double> kenSalary
= mp1.valueAt("Ken");
        System.out.println("Ken Salary: "
```

```

+ kenSalary.get()));

        // Bind the size and empty properties of the
MapProperty
        // to create a description of the map
        StringProperty initStr = new
SimpleStringProperty("Size: ");
        StringProperty desc = new SimpleStringProperty();
        desc.bind(initStr.concat(mp1.sizeProperty()))
            .concat(", Empty: ")
            .concat(mp1.emptyProperty())
            .concat(", Map: ")
            .concat(mp1.asString())
            .concat(", Ken Salary: ")
            .concat(kenSalary));

        System.out.println("Before mp1.put(): "
+ desc.get());

        // Add some entries to mp1
        mp1.put("Ken", 7890.90);
        mp1.put("Jim", 9800.80);
        mp1.put("Lee", 6000.20);
        System.out.println("After mp1.put(): "
+ desc.get());

        // Create a new MapProperty
        MapProperty<String, Double> mp2 =
new
SimpleMapProperty<>(FXCollections.observableHashMap());

        // Bind the content of mp1 to the content of mp2
        mp1.bindContent(mp2);
        System.out.println("Called
mp1.bindContent(mp2)...");

        /* At this point, you can change the content of mp1.
However,
         * that will defeat the purpose of content binding,
because the
         * content of mp1 is no longer in sync with the
content of mp2.
         * Do not do this:
         * mp1.put("k1", 8989.90);
         */
        System.out.println("Before mp2.put(): "
+ desc.get());
        mp2.put("Ken", 7500.90);
        mp2.put("Cindy", 7800.20);
        System.out.println("After mp2.put(): "
+ desc.get());
    }
}

```

```
Ken Salary: null
Before mp1.put(): Size: 0, Empty: true, Map: {}, Ken Salary: null
After mp1.put(): Size: 3, Empty: false, Map: {Jim=9800.8,
Lee=6000.2, Ken=7890.9}, Ken Salary: 7890.9
Called mp1.bindContent(mp2)...
Before mp2.put(): Size: 0, Empty: true, Map: {}, Ken Salary: null
After mp2.put(): Size: 2, Empty: false, Map: {Cindy=7800.2,
Ken=7500.9}, Ken Salary: 7500.9
```

## Summary

JavaFX extends the collections framework in Java by adding support for observable lists, sets, and maps that are called observable collections. An observable collection is a list, set, or map that may be observed for invalidation and content changes. Instances of the `ObservableList`, `ObservableSet`, and `ObservableMap` interfaces in the `javafx.collections` package represent observable interfaces in JavaFX. You can add invalidation and change listeners to instances of these observable collections.

The `FXCollections` class is a utility class to work with JavaFX collections. It consists of all static methods. JavaFX does not expose the implementation classes of observable lists, sets, and maps. You need to use one of the factory methods in the `FXCollections` class to create objects of the `ObservableList`, `ObservableSet`, and `ObservableMap` interfaces.

JavaFX library provides two classes named `FilteredList` and `SortedList` that are in the `javafx.collections.transformation` package. A `FilteredList` is an `ObservableList` that filters its contents using a specified `Predicate`. A `SortedList` sorts its contents.

The next chapter will discuss how to create and customize stages in JavaFX applications.

## CHAPTER 4



### Managing Stages

---

In this chapter, you will learn:

- How to get details of screens such as their number, resolutions, and dimensions
- What a stage is in JavaFX and how to set bounds and styles of a stage
- How to move an undecorated stage
- How to set the modality and opacity of a stage
- How to resize a stage and how to show a stage in full-screen mode

### Knowing the Details of Your Screens

The `Screen` class in the `javafx.stage` package is used to get the details, for example, dots-per-inch (DPI) setting and dimensions of user screens (or monitors). If multiple screens are hooked up to a computer, one of the screens is known as the primary screen and others as nonprimary screens. You can get the reference of the `Screen` object for the primary monitor using the static `getPrimary()` method of the `Screen` class with the following code:

```
// Get the reference to the primary screen
Screen primaryScreen = Screen.getPrimary();
```

The static `getScreens()` method returns an `ObservableList` of `Screen` objects:

```
ObservableList<Screen> screenList = Screen.getScreens();
```

You can get the resolution of a screen in DPI using the `getDpi()` method of the `Screen` class as follows:

```
Screen primaryScreen = Screen.getPrimary();
double dpi = primaryScreen.getDpi();
```

You can use

the `getBounds()` and `getVisualBounds()` methods to get the bounds and visual bounds, respectively. Both methods return a `Rectangle2D` object, which encapsulates the (x, y) coordinates of the upper-left and the lower-right corners, the width, and the height of a rectangle. The `getMinX()` and `getMinY()` methods return the x and y coordinates of the upper-left corner of the rectangle, respectively.

The `getMaxX()` and `getMaxY()` methods return the x and y

coordinates of the lower-right corner of the rectangle, respectively. The `getWidth()` and `getHeight()` methods return the width and height of the rectangle, respectively.

The bounds of a screen cover the area that is available on the screen. The visual bounds represent the area on the screen that is available for use, after taking into account the area used by the native windowing system such as task bars and menus. Typically, but not necessarily, the visual bounds of a screen represents a smaller area than its bounds.

If a desktop spans multiple screens, the bounds of the nonprimary screens are relative to the primary screen. For example, if a desktop spans two screens with the (x, y) coordinates of the upper-left corner of the primary screen at (0, 0) and its width 1600, the coordinates of the upper-left corner of the second screen would be (1600, 0).

The program in Listing 4-1 prints the screens details when it was run on a Windows desktop with two screens. You may get a different output. Notice the difference in height for bounds and visual bounds for one screen and not for the other. The primary screen displays a task bar at the bottom that takes away some part of the height from the visual bounds. The nonprimary screen does not display a task bar, and therefore, its bounds and visual bounds are the same.

**Tip** Although it is not mentioned in the API documentation for the `Screen` class, you cannot use this class until the JavaFX launcher has started. That is, you cannot get screen descriptions in a non-JavaFX application. This is the reason that you would write the code in the `start()` method of a JavaFX application class. There is no requirement that the `Screen` class needs to be used on the JavaFX Application Thread. You could also write the same code in the `init()` method of your class.

### ***Listing 4-1.*** Accessing Screens Details

```
// ScreenDetailsApp.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.collections.ObservableList;
import javafx.geometry.Rectangle2D;
import javafx.stage.Screen;
import javafx.stage.Stage;

public class ScreenDetailsApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    public void start(Stage stage) {
```

```
ObservableList<Screen> screenList
= Screen.getScreens();
System.out.println("Screens Count: "
+ screenList.size());

// Print the details of all screens
for(Screen screen: screenList) {
    print(screen);
}

Platform.exit();
}

public void print(Screen s) {
    System.out.println("DPI: " + s.getDpi());

    System.out.print("Screen Bounds: ");
    Rectangle2D bounds = s.getBounds();
    print(bounds);

    System.out.print("Screen Visual Bounds: ");
    Rectangle2D visualBounds = s.getVisualBounds();
    print(visualBounds);
    System.out.println("-----");
}

public void print(Rectangle2D r) {
    System.out.format("minX=%f, minY=%f, width=%f,
height=%f%n",
r.getMinX(), r.getMinY(), r.getWidth(),
r.getHeight());
}
}

Screens Count: 2
DPI: 96.0
Screen Bounds: minX=0.00, minY=0.00, width=1680.00,
height=1050.00
Screen Visual Bounds: minX=0.00, minY=0.00, width=1680.00,
height=1022.00
-----
DPI: 96.0
Screen Bounds: minX = 1680.00, minY=0.00, width= 1680.00,
height=1050.00
Screen Visual Bounds: minX = 1680.00, minY=0.00, width= 1680.00,
height=1050.0
-----
```

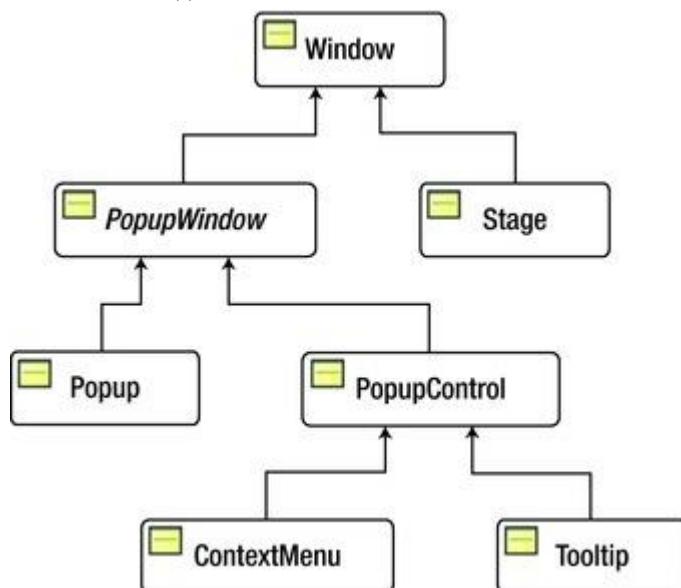
## What Is a Stage?

A stage in JavaFX is a top-level container that hosts a scene, which consists of visual elements. The `Stage` class in the `javafx.stage` package represents a stage in a JavaFX application.

The primary stage is created by the platform and passed to the `start(Stage s)` method of the `Application` class. You can create additional stages as needed.

**Tip** A stage in a JavaFX application is a top-level container. This does not mean that it is always displayed as a separate window. For example, in a web environment, the primary stage of a JavaFX application is embedded inside the browser window.

Figure 4-1 shows the class diagram for the `Stage` class, which inherits from the `Window` class. The `Window` class is the superclass for several window-like container classes. It contains the basic functionalities that are common to all types of windows (e.g., methods to show and hide the window, set x, y, width, and height properties, set the opacity of the window, etc.). The `Window` class defines `x`, `y`, `width`, `height`, and `opacity` properties. It has `show()` and `hide()` methods to show and hide a window, respectively. The `setScene()` method of the `Window` class sets the scene for a window. The `Stage` class defines a `close()` method, which has the same effect as calling the `hide()` method of the `Window` class.



**Figure 4-1.** The class diagram for the `Stage` class

A `Stage` object must be created and modified on the JavaFX Application Thread. Recall that the `start()` method of the `Application` class is called on the JavaFX Application Thread, and a primary `Stage` is created and passed to this method. Note that the primary stage that is passed the `start()` method is not shown. You need to call the `show()` method to show it.

Several aspects of working with stages need to be discussed. I will handle them one by one from the basic to the advanced level in the sections that follow.

## Showing the Primary Stage

Let's start with the simplest JavaFX application, as shown in Listing 4-2. The `start()` method has no code. When you run the application, you do not see a window, nor do you see output on the console. The application runs forever. You will need to use the system-specific keys to cancel the application. If you are using Windows, use your favorite key combination Ctrl + Alt + Del to activate the task manager! If you are using the command prompt, use Ctrl + C.

### ***Listing 4-2.*** An Ever-Running JavaFX Application

```
// EverRunningApp.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.stage.Stage;

public class EverRunningApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Do not write any code here
    }
}
```

To determine what is wrong with the program in Listing 4-2, you need to understand what the JavaFX application launcher does. Recall that JavaFX Application Thread is terminated when the `Platform.exit()` method is called or the last shown stage is closed. The JVM terminates when all nondaemon threads die. JavaFX Application Thread is a nondaemon thread. The `Application.launch()` method returns when the JavaFX Application Thread terminates. In the above example, there is no way to terminate the JavaFX Application Thread. This is the reason the application runs forever.

Using the `Platform.exit()` method in the `start()` method will fix the problem. The modified code for the `start()` method is shown in Listing 4-3. When you run the program, it exits without doing anything meaningful.

### ***Listing 4-3.*** A Short-Lived JavaFX Application

```
// ShortLivedApp.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;

public class ShortLivedApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Platform.exit(); // Exit the application
    }
}
```

Let's try to fix the ever-running program by closing the primary stage. You have only one stage when the `start()` method is called and closing it should terminate the JavaFX Application Thread. Let's modify the `start()` method of the `EverRunningApp` with the following code:

```
@Override
public void start(Stage stage) {
    stage.close(); // Close the only stage you have
}
```

Even with this code for the `start()` method, the `EverRunningApp` runs forever. The `close()` method does not close the stage if the stage is not showing. The primary stage was never shown. Therefore, adding a `stage.close()` call to the `start()` method did not do any good. The following code for the `start()` method would work. However, this will cause the screen to flicker as the stage is shown and closed:

```
@Override
public void start(Stage stage) {
    stage.show(); // First show the stage
    stage.close(); // Now close it
}
```

**Tip** The `close()` method of the `Stage` class has the same effect as calling the `hide()` method of the `Window` class. The JavaFX API documentation does not mention that attempting to close a not showing window has no effect.

## **Setting the Bounds of a Stage**

The bounds of a stage consist of four properties: `x`, `y`, `width`, and `height`. The `x` and `y` properties determine the location (or position)

of the upper-left corner of the stage.

The `width` and `height` properties determine its size. In this section, you will learn how to position and size a stage on the screen. You can use the getters and setters for these properties to get and set their values.

Let's start with a simple example as shown in Listing 4-4. The program sets the title for the primary stage before showing it. When you run this code, you would see a window with the title bar, borders, and an empty area. If other applications are open, you can see their content through the transparent area of the stage. The position and size of the window are decided by the platform.

**Tip** When a stage does not have a scene and its position and size are not set explicitly, its position and size are determined and set by the platform.

#### **Listing 4-4.** Displaying a Stage with No Scene and with the Platform Default Position and Size

```
// BlankStage.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.stage.Stage;

public class BlankStage extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("Blank Stage");
        stage.show();
    }
}
```

Let's modify the logic a bit. Here you will set an empty scene to the stage without setting the size of the scene. The modified `start()` method would look as follows:

```
import javafx.scene.Group;
import javafx.scene.Scene;
...
@Override
public void start(Stage stage) {
    stage.setTitle("Stage with an Empty Scene");
    Scene scene = new Scene(new Group());
    stage.setScene(scene);
    stage.show();
}
```

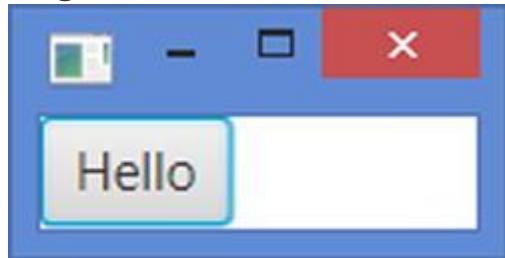
Notice that you have set a `Group` with no children nodes as the root node for the scene, because you cannot create a scene without a root

node. When you run the program in Listing 4-4 with the above code as its `start()` method, the position and size of the stage are determined by the platform. This time, the content area will have a white background, because the default background color for a scene is white.

Let's modify the logic again. Here let's add a button to the scene. The modified `start()` method would be as follows:

```
import javafx.scene.control.Button;
...
@Override
public void start(Stage stage) {
    stage.setTitle("Stage with a Button in the Scene");
    Group root = new Group(new Button("Hello"));
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
}
```

When you run the program in Listing 4-4 with the above code as its `start()` method, the position and size of the stage are determined by the computed size of the scene. The content area of the stage is wide enough to show the title bar menus or the content of the scene, whichever is bigger. The content area of the stage is tall enough to show the content of the scene, which in this case has only one button. The stage is centered on the screen, as shown in Figure 4-2.



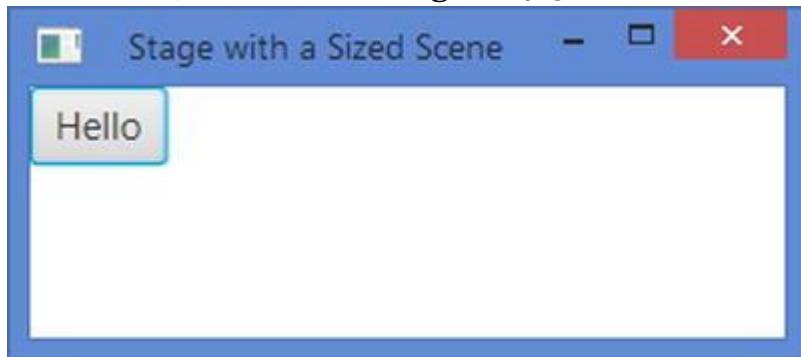
**Figure 4-2.** A stage with a scene that contains a button where the size of the scene is not specified

Let's add another twist to the logic by adding a button to the scene and set the scene width and height to 300 and 100, respectively, as follows:

```
@Override
public void start(Stage stage) {
    stage.setTitle("Stage with a Sized Scene");
    Group root = new Group(new Button("Hello"));
    Scene scene = new Scene(root, 300, 100);
    stage.setScene(scene);
    stage.show();
}
```

When you run the program in Listing 4-4 with the above code as its `start()` method, the position and size of the stage are determined by the specified size of the scene. The content area of the stage is the

same as the specified size of the scene. The width of the stage includes the borders on the two sides, and the height of the stage includes the height of the title bar and the bottom border. The stage is centered on the screen, as shown in Figure 4-3.

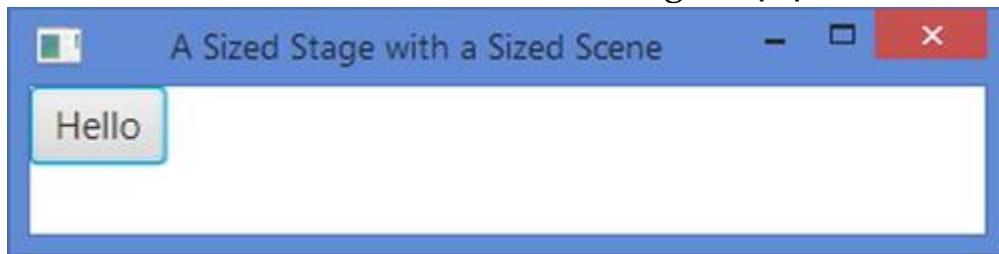


**Figure 4-3.** A stage with a scene with a specified size

Let's add one more twist to the logic. You will set the size of the scene and the stage using the following code:

```
@Override
public void start(Stage stage) {
    stage.setTitle("A Sized Stage with a Sized Scene");
    Group root = new Group(new Button("Hello"));
    Scene scene = new Scene(root, 300, 100);
    stage.setScene(scene);
    stage.setWidth(400);
    stage.setHeight(100);
    stage.show();
}
```

When you run the program in Listing 4-4 with the above code as its `start()` method, the position and size of the stage are determined by the specified size of the stage. The stage is centered on the screen and it will then look like the one shown in Figure 4-4.



**Figure 4-4.** A sized stage with a sized scene

**Tip** The default centering of a stage centers it horizontally on the screen. The y coordinate of the upper-left corner of the stage is one-third of the height of the screen minus the height of the stage. This is the logic used in the `centerOnScreen()` method in the `Window` class.

Let me recap the rules for positioning and resizing a stage. If you do not specify the bounds of a stage and:

- It has no scene, its bounds are determined by the platform.
- It has a scene with no visual nodes, its bounds are determined by the platform. In this case, the size of the scene is not specified.
- It has a scene with some visual nodes, its bounds are determined by the visual nodes in the scene. In this case, the size of the scene is not specified and the stage is centered in the screen.
- It has a scene and the size of the scene is specified, its bounds are determined by the specified size of the scene. The stage is centered on the screen.

If you specify the size of the stage but not its position, the stage is sized according the set size and centered on the screen, irrespective of the presence of a scene and the size of the scene. If you specify the position of the stage (x, y coordinates), it is positioned accordingly.

**Tip** If you want to set the width and height of a stage to fit the content of its scene, use the `sizeToScene()` method of the `Window` class. The method is useful if you want to synchronize the size of a stage with the size of its scene after modifying the scene at runtime. Use the `centerOnScreen()` method of the `Window` class to center the stage on the screen.

If you want to center a stage on the screen horizontally as well as vertically, use the following logic:

```
Rectangle2D bounds = Screen.getPrimary().getVisualBounds();
double x = bounds.getMinX() + (bounds.getWidth() -
    stage.getWidth()) / 2.0;
double y = bounds.getMinY() + (bounds.getHeight() -
    stage.getHeight()) / 2.0;
stage.setX(x);
stage.setY(y);
```

Be careful in using the above snippet of code. It makes use of the size of the stage. The size of a stage is not known until the stage is shown for the first time. Using the above logic before a stage is shown will not really center the stage on the screen. The following `start()` method of a JavaFX application will not work as intended:

```
@Override
public void start(Stage stage) {
    stage.setTitle("A Truly Centered Stage");
    Group root = new Group(new Button("Hello"));
    Scene scene = new Scene(root);
    stage.setScene(scene);
```

```

        // Wrong!!!! Use the logic shown below after the
        stage.show() call
        // At this point, stage width and height are not known.
        They are NaN.
        Rectangle2D bounds = Screen.getPrimary().getVisualBounds();
        double x = bounds.getMinX() + (bounds.getWidth() -
        stage.getWidth()) / 2.0;
        double y = bounds.getMinY() + (bounds.getHeight() -
        stage.getHeight()) / 2.0;
        stage.setX(x);
        stage.setY(y);

        stage.show();
    }
}

```

## Initializing the Style of a Stage

The area of a stage can be divided into two parts: content area and decorations. The content area displays the visual content of its scene. Typically, decorations consist of a title bar and borders. The presence of a title bar and its content varies depending on the type of decorations provided by the platform. Some decorations provide additional features rather than just an aesthetic look. For example, a title bar may be used to drag a stage to a different location; buttons in a title bar may be used to minimize, maximize, restore, and close a stage; or borders may be used to resize a stage.

In JavaFX, the style attribute of a stage determines its background color and decorations. Based on styles, you can have the following five types of stages in JavaFX:

- Decorated
- Undecorated
- Transparent
- Unified
- Utility

A *decorated* stage has a solid white background and platform decorations. An *undecorated* stage has a solid white background and no decorations. A *transparent* stage has a transparent background and no decorations. A *unified* stage has platform decorations and no border between the client area and decorations; the client area background is unified with the decorations. To see the effect of the unified stage style, the scene should be filled with `Color.TRANSPARENT`. Unified style is a conditional feature. A *utility* stage has a solid white background and minimal platform decorations.

**Tip** The style of a stage specifies only its decorations. The background color is controlled by its scene background, which is solid white by default. If you set

the style of a stage to TRANSPARENT, you will get a stage with a solid white background, which is the background of the scene. To get a truly transparent stage, you will need to set the background color of the scene to null using its `setFill()` method.

You can set the style of a stage using the `initStyle(StageStyle style)` method of the `Stage` class. The style of a stage must be set before it is shown for the first time. Setting it the second time, after the stage has been shown, throws a runtime exception. By default, a stage is decorated.

The five types of styles for a stage are defined as five constants in the `StageStyle` enum:

- `StageStyle.DECORATED`
- `StageStyle.UNDECORATED`
- `StageStyle.TRANSPARENT`
- `StageStyle.UNIFIED`
- `StageStyle.UTILITY`

**Listing 4-5** shows how to use these five styles for a stage. In the `start()` method, you need to uncomment only one statement at a time, which initializes the style of the stage. You will use a `VBox` to display two controls: a `Label` and a `Button`. The `Label` displays the style of the stage. The `Button` is provided to close the stage, because not all styles provide a title bar with a close button. Figure 4-5 shows the stage using four styles. The contents of windows in the background can be seen through a transparent stage. This is the reason that when you use the transparent style, you will see more content that has been added to the stage.

### ***Listing 4-5.*** Using Different Styles for a Stage

```
// StageStyleApp.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.stage.StageStyle;
import static javafx.stage.StageStyle.DECORATED;
import static javafx.stage.StageStyle.UNDECORATED;
import static javafx.stage.StageStyle.TRANSPARENT;
```

```
import static javafx.stage.StageStyle.UNIFIED;
import static javafx.stage.StageStyle.UTILITY;

public class StageStyleApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // A label to display the style type
        Label styleLabel = new Label("Stage Style");

        // A button to close the stage
        Button closeButton = new Button("Close");
        closeButton.setOnAction(e -> stage.close());

        VBox root = new VBox();
        root.getChildren().addAll(styleLabel, closeButton);
        Scene scene = new Scene(root, 100, 70);
        stage.setScene(scene);

        // The title of the stage is not visible for all
        styles.
        stage.setTitle("The Style of a Stage");

        /* Uncomment one of the following statements at
        a time */
        this.show(stage, styleLabel, DECORATED);
        //this.show(stage, styleLabel, UNDECORATED);
        //this.show(stage, styleLabel, TRANSPARENT);
        //this.show(stage, styleLabel, UNIFIED);
        //this.show(stage, styleLabel, UTILITY);
    }

    private void show(Stage stage, Label styleLabel, StageStyle
style) {
        // Set the text for the label to match the style
        styleLabel.setText(style.toString());

        // Set the style
        stage.initStyle(style);

        // For a transparent style, set the scene fill to
        null. Otherwise, the
        // content area will have the default white
        background of the scene.
        if (style == TRANSPARENT) {
            stage.getScene().setFill(null);
            stage.getScene().getRoot().setStyle(
                "-fx-background-color: transparent");
        } else if(style == UNIFIED) {
            stage.getScene().setFill(Color.TRANSPARENT);
        }
    }
}
```

```

        // Show the stage
        stage.show();
    }
}

```



**Figure 4-5.** A stage using different styles

## Moving an Undecorated Stage

You can move a stage to a different location by dragging its title bar. In an undecorated or transparent stage, a title bar is not available. You need to write a few lines of code to let the user move this kind of stage by dragging the mouse over the scene area. Listing 4-6 shows how to write the code to support dragging of a stage. If you change the stage to be transparent, you will need to drag the stage by dragging the mouse over only the message label, as the transparent area will not respond to the mouse events.

This example uses mouse event handling. I will cover event handling in detail in Chapter 9. It is briefly presented here to complete the discussion on using different styles of a stage.

### **Listing 4-6.** Dragging a Stage

```

// DraggingStage.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;

import javafx.stage.Stage;
import javafx.stage.StageStyle;

public class DraggingStage extends Application {
    private Stage stage;
    private double dragOffsetX;

```

```

private double dragOffsetY;

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

@Override
public void start(Stage stage) {
    // Store the stage reference in the instance variable
to
    // use it in the mouse pressed event handler later.
    this.stage = stage;

    Label msgLabel = new Label("Press the mouse button
and drag.");
    Button closeButton = new Button("Close");
    closeButton.setOnAction(e -> stage.close());

    VBox root = new VBox();
    root.getChildren().addAll(msgLabel, closeButton);

    Scene scene = new Scene(root, 300, 200);

    // Set mouse pressed and dragged even handlers for
the scene
    scene.setOnMousePressed(e -> handleMousePressed(e));
    scene.setOnMouseDragged(e -> handleMouseDragged(e));

    stage.setScene(scene);
    stage.setTitle("Moving a Stage");
    stage.initStyle(StageStyle.UNDECORATED);
    stage.show();
}

protected void handleMousePressed(MouseEvent e) {
    // Store the mouse x and y coordinates with respect
to the
    // stage in the reference variables to use them in
the drag event
    this.dragOffsetX = e.getScreenX() - stage.getX();
    this.dragOffsetY = e.getScreenY() - stage.getY();
}

protected void handleMouseDragged(MouseEvent e) {
    // Move the stage by the drag amount
    stage.setX(e.getScreenX() - this.dragOffsetX);
    stage.setY(e.getScreenY() - this.dragOffsetY);
}
}

```

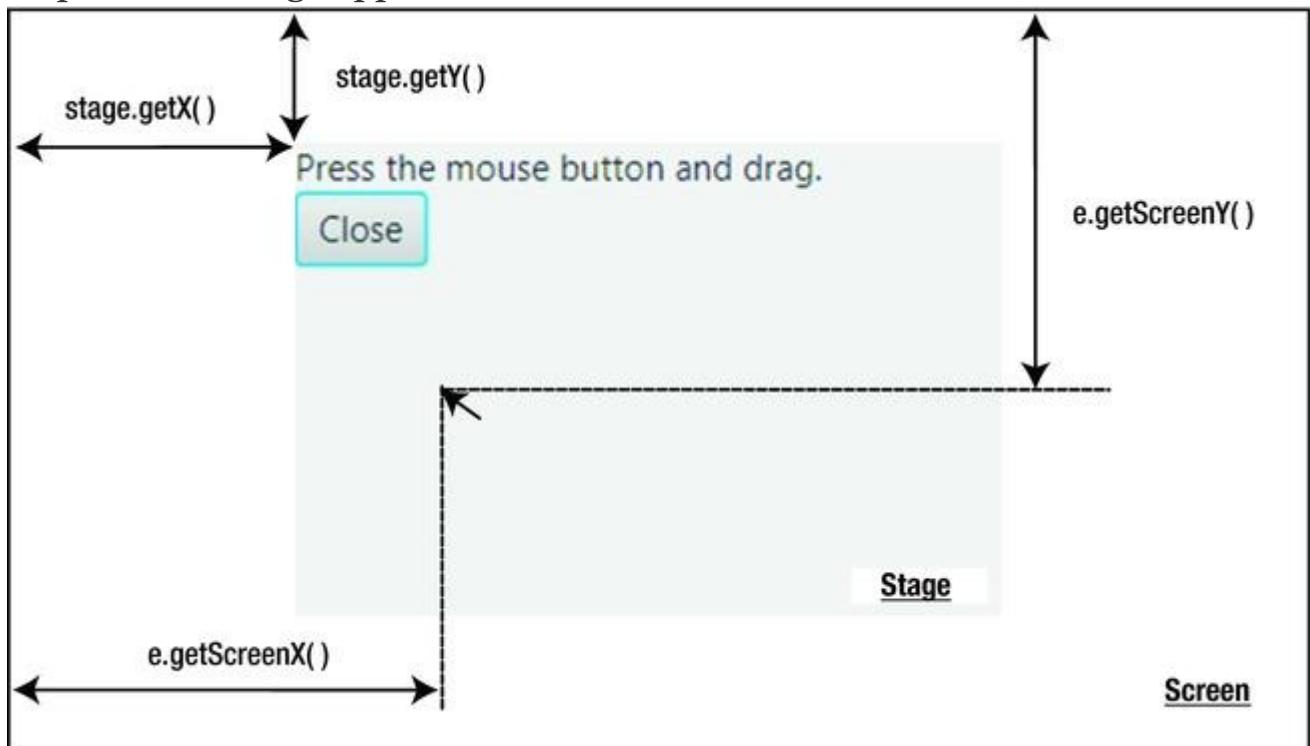
The following snippet of code adds the mouse pressed and mouse dragged event handlers to the scene:

```

scene.setOnMousePressed(e -> handleMousePressed(e));
scene.setOnMouseDragged(e -> handleMouseDragged(e));

```

When you press the mouse in the scene (except the button area), the `handleMousePressed()` method is called. The `getScreenX()` and `getScreenY()` methods of the `MouseEvent` object return the x and y coordinates of the mouse with respect to the upper-left corner of the screen. Figure 4-6 shows a diagrammatic view of the coordinate systems. It shows a thin border around the stage. However, when you run the example code, you will not see any border. This is shown here to distinguish the screen area from the stage area. You store the x and y coordinates of the mouse with respect to the stage upper-left corner in instance variables.



**Figure 4-6.** Computing the mouse coordinates with respect to the stage

When you drag the mouse, the `handleMouseDragged()` method is called. The method computes and sets the position of the stage using the position of the mouse when it was pressed and its position during the drag.

## Initializing Modality of a Stage

In a GUI application, you can have two types of windows: modal and modeless. When a modal window is displayed, the user cannot work with other windows in the application until the modal window is dismissed. If an application has multiple modeless windows showing, the user can switch between them at any time.

JavaFX has three types of modality for a stage:

- None
- Window modal
- Application modal

Modality of a stage is defined by one of the following three constants in the `Modality` enum in the `javafx.stage` package:

- `NONE`
- `WINDOW_MODAL`
- `APPLICATION_MODEL`

You can set the modality of a stage using the `initModality(Modality m)` method of the `Stage` class as follows:

```
// Create a Stage object and set its modality
Stage stage = new Stage();
stage.initModality(Modality.WINDOW_MODAL);

/* More code goes here.*/

// Show the stage
stage.show();
```

**Tip** The modality of a stage must be set before it is shown. Setting the modality of a stage after it has been shown throws a runtime exception. Setting the modality for the primary stage also throws a runtime exception.

A `Stage` can have an owner. An owner of a `Stage` is another `Window`. You can set an owner of a `Stage` using the `initOwner(Window owner)` method of the `Stage` class. The owner of a `Stage` must be set before the stage is shown. The owner of a `Stage` may be `null`, and in this case, it is said that the `Stage` does not have an owner. Setting an owner of a `Stage` creates an owner-owned relationship. For example, a `Stage` is minimized or hidden if its owner is minimized or hidden, respectively.

The default modality of a `Stage` is `NONE`. When a `Stage` with the modality `NONE` is displayed, it does not block any other windows in the application. It behaves as a modeless window.

A `Stage` with the `WINDOW_MODAL` modality blocks all windows in its owner hierarchy. Suppose there are four stages: `s1`, `s2`, `s3`, and `s4`. Stages `s1` and `s4` have modalities set to `NONE` and do not have an owner; `s1` is the owner of `s2`; `s2` is the owner of `s3`. All four stages are displayed. If `s3` has its modality set to `WINDOW_MODAL`, you can work with `s3` or `s4`, but not with `s2` and `s1`. The owner-owned relationship is defined as `s1` to `s2` to `s3`. When `s3` is displayed, it blocks `s2` and `s1`, which are in its owner

hierarchy. Because s4 is not in the owner hierarchy of s3, you can still work with s4.

**Tip** The modality of WINDOW\_MODAL for a stage that has no owner has the same effect as if the modality is set to NONE.

If a Stage with its modality set to APPLICATION\_MODAL is displayed, you must work with the Stage and dismiss it before you can work with any other windows in the application. Continuing with the same example from the previous paragraph of displaying four stages, if you set the modality of s4 to APPLICATION\_MODAL, the focus will be set to s4 and you must dismiss it before you can work with other stages. Notice that an APPLICATION\_MODAL stage blocks all other windows in the same application, irrespective of the owner-owned relationships.

Listing 4-7 shows how to use different modalities for a stage. It displays the primary stage with six buttons. Each button opens a secondary stage with a specified modality and owner. The text of the buttons tells you what kind of secondary stage they will open. When the secondary stage is shown, try clicking on the primary stage. When the modality of the secondary stage blocks the primary stage, you will not be able to work with the primary stage; clicking the primary stage will set the focus back to the secondary stage.

### ***Listing 4-7.*** Using Different Modalities for a Stage

```
// StageModalityApp.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.stage.Modality;
import static javafx.stage.Modality.NONE;
import static javafx.stage.Modality.WINDOW_MODAL;
import static javafx.stage.Modality.APPLICATION_MODAL;
import javafx.stage.Window;

public class StageModalityApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        /* Buttons to display each kind of modal stage */
        Button ownedNoneButton = new Button("Owned None");
        ...
    }
}
```

```

        ownedNoneButton.setOnAction(e -> showDialog(stage,
NONE));

        Button nonOwnedNoneButton = new Button("Non-owned
None");
        nonOwnedNoneButton.setOnAction(e -> showDialog(null,
NONE));

        Button ownedWinButton = new Button("Owned Window
Modal");
        ownedWinButton.setOnAction(e -> showDialog(stage,
WINDOW_MODAL));

        Button nonOwnedWinButton = new Button("Non-owned
Window Modal");
        nonOwnedWinButton.setOnAction(e -> showDialog(null,
WINDOW_MODAL));

        Button ownedAppButton = new Button("Owned Application
Modal");
        ownedAppButton.setOnAction(e -> showDialog(stage,
APPLICATION_MODAL));

        Button nonOwnedAppButton = new Button("Non-owned
Application Modal");
        nonOwnedAppButton.setOnAction(e -> showDialog(null,
APPLICATION_MODAL));

        VBox root = new VBox();
        root.getChildren().addAll(ownedNoneButton,
nonOwnedNoneButton,
nonOwnedWinButton,
nonOwnedAppButton);
        ownedWinButton,
        ownedAppButton,
        Scene scene = new Scene(root, 300, 200);
        stage.setScene(scene);
        stage.setTitle("The Primary Stage");
        stage.show();
    }

    private void showDialog(Window owner, Modality modality) {
        // Create a Stage with specified owner and modality
        Stage stage = new Stage();
        stage.initOwner(owner);
        stage.initModality(modality);

        Label modalityLabel = new Label(modality.toString());
        Button closeButton = new Button("Close");
        closeButton.setOnAction(e -> stage.close());

        VBox root = new VBox();
        root.getChildren().addAll(modalityLabel,
closeButton);
        Scene scene = new Scene(root, 200, 100);
    }
}

```

```

        stage.setScene(scene);
        stage.setTitle("A Dialog Box");
        stage.show();
    }
}

```

## Setting the Opacity of a Stage

The opacity of a stage determines how much you can see through the stage. You can set the opacity of a stage using the `setOpacity(double opacity)` method of the `Window` class. Use the `getOpacity()` method to get the current opacity of a stage.

The opacity value ranges from 0.0 to 1.0. Opacity of 0.0 means the stage is fully translucent; opacity of 1.0 means the stage is fully opaque. Opacity affects the entire area of a stage, including its decorations. Not all JavaFX runtime platforms are required to support opacity. Setting opacity on the JavaFX platforms that do not support opacity has no effect. The following snippet of code sets the opacity of a state to half-translucent:

```

Stage stage = new Stage();
stage.setOpacity(0.5); // A half-translucent stage

```

## Resizing a Stage

You can set whether a user can or cannot resize a stage by using its `setResizable(boolean resizable)` method. Note that a call to the `setResizable()` method is a *hint* to the implementation to make the stage resizable. By default, a stage is resizable. Sometimes, you may want to restrict the use to resize a stage within a range of width and height. The `setMinWidth()`, `setMinHeight()`, `setMaxWidth()`, and `setMaxHeight()` methods of the `Stage` class let you set the range within which the user can resize a stage.

**Tip** Calling the `setResizable(false)` method on a `Stage` object prevents the user from resizing the stage. You can still resize the stage programmatically.

It is often required to open a window that takes up the entire screen space. To achieve this, you need to set the position and size of the window to the available visual bounds of the screen. Listing 4-8 provides the program to illustrate this. It opens an empty stage, which takes up the entire visual area of the screen.

**Listing 4-8.** Opening a Stage to Take Up the Entire Available Visual Screen Space

```

// MaximizedStage.java
package com.jdojo.stage;

import javafx.application.Application;
import javafx.geometry.Rectangle2D;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Screen;
import javafx.stage.Stage;

public class MaximizedStage extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setScene(new Scene(new Group()));
        stage.setTitle("A Maximized Stage");

        // Set the position and size of the stage equal to
        // the position and
        // size of the screen
        Rectangle2D visualBounds
= Screen.getPrimary().getVisualBounds();
        stage.setX(visualBounds.getMinX());
        stage.setY(visualBounds.getMinY());
        stage.setWidth(visualBounds.getWidth());
        stage.setHeight(visualBounds.getHeight());

        // Show the stage
        stage.show();
    }
}

```

## Showing a Stage in Full-Screen Mode

The `Stage` class has a `fullScreen` property that specified whether a stage should be displayed in full-screen mode. The implementation of full-screen mode depends on the platform and profile. If the platform does not support full-screen mode, the JavaFX runtime will simulate it by displaying the stage maximized and undecorated. A stage may enter full-screen mode by calling the `setFullScreen(true)` method. When a stage enters full-screen mode, a brief message is displayed about how to exit the full-screen mode: You will need to press the ESC key to exit full-screen mode. You can exit full-screen mode programmatically by calling the `setFullScreen(false)` method. Use the `isFullScreen()` method to check if a stage is in full-screen mode.

## Showing a Stage and Waiting for It to Close

You often want to display a dialog box and suspend further processing until it is closed. For example, you may want to display a message box to the user with options to click yes and no buttons, and you want different actions performed based on which button is clicked by the user. In this case, when the message box is displayed to the user, the program must wait for it to close before it executes the next sequence of logic. Consider the following pseudo-code:

```
Option userSelection = messageBox("Close", "Do you want to
exit?", YESNO);
if (userSelection == YES) {
    stage.close();
}
```

In this pseudo-code, when the `messageBox()` method is called, the program needs to wait to execute the subsequent `if` statement until the message box is dismissed.

The `show()` method of the `Window` class returns immediately, making it useless to open a dialog box in the above example. You need to use the `showAndWait()` method, which shows the stage and waits for it to close before returning to the caller. The `showAndWait()` method stops processing the current event temporarily and starts a nested event loop to process other events.

**Tip** The `showAndWait()` method must be called on the JavaFX Application Thread. It should not be called on the primary stage or a runtime exception will be thrown.

You can have multiple stages open using the `showAndWait()` method. Each call to the method starts a new nested event loop. A specific call to the method returns to the caller when all nested event loops created after this method call have terminated.

This rule may be confusing in the beginning. Let's look at an example to explain this in detail. Suppose you have three stages: `s1`, `s2`, and `s3`. Stage `s1` is opened using the call `s1.showAndWait()`. From the code in `s1`, stage `s2` is opened using the call `s2.showAndWait()`. At this point, there are two nested event loops: one created

by `s1.showAndWait()` and another by `s2.showAndWait()`. The call to `s1.showAndWait()` will return only after both `s1` and `s2` have been closed, irrespective of the order they were closed.

The `s2.showAndWait()` call will return after `s2` has been closed.

Listing 4-9 contains a program that will allow you to play with the `showAndWait()` method call using multiple stages. The primary stage is opened with an Open button. Clicking the Open button opens a secondary stage using the `showAndWait()` method. The secondary stage has two buttons—Say Hello and Open—which will, respectively,

will print a message on the console and open another secondary stage. A message is printed on the console before and after the call to the `showAndWait()` method. You need to open multiple secondary stages, print messages by clicking the `Say Hello` button, close them in any order you want, and then look at the output on the console.

### ***Listing 4-9.*** Playing with `showAndWait()` Call

```
// ShowAndWaitApp.java
package com.jdojo.stage;

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

public class ShowAndWaitApp extends Application {
    protected static int counter = 0;
    protected Stage lastOpenStage;

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

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        Button openButton = new Button("Open");
        openButton.setOnAction(e -> open(++counter));
        root.getChildren().add(openButton);
        Scene scene = new Scene(root, 400, 400);
        stage.setScene(scene);
        stage.setTitle("The Primary Stage");
        stage.show();

        this.lastOpenStage = stage;
    }

    private void open(int stageNumber) {
        Stage stage = new Stage();
        stage.setTitle("#" + stageNumber);

        Button sayHelloButton = new Button("Say Hello");
        sayHelloButton.setOnAction(
            e -> System.out.println("Hello from #" +
+ stageNumber));

        Button openButton = new Button("Open");
        openButton.setOnAction(e -> open(++counter));

        VBox root = new VBox();
        root.getChildren().addAll(sayHelloButton,
```

```

openButton);
    Scene scene = new Scene(root, 200, 200);
    stage.setScene(scene);
    stage.setX(this.lastOpenStage.getX() + 50);
    stage.setY(this.lastOpenStage.getY() + 50);
    this.lastOpenStage = stage;

    System.out.println("Before stage.showAndWait(): " +
+ stageNumber);

    // Show the stage and wait for it to close
    stage.showAndWait();

    System.out.println("After stage.showAndWait(): " +
+ stageNumber);
}
}

```

**Tip** JavaFX does not provide a built-in window that can be used as a dialog box (a message box or a prompt window). You can develop one by setting the appropriate modality for a stage and showing it using the `showAndWait()` method.

## Summary

The `Screen` class in the `javafx.stage` package is used to obtain the details, such as the DPI setting and dimensions, of the user's screens hooked to the machine running the program. If multiple screens are present, one of the screens is known as the primary screen and the others are the nonprimary screens. You can get the reference of the `Screen` object for the primary monitor using the `static getPrimary()` method of the `Screen` class.

A stage in JavaFX is a top-level container that hosts a scene, which consists of visual elements. The `Stage` class in the `javafx.stage` package represents a stage in a JavaFX application. The primary stage is created by the platform and passed to the `start(Stage s)` method of the `Application` class. You can create additional stages as needed.

A stage has bounds that comprise its position and size. The bounds of a stage are defined by its four properties: `x`, `y`, `width`, and `height`. The `x` and `y` properties determine the location (or position) of the upper-left corner of the stage. The `width` and `height` properties determine its size.

The area of a stage can be divided into two parts: content area and decorations. The content area displays the visual content of its scene. Typically, decorations consist of a title bar and borders. The presence of a title bar and its content vary depending on the type of decorations

provided by the platform. You can have five types of stages in JavaFX: decorated, undecorated, transparent, unified, and utility.

JavaFX allows you to have two types of windows: modal and modeless. When a modal window is displayed, the user cannot work with other windows in the application until the modal window is dismissed. If an application has multiple modeless windows showing, the user can switch between them at any time. JavaFX defines three types of modality for a stage: none, window modal, and application modal. A stage with none as its modality is modeless window. A stage with window modal as its modality blocks all windows in its owner hierarchy. A stage with application modal as its modality blocks all other windows in the application.

The opacity of a stage determines how much you can see through the stage. You can set the opacity of a stage using its `setOpacity(double opacity)` method. The opacity value ranges from 0.0 to 1.0. Opacity of 0.0 means the stage is fully translucent; the opacity of 1.0 means the stage is fully opaque. Opacity affects the entire area of a stage, including its decorations.

You can set a hint whether a user can resize a stage by using its `setResizable(boolean resizable)` method.

The `setMinWidth()`, `setMinHeight()`, `setMaxWidth()`, and `setMaxHeight()` methods of the `Stage` class let you set the range within which the user can resize a stage. A stage may enter full-screen mode by calling its `setFullScreen(true)` method.

You can use the `show()` and `showAndWait()` methods of the `Stage` class to show a stage. The `show()` method shows the stage and returns, whereas the `showAndWait()` method shows the stage and blocks until the stage is closed.

The next chapter will show you how to create scenes and work with scene graphs.

## CHAPTER 5



### Making Scenes

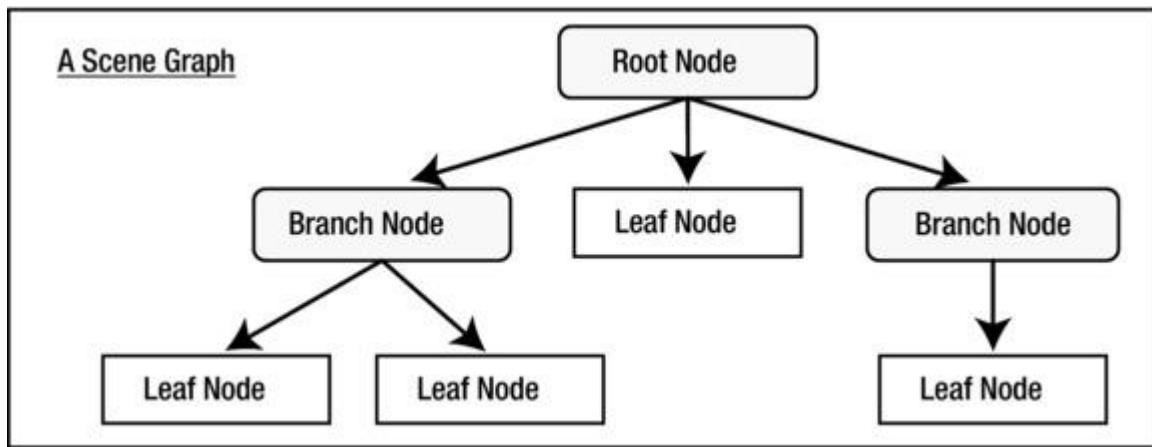
In this chapter, you will learn:

- What a scene and a scene graph are in a JavaFX application
- About different rendering modes of a scene graph
- How to set the cursor for a scene
- How to determine the focus owner in a scene
- How to use the `Platform` and `HostServices` classes

### What Is a Scene?

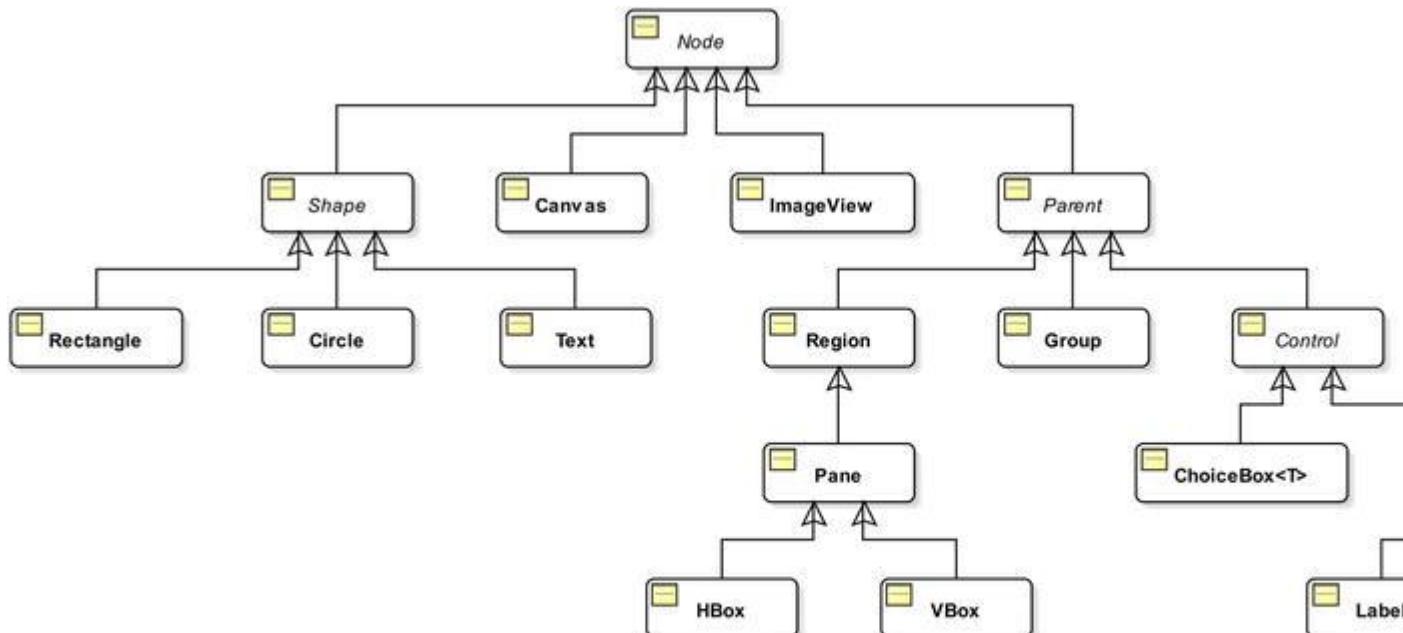
A *scene* represents the visual contents of a stage. The `Scene` class in the `javafx.scene` package represents a scene in a JavaFX program. A `Scene` object is attached to, at the most, one stage at a time. If an already attached scene is attached to another stage, it is first detached from the previous stage. A stage can have, at the most, one scene attached to it at any time.

A scene contains a scene graph that consists of visual nodes. In this sense, a scene acts as a container for a scene graph. A scene graph is a tree data structure whose elements are known as *nodes*. Nodes in a scene graph form a parent-child hierarchical relationship. A node in a scene graph is an instance of the `javafx.scene.Node` class. A node can be a branch node or a leaf node. A branch node can have children nodes, whereas a leaf node cannot. The first node in a scene graph is called the *root* node. The root node can have children nodes; however, it never has a parent node. Figure 5-1 shows the arrangement of nodes in a scene graph. Branch nodes are shown in rounded rectangles and leaf nodes in rectangles.



**Figure 5-1.** The arrangement of nodes in a scene graph

The JavaFX class library provides many classes to represent branch and leaf nodes in a scene graph. The `Node` class in the `javafx.scene` package is the superclass of all nodes in a scene graph. Figure 5-2 shows a partial class diagram for classes representing nodes.



**Figure 5-2.** A partial class diagram for the `javafx.scene.Node` class

A scene always has a root node. If the root node is resizable, for example, a `Region` or a `Control`, it tracks the size of the scene. That is, if the scene is resized, the resizable root node resizes itself to fill the

entire scene. Based on the policy of a root node, the scene graph may be laid out again when the size of the scene changes.

A `Group` is a nonresizable `Parent` node that can be set as the root node of a scene. If a `Group` is the root node of a scene, the content of the scene graph is clipped by the size of the scene. If the scene is resized, the scene graph is not laid out again.

`Parent` is an abstract class. It is the base class for all branch nodes in a scene graph. If you want to add a branch node to a scene graph, use objects of one of its concrete subclasses, for example, `Group`, `Pane`, `HBox`, or `VBox`. Classes that are subclasses of the `Node` class, but not the `Parent` class, represent leaf nodes, for example, `Rectangle`, `Circle`, `Text`, `Canvas`, or `ImageView`. The root node of a scene graph is a special branch node that is the topmost node. This is the reason you use a `Group` or a `VBox` as the root node while creating a `Scene` object. I will discuss classes representing branch and leaf nodes in detail in Chapters 10 and 12. Table 5-1 lists some of the commonly used properties of the `Scene` class.

**Table 5-1.** Commonly Used Properties of the `Scene` Class

Type	Name	Property and Description
<code>ObjectProperty&lt;Cursor&gt;</code>	<code>cursor</code>	It defines the mouse cursor for the <code>Scene</code> .
<code>ObjectProperty&lt;Paint&gt;</code>	<code>fill</code>	It defines the background fill for the <code>Scene</code> .
<code>ReadOnlyObjectProperty&lt;Node&gt;</code>	<code>focusOwner</code>	It defines the node in the <code>Scene</code> that owns the focus.
<code>ReadOnlyDoubleProperty</code>	<code>height</code>	It defines the height of the <code>Scene</code> .
<code>ObjectProperty&lt;Parent&gt;</code>	<code>root</code>	It defines the root <code>Node</code> of the scene graph.
<code>ReadOnlyDoubleProperty</code>	<code>width</code>	It defines the width of the <code>Scene</code> .

Type	Name	Property and Description
ReadOnlyObjectProperty<Window>	window	It defines the Window for the Scene.
ReadOnlyDoubleProperty	x	It defines the horizontal location of the Scene on the Window.
ReadOnlyDoubleProperty	y	It defines the vertical location of the Scene on the window.

## Graphics Rendering Modes

The scene graph plays a vital role in rendering the content of a JavaFX application on the screen. Typically, two types of APIs are used to render graphics on a screen:

- Immediate mode API
- Retained mode API

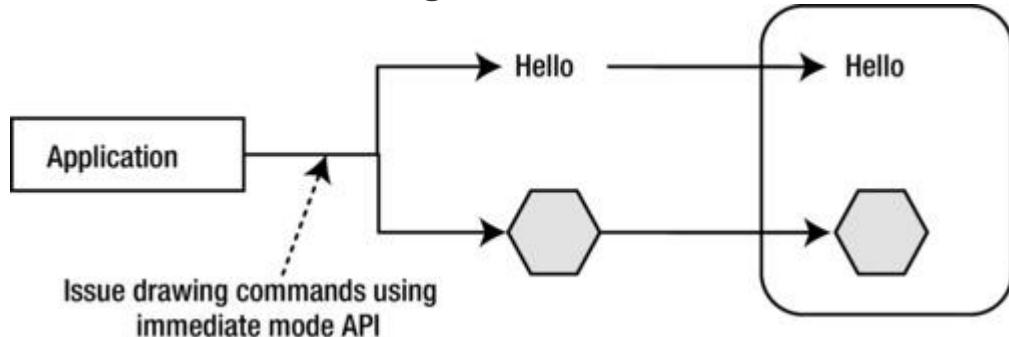
In immediate mode API, the application is responsible for issuing the drawing commands when a frame is needed on the screen. The graphics are drawn directly on the screen. When the screen needs to be repainted, the application needs to reissue the drawing commands to the screen. Java2D is an example of the immediate mode graphics-rendering API.

In retained mode API, the application creates and attaches drawing objects to a graph. The graphics library, not the application code, retains the graph in memory. Graphics are rendered on the screen by the graphics library when needed. The application is responsible only for creating the graphic objects—the “what” part; the graphics library is responsible for storing and rendering the graphics—the “when” and “how” parts. Retained mode rendering API relieves developers of writing the logic for rendering the graphics. For example, adding or removing part of a graphic from a screen is simple by adding or removing a graphic object from the graph using high-level APIs; the graphics library takes care of the rest. In comparison to the immediate mode, retained mode API uses more memory, as the graph is stored in memory. The JavaFX scene graph uses retained mode APIs.

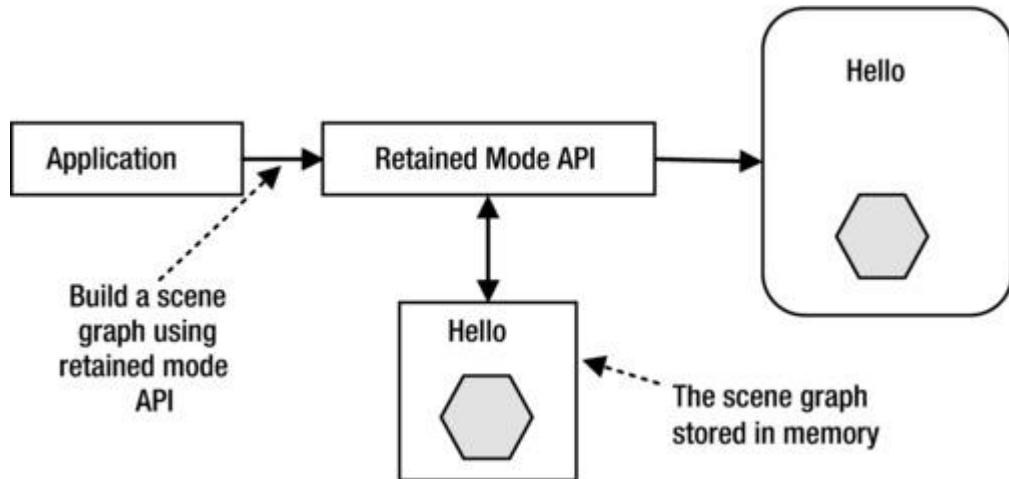
You might think that using immediate mode API would always be faster than using retained mode API because the former renders graphics directly on the screen. However, using retained mode API opens the door for optimizations by the class library that is not possible

in the immediate mode where every developer is in charge of writing the logic as to what and when it should be rendered.

Figures 5-3 and 5-4 illustrate how immediate and retained mode APIs work, respectively. They show how a text, Hello, and a hexagon are drawn on the screen using the two APIs.



**Figure 5-3.** An illustration of the immediate mode API



**Figure 5-4.** An illustration of the retained mode API

## Setting the Cursor for a Scene

An instance of the `javafx.scene.Cursor` class represents a mouse cursor. The `Cursor` class contains many constants, for example, `HAND`, `CLOSED_HAND`, `DEFAULT`, `TEXT`, `NONE`, `WAIT`, for standard mouse cursors. The following snippet of code sets the `WAIT` cursor for a scene:

```
Scene scene;
...
scene.setCursor(Cursor.WAIT);
```

You can also create and set a custom cursor to a scene.

The `cursor(String name)` static method of the `Cursor` class returns a standard cursor if the specified `name` is the name of a standard cursor. Otherwise, it treats the specified `name` as a URL for the cursor bitmap.

The following snippet of code creates a cursor from a bitmap file named mycur.png, which is assumed to be in the CLASSPATH:

```
// Create a Cursor from a bitmap
URL url = getClass().getClassLoader().getResource("mycur.png");
Cursor myCur = Cursor.cursor(url.toExternalForm());
scene.setCursor(myCur);

// Get the WAIT standard cursor using its name
Cursor waitCur = Cursor.cursor("WAIT")
scene.setCursor(waitCur);
```

## The Focus Owner in a Scene

Only one node in a scene can be the focus owner.

The `focusOwner` property of the `Scene` class tracks the `Node` class that has the focus. Note that the `focusOwner` property is read-only. If you want a specific node in a scene to be the focus owner, you need to call the `requestFocus()` method of the `Node` class.

You can use the `getFocusOwner()` method of the `Scene` class to get the reference of the node having the focus in the scene. A scene may not have a focus owner, and in that case, the `getFocusOwner()` method returns `null`. For example, a scene does not have a focus owner when it is created but is not attached to a window.

It is important to understand the distinction between a focus owner and a node having focus. Each scene may have a focus owner. For example, if you open two windows, you will have two scenes and you can have two focus owners. However, only one of the two focus owners can have the focus at a time. The focus owner of the active window will have the focus. To check if the focus owner node also has the focus, you need to use the `focused` property of the `Node` class. The following snippet of code shows the typical logic in using the focus owner:

```
Scene scene;
...
Node focusOwnerNode = scene.getFocusOwner();
if (focusOwnerNode == null) {
    // The scene does not have a focus owner
}
else if (focusOwnerNode.isFocused()) {
    // The focus owner is the one that has the focus
}
else {
    // The focus owner does not have the focus
}
```

## Using Builder Classes

JavaFX provides two classes for creating and configuring objects that constitute the building blocks of a scene graph. One class is named after the type of object that the class represents; another with the former class name suffixed with the word “Builder.” For example, `Rectangle` and `RectangleBuilder` classes exist to work with rectangles, `Scene` and `SceneBuilder` classes exist to work with scenes, and so on.

**Note** As of JavaFX 8, builder classes have been deprecated and they are not visible in the API documentation. This section is provided in case you need to maintain JavaFX code written in the older version such as version 2. Do not use the builder classes in JavaFX 8 or later. If you do not have to look at older version of JavaFX code, you can skip this section.

Builder classes provide three types of methods:

- They have a `create()` static method to create an instance of the builder class.
- They contain methods to set properties. Method names are the same as the property names that they set.
- They have a `build()` method that returns the object of the class for which the builder class exists. For example, the `build()` method of the `RectangleBuilder` class returns an object of the `Rectangle` class.

Builder classes are designed to use method chaining. Their methods to configure properties return the same builder instance. Assuming that `p1` and `p2` are properties of an object of `XXX` type, the following statement uses the builder class to create an object of the `XXX` type. It sets the properties `p1` and `p2` to `v1` and `v2`, respectively:

```
XXX x = XXXBuilder.create()
    .p1(v1)
    .p2(v2)
    .build();
```

The following snippet of code creates a rectangle, using the `Rectangle` class, with  $(x, y)$  coordinates at  $(10, 20)$ , with a width of 100px and a height of 200px. It also sets the fill property to red:

```
Rectangle r1 = new Rectangle(10, 20, 100, 200);
r1.setFill(Color.RED);
```

You can use the `RectangleBuilder` class to create the same rectangle:

```
Rectangle r1 = RectangleBuilder.create()
    .x(10)
    .y(20)
    .width(100)
    .height(200)
```

```
.fill(Color.RED)
.build();
```

Using builder classes requires longer code; however, it is more readable compared to using constructors to set the properties. Another advantage of builder classes is that they can be reused to build objects with slightly different properties. Suppose you want to create multiple rectangles with a 100px width and a 200px height, filled with the color red. However, they have different x and y coordinates. You can do so with the following code:

```
// Create a partially configured RectangleBuilder
RectangleBuilder builder = RectangleBuilder.create()
    .width(100)
    .height(200)
    .fill(Color.RED);

// Create a Rectangles at (10, 20) and (120, 20) using the
builder
Rectangle r3 = builder.x(10).y(20).build();
Rectangle r4 = builder.x(120).y(20).build();
```

The program in Listing 5-1 constructs a scene graph using builder classes. It adds a Label and a Button to a VBox. It also sets an action event handler for the button. The resulting screen is shown in Figure 5-5.

### ***Listing 5-1.*** Using Builder Classes to Construct Scene Graphs

```
// BuilderApp.java
package com.jdojo.scene;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.SceneBuilder;
import javafx.scene.control.ButtonBuilder;
import javafx.scene.control.LabelBuilder;
import javafx.scene.layout.VBoxBuilder;
import javafx.stage.Stage;

public class BuilderApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

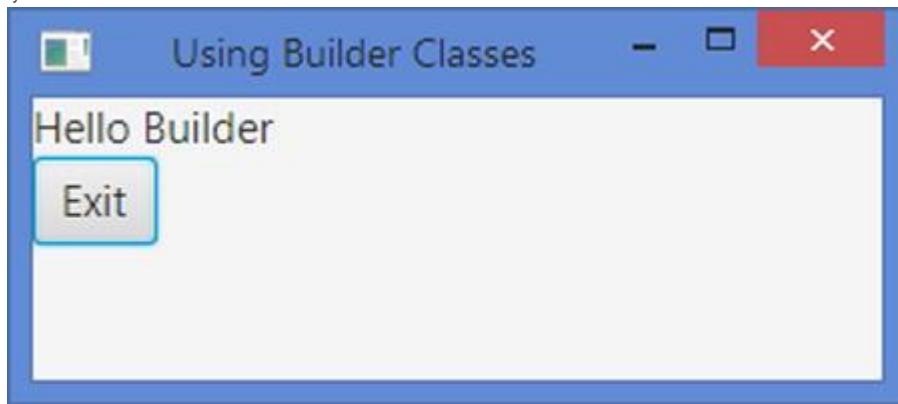
    @Override
    public void start(Stage stage) {
        Scene scene = SceneBuilder.create()
            .width(300)
            .height(100)
            .root(VBoxBuilder.create()
                .children(LabelBuilder.create()
                    .text("Hello Builder")
                    .build(),
                .build(),
            .build(),
        .build(),
    }
}
```

```

        ButtonBuilder.create()
            .text("Exit")
            .onAction(e ->
Platform.exit())
                .build()
            )
        .build()
    )
.build();

stage.setScene(scene);
stage.setTitle("Using Builder Classes");
stage.show();
}
}

```



**Figure 5-5.** A screen whose scene graph is created using builder classes

If you used JavaFX script, which was removed from JavaFX version 2.0, you may be inclined to use builder classes for building scenes. If you have been using Swing/AWT, you may be comfortable using constructors and setters instead. The examples in this book do not use builder classes.

**Table 5-2.** Methods of the Platform Class

Method	Description
void exit()	It terminates a JavaFX application.
boolean isFxApplicationThread()	It returns true if the calling thread is the JavaFX Application Thread. Otherwise, it returns false.
boolean isImplicitExit()	It returns the value of the implicit implicitExit attribute of the application. If it returns true, it means that the application

Method	Description
boolean isSupported(ConditionalFeature feature)	terminate after the last window is closed. Otherwise, you need to call the <code>exit()</code> method of this class to terminate the application.
void runLater(Runnable runnable)	It returns <code>true</code> if the specified conditional feature is supported by the platform. Otherwise, it returns <code>false</code> .
void setImplicitExit(boolean value)	It executes the specified <code>Runnable</code> on the JavaFX Application Thread. The timing of the execution is not specified. The method posts the <code>Runnable</code> to the event queue and returns immediately. If multiple <code>Runnables</code> are posted using this method, they are executed in the order they are submitted to the queue.

## Understanding the *Platform* Class

The `Platform` class in the `javafx.application` package is a utility class used to support platform-related functionalities. It consists of all static methods, which are listed in Table 5-2.

The `runLater()` method is used to submit a `Runnable` task to an event queue, so it is executed on the JavaFX Application Thread. JavaFX allows developers to execute some of the code only on the JavaFX Application Thread. Listing 5-2 creates a task in the `init()` method that is called on the JavaFX Launcher Thread. It uses the `Platform.runLater()` method to submit the task to be executed on the JavaFX Application Thread later.

**Tip** Use the `Platform.runLater()` method to execute a task that is created on a thread other than the JavaFX Application Thread but needs to run on the JavaFX Application Thread.

### **Listing 5-2.** Using the `Platform.runLater()` Method

```
// RunLaterApp.java
package com.jdojo.scene;
```

```

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class RunLaterApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void init() {
        System.out.println("init(): "
+ Thread.currentThread().getName());

        // Create a Runnable task
        Runnable task = () -> System.out.println("Running
the task on the "
+ Thread.currentThread().getName());

        // Submit the task to be run on the JavaFX
Application Thread
        Platform.runLater(task);
    }

    @Override
    public void start(Stage stage) throws Exception {
        stage.setScene(new Scene(new Group(), 400, 100));
        stage.setTitle("Using Platform.runLater() Method");
        stage.show();
    }
}
init(): JavaFX-Launcher
Running the task on the JavaFX Application Thread

```

Some features in a JavaFX implementation are optional (or conditional). They may not be available on all platforms. Using an optional feature on a platform that does not support the feature does not result in an error; the optional feature is simply ignored. Optional features are defined as enum constants in the `ConditionalFeature` enum in the `javafx.application` package, as listed in Table 5-3.

**Table 5-3.** Constants Defined in the `ConditionalFeature` Enum

Enum Constant	Description
EFFECT	Indicates the availability of filter effects, for example, reflection

Enum Constant	Description
	etc.
INPUT_METHOD	Indicates the availability of text input method.
SCENE3D	Indicates the availability of 3D features.
SHAPE_CLIP	Indicates the availability of clipping of a node against an arbitrary shape.
TRANSPARENT_WINDOW	Indicates the availability of the full window transparency.

Suppose your JavaFX application uses 3D GUI on user demand. You can write your logic for enabling 3D features as shown in the following code:

```
import javafx.application.Platform;
import static javafx.application.ConditionalFeature.SCENE3D;
...
if (Platform.isSupported(SCENE3D)) {
    // Enable 3D features
}
else {
    // Notify the user that 3D features are not available
}
```

## Knowing the Host Environment

The `HostServices` class in the `javafx.application` package provides services related to the launching environment (desktop, web browser, or WebStart) hosting the JavaFX application. You cannot create an instance of the `HostServices` class directly.

The `getHostServices()` method of the `Application` class returns an instance of the `HostServices` class. The following is an example of how to get an instance of `HostServices` inside a class that inherits from the `Application` class:

```
HostServices host = getHostServices();
```

The `HostServices` class contains the following methods:

- `String getCodeBase()`
- `String getDocumentBase()`
- `JSONObject getWebContext()`
- `String resolveURI(String base, String relativeURI)`

- void showDocument(String uri)

The getCodeBase() method returns the code base uniform resource identifier (URI) of the application. In a stand-alone mode, it returns the URI of the directory that contains the JAR file used to launch the application. If the application is launched using a class file, it returns an empty string. If the application is launched using a JNLP file, it returns the value for the specified code base parameter in the JNLP file.

The getDocumentBase() method returns the URI of the document base. In a web environment, it returns the URI of the web page that contains the application. If the application is launched using WebStart, it returns the code base parameter specified in the JNLP file. It returns the URI of the current directory for application launched in stand-alone mode.

The getWebContext() method returns a JSObject that allows a JavaFX application to interact with the JavaScript objects in a web browser. If the application is not running in a web page, it returns null. You can use the eval() method of the JSObject to evaluate a JavaScript expression from inside your JavaFX code. The following snippet of code displays an alert box using the window.alert() function. If the application runs in a nonweb environment, it shows a JavaFX modal stage instead:

```
HostServices host = getHostServices();
JSObject js = host.getWebContext();
if (js == null) {
    Stage s = new Stage(StageStyle.UTILITY);
    s.initModality(Modality.WINDOW_MODAL);
    s.setTitle("FX Alert");

    Scene scene = new Scene(new Group(new Label("This is an FX
alert!")));
    s.setScene(scene);
    s.show();
}
else {
    js.eval("window.alert('This is a JavaScript alert!')");
}
```

The resolveURI() method resolves the specified relative URI with respect to the specified base URI and returns the resolved URI.

The showDocument() method opens the specified URI in a new browser window. Depending on the browser preference, it may open the URI in a new tab instead. This method can be used in a stand-alone mode as well as in a web environment. The following snippet of code opens the Yahoo! home page:

```
getHostServices().showDocument("http://www.yahoo.com");
```

The program in Listing 5-3 uses all of the methods of the `HostServices` class. It shows a stage with two buttons and host details. One button opens the Yahoo! home page and another shows an alert box. The output shown on the stage will vary depending on how the application is launched.

### ***Listing 5-3.*** Knowing the Details of the Host Environment for a JavaFX Application

```
// KnowingHostDetailsApp.java
package com.jdojo.scene;

import java.util.HashMap;
import java.util.Map;
import javafx.application.Application;
import javafx.application.HostServices;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javafx.stage.StageStyle;
import netscape.javascript.JSObject;

public class KnowingHostDetailsApp extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String yahooURL = "http://www.yahoo.com";
        Button openURLButton = new Button("Go to Yahoo!");
        openURLButton.setOnAction(e ->
getHostServices().showDocument(yahooURL));

        Button showAlert = new Button("Show Alert");
        showAlert.setOnAction(e -> showAlert());

        VBox root = new VBox();
        // Add buttons and all host related details to the
        // VBox
        root.getChildren().addAll(openURLButton, showAlert);

        Map<String, String> hostdetails = getHostDetails();
        for(Map.Entry<String, String> entry
: hostdetails.entrySet()) {
            String desc = entry.getKey() + ": "
+ entry.getValue();
            root.getChildren().add(new Label(desc));
        }
    }
}
```

```

        }

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Knowing the Host");
        stage.show();
    }

protected Map<String, String> getHostDetails() {
    Map<String, String> map = new HashMap<>();
    HostServices host = this.getHostServices();

    String codeBase = host.getCodeBase();
    map.put("CodeBase", codeBase);

    String documentBase = host.getDocumentBase();
    map.put("DocumentBase", documentBase);

    JSONObject js = host.getWebContext();
    map.put("Environment", js == null?"Non-Web":"Web");

    String splashImageURI
= host.resolveURI(documentBase, "splash.jpg");
    map.put("Splash Image URI", splashImageURI);

    return map;
}

protected void showAlert() {
    HostServices host = getHostServices();
    JSONObject js = host.getWebContext();
    if (js == null) {
        Stage s = new Stage(StageStyle.UTILITY);
        s.initModality(Modality.WINDOW_MODAL);

        Label msgLabel = new Label("This is an FX
alert!");
        Group root = new Group(msgLabel);
        Scene scene = new Scene(root);
        s.setScene(scene);

        s.setTitle("FX Alert");
        s.show();
    }
    else {
        js.eval("window.alert('This is a JavaScript
alert!')");
    }
}
}

```

## Summary

A scene represents the visual contents of a stage. The `Scene` class in the `javafx.scene` package represents a scene in a JavaFX program.

A Scene object is attached to at the most one stage at a time. If an already-attached scene is attached to another stage, it is first detached from the previous stage. A stage can have at the most one scene attached to it at any time.

A scene contains a scene graph that consists of visual nodes. In this sense, a scene acts as a container for a scene graph. A scene graph is a tree data structure whose elements are known as nodes. Nodes in a scene graph form a parent-child hierarchical relationship. A node in a scene graph is an instance of the `javafx.scene.Node` class. A node can be a branch node or a leaf node. A branch node can have children nodes, whereas a leaf node cannot. The first node in a scene graph is called the root node. The root node can have children nodes; however, it never has a parent node.

An instance of the `javafx.scene.Cursor` class represents a mouse cursor. The `Cursor` class contains many constants, for example, `HAND`, `CLOSED_HAND`, `DEFAULT`, `TEXT`, `NONE`, `WAIT`, for standard mouse cursors. You can set a cursor for the scene using the `setCursor()` method of the `Scene` class.

Only one node in a scene can be the focus owner. The read-only `focusOwner` property of the `Scene` class tracks the node that has the focus. If you want a specific node in a scene to be the focus owner, you need to call the `requestFocus()` method of the `Node` class. Each scene may have a focus owner. For example, if you open two windows, you will have two scenes and you may have two focus owners. However, only one of the two focus owners can have the focus at a time. The focus owner of the active window will have the focus. To check if the focus owner node also has the focus, you need to use the `focused` property of the `Node` class.

The `Platform` class in the `javafx.application` package is a utility class used to support platform-related functionalities. It contains methods for terminating the application, checking if the code being executed is executed on the JavaFX Application Thread, and so on.

The `HostServices` class in the `javafx.application` package provides services related to the launching environment (desktop, web browser, or WebStart) hosting the JavaFX application. You cannot create an instance of the `HostServices` class directly.

The `getHostServices()` method of the `Application` class returns an instance of the `HostServices` class.

The next chapter will discuss nodes in detail.

## CHAPTER 23



### Understanding Charts

In this chapter, you will learn

- What a chart is
- What the Chart API is in JavaFX
- How to create different types of charts using the Chart API
- How to style charts with CSS

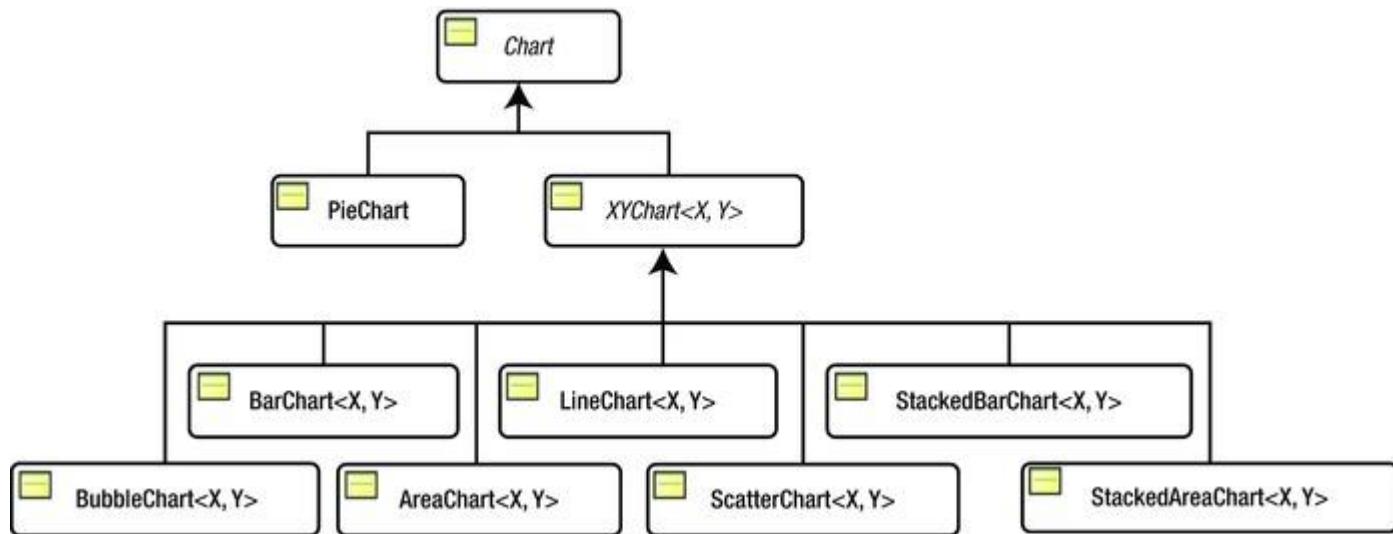
#### What Is a Chart?

A chart is a graphical representation of data. Charts provide an easier way to analyze large volume of data visually. Typically, they are used for reporting purposes. Different types of charts exist. They differ in the way they represent the data. Not all types of charts are suitable for analyzing all types of data. For example, a line chart is suitable for understanding the comparative trend in data whereas a bar chart is suitable for comparing data in different categories.

JavaFX supports charts, which can be integrated in a Java application by writing few lines of code. It contains a comprehensive, extensible Chart API that provides built-in support for several types of charts.

#### Understanding the Chart API

The Chart API consists of a number of predefined classes in the `javafx.scene.chart` package. [Figure 23-1](#) shows a class diagram for classes representing different types of charts.



**Figure 23-1.** A class diagram for the classes representing charts in JavaFX

The abstract `Chart` is the base class for all charts. It inherits the `Node` class. Charts can be added to a scene graph. They can also be styled with CSS as any other nodes. I will discuss styling charts in the sections that discuss specific type of charts. The `Chart` class contains properties and methods common to all type of charts.

JavaFX divides charts into two categories:

- Charts having no-axis
- Charts having an x-axis and a y-axis

The `PieChart` class falls into the first category. It has no axis, and it is used to draw a pie chart.

The `XYChart` class falls into the second category. It is the abstract base class for all charts having two axes. Its subclasses, for example, `LineChart`, `BarChart`, etc., represent specific type of charts.

Every chart in JavaFX has three parts:

- A title
- A legend
- Content (or data)

Different types of charts define their data differently. The `Chart` class contains the following properties that are common to all types of charts:

- `title`
- `titleSide`
- `legend`
- `legendSide`

- legendVisible
- animated

The title property specifies the title for a chart.

The titleSide property specifies the location of the title. By default, the title is placed above the chart content. Its value is one of the constants of the Side enum: TOP (default), RIGHT, BOTTOM, and LEFT.

Typically, a chart uses different types of symbols to represent data in different categories. A legend lists symbols with their descriptions.

The legend property is a Node and it specifies the legend for the chart. By default, a legend is placed below the chart content.

The legendSide property specifies the location of the legend, which is one of the constants of the Side enum: TOP, RIGHT, BOTTOM (default), and LEFT. The legendVisible property specifies whether the legend is visible. By default, it is visible.

The animated property specifies whether the change in the content of the chart is shown with some type of animation. By default, it is true.

## Styling Charts with CSS

You can style all types of charts. The Chart class defines properties common to all types of charts. The default CSS style-class name for a chart is *chart*. You can specify the legendSide, legendVisible, and titleSide properties for all charts in a CSS as shown:

```
.chart {  
    -fx-legend-side: top;  
    -fx-legend-visible: true;  
    -fx-title-side: bottom;  
}
```

Every chart defines two substructures:

- chart-title
- chart-content

The chart-title is a Label and the chart-content is a Pane. The following styles sets the background color for all charts to yellow and the title font to Arial 16px bold.

```
.chart-content {  
    -fx-background-color: yellow;  
}  
  
.chart-title {  
    -fx-font-family: "Aeial";  
    -fx-font-size: 16px;  
    -fx-font-weight: bold;  
}
```

The default style-class name for legends is *chart-legend*. The following style sets the legend background color to light gray.

```
.chart-legend {
    -fx-background-color: lightgray;
}
```

Every legend has two substructures:

- chart-legend-item
- chart-legend-item-symbol

The *chart-legend-item* is a *Label*, and it represents the text in the legend. The *chart-legend-item-symbol* is a *Node*, and it represents the symbol next to the label, which is a circle by default. The following style sets the font size for the labels in legends to 10px and the legend symbols to an arrow.

```
.chart-legend-item {
    -fx-font-size: 16px;
}

.chart-legend-item-symbol {
    -fx-shape: "M0 -3.5 v7 1 4 -3.5z";
}
```

**Note** Many examples in this chapter use external resources such as CSS files. You will need to include the *resources* directory and its contents in *CLASSPATH* for all programs to work correctly. The *resources* directory is located under the *src* directory in the source code bundle that you can download from [www.apress.com/source-code](http://www.apress.com/source-code).

## Data Used in Chart Examples

I will discuss different types of charts shortly. Charts will use data from **Table 23-1**, which has the actual and estimated population of some countries in the world. The data has been taken from the report published by the United Nations at <http://www.un.org>. The population values have been rounded.

**Table 23-1.** Current and Estimated Populations (in Millions) of Some Countries in the World

	1950	2000	2050	2100	2150	2200	2250	2300
<b>China</b>	555	1275	1395	1182	1149	1201	1247	1283
<b>India</b>	358	1017	1531	1458	1308	1304	1342	1371
<b>Brazil</b>	54	172	233	212	202	208	216	223
<b>UK</b>	50	59	66	64	66	69	71	73
<b>USA</b>	158	285	409	437	453	470	483	493

## Understanding the PieChart

A pie chart consists of a circle divided into sectors of different central angles. Typically, a pie is circular. The sectors are also known as *pie pieces* or *pie slices*. Each sector in the circle represents a quantity of some kind. The central angle of the area of a sector is proportional to the quantity it represents. Figure 23-2 shows a pie chart that displays the population of five countries in the year 2000.

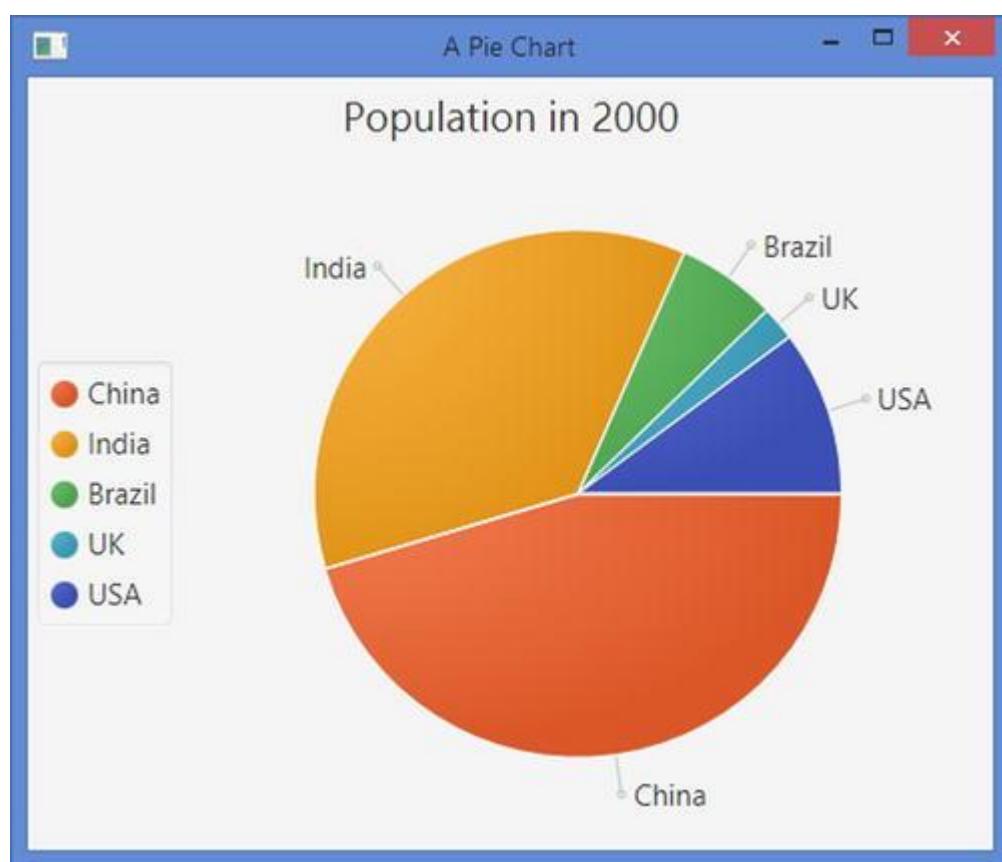


Figure 23-2. A pie chart showing population of five countries in 2000

An instance of the `PieChart` class represents a pie chart. The class contains two constructors:

- `PieChart()`
- `PieChart(ObservableList<PieChart.Data> data)`

The no-args constructor creates a pie chart with no content. You can add the content later using its `data` property. The second constructor creates a pie chart with the specified data as its content.

```
// Create an empty pie chart
PieChart chart = new PieChart();
```

A slice in a pie chart is specified as an instance of the `PieChart.Data` class. A slice has a name (or a label) and a pie value represented by the `name` and `pieValue` properties of the `PieChart.Data` class, respectively. The following statement creates a slice for a pie chart. The slice name is “China,” and the pie value is 1275.

```
PieChar.Data chinaSlice = new PieChart.Data("China", 1275);
```

The content of a pie chart (all slices) is specified in an `ObservableList<PieChart.Data>`. The following snippet of code creates an `ObservableList<PieChart.Data>` and adds three pie slices to it.

```
ObservableList<PieChart.Data> chartData
= FXCollections.observableArrayList();
chartData.add(new PieChart.Data("China", 1275));
chartData.add(new PieChart.Data("India", 1017));
chartData.add(new PieChart.Data("Brazil", 172));
```

Now, you can use the second constructor to create a pie chart by specifying the chart content:

```
// Create a pie chart with content
PieChart charts = new PieChart(chartData);
```

You will use populations of different countries in 2050 as the data for all our pie charts. [Listing 23-1](#) contains a utility class. Its `getChartData()` method returns an `ObservableList` of `PieChart.Data` to be used as data for a pie chart. You will use this class in our examples in this section.

### **[Listing 23-1](#). A Utility Class to Generate Data for Pie Charts**

```
// PieChartUtil.java
package com.jdojo.chart;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.chart.PieChart;
```

```

public class PieChartUtil {
    public static ObservableList<PieChart.Data> getChartData()
    {
        ObservableList<PieChart.Data> data = FXCollections.
        observableArrayList();
        data.add(new PieChart.Data("China", 1275));
        data.add(new PieChart.Data("India", 1017));
        data.add(new PieChart.Data("Brazil", 172));
        data.add(new PieChart.Data("UK", 59));
        data.add(new PieChart.Data("USA", 285));
        return data;
    }
}

```

The `PieChart` class contains several properties:

- `data`
- `startAngle`
- `clockwise`
- `labelsVisible`
- `labelLineLength`

The `data` property specifies the content for the chart in an `ObservableList<PieChart.Data>`.

The `startAngle` property specifies the angle in degrees to start the first pie slice. By default, it is zero degrees, which corresponds to three o'clock position. A positive `startAngle` is measured anticlockwise. For example, a 90-degree `startAngle` will start at the 12 o'clock position.

The `clockwise` property specifies whether the slices are placed clockwise starting at the `startAngle`. By default, it is true.

The `labelsVisible` property specifies whether the labels for slices are visible. Labels for slices are displayed close to the slice and they are placed outside the slices. The label for a slice is specified using the `name` property of the `PieChart.Data` class. In [Figure 23-2](#), "China," India," Brazil,, etc., are labels for slices.

Labels and slices are connected through straight lines.

The `labelLineLength` property specifies the length of those lines. Its default value is 20.0 pixels.

The program in [Listing 23-2](#) uses a pie chart to display the population for five countries in 2000. The program creates an empty pie chart and sets its title. The legend is placed on the left side. Later, it sets the data for the chart. The data is generated in the `getChartData()` method, which returns an `ObservableList<PieChart.Data>` containing the name of the countries as the labels for pie slices and their populations as pie values. The program displays a window as shown in [Figure 23-2](#).

## ***Listing 23-2.*** Using the PieChart Class to Create a Pie Chart

```
// PieChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Side;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class PieChartTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        PieChart chart = new PieChart();
        chart.setTitle("Population in 2000");

        // Place the legend on the left side
        chart.setLegendSide(Side.LEFT);

        // Set the data for the chart
        ObservableList<PieChart.Data> chartData
= PieChartUtil.getChartData();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.setTitle("A Pie Chart");
        stage.show();
    }
}
```

### Customizing Pie Slices

Each pie slice data is represented by a `Node`. The reference to the `Node` can be obtained using the `getNode()` method of the `PieChart.Data` class. The `Node` is created when the slices are added to the pie chart. Therefore, you must call the `getNode()` method on the `PieChart.Data` representing the slice after adding it to the chart. Otherwise, it returns `null`. The program in [Listing 23-3](#) customizes all pie slices of a pie chart to add a tooltip to them. The tooltip shows the slice name, pie value, and percent pie value. The `addSliceTooltip()` method contains the logic to accessing the

slice Nodes and adding the tooltips. You can customize pie slices to animate them, let the user drag them out from the pie using the mouse, etc.

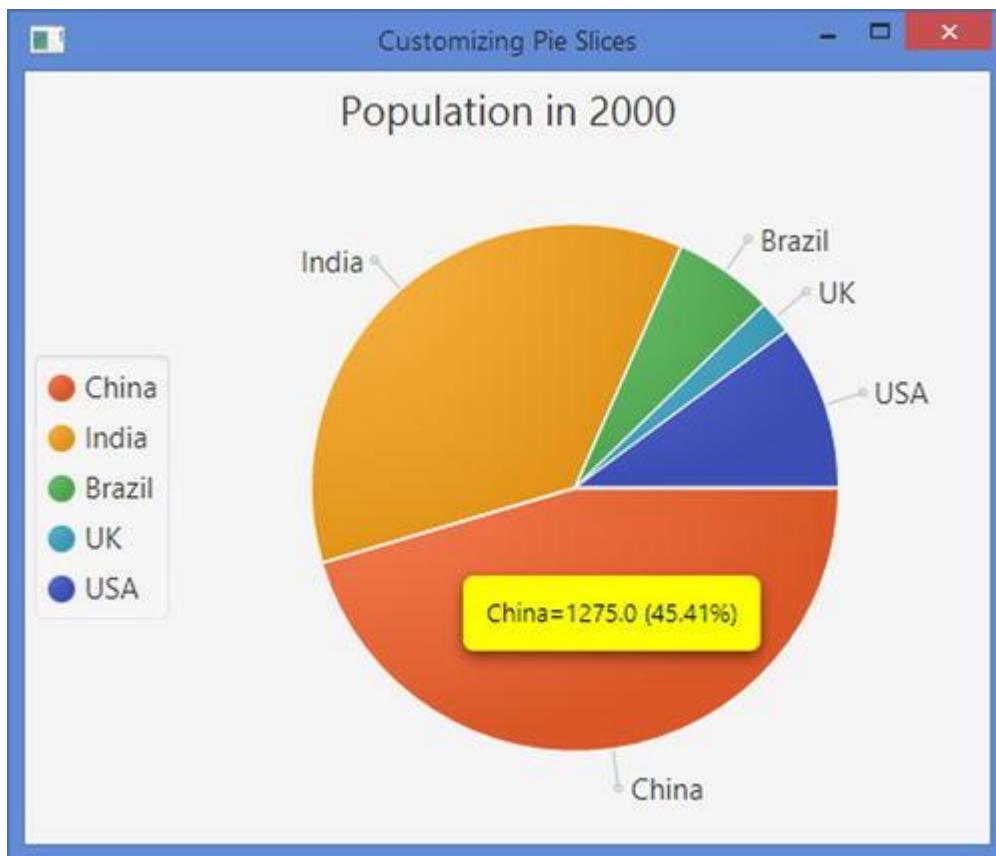


Figure 23-3. A pie slice showing a tooltip with its pie value and percent of the total pie

### **Listing 23-3.** Adding Tooltips to Pie Slices

```
// PieSliceTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Side;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class PieSliceTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
```

```

public void start(Stage stage) {
    PieChart chart = new PieChart();
    chart.setTitle("Population in 2000");

    // Place the legend on the left side
    chart.setLegendSide(Side.LEFT);

    // Set the data for the chart
    ObservableList<PieChart.Data> chartData
= PieChartUtil.getChartData();
    chart.setData(chartData);

    // Add a Tooltip to all pie slices
    this.addSliceTooltip(chart);

    StackPane root = new StackPane(chart);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Customizing Pie Slices");
    stage.show();
}

private void addSliceTooltip(PieChart chart) {
    // Compute the total pie value
    double totalPieValue = 0.0;
    for (PieChart.Data d : chart.getData()) {
        totalPieValue += d.getPieValue();
    }

    // Add a tooltip to all pie slices
    for (PieChart.Data d : chart.getData()) {
        Node sliceNode = d.getNode();
        double pieValue = d.getPieValue();
        double percentPieValue = (pieValue
/ totalPieValue) * 100;

        // Create and install a Tooltip for the slice
        String msg = d.getName() + "=" + pieValue +
                    " (" + String.format("%.2f",
percentPieValue) + "%)";
        Tooltip tt = new Tooltip(msg);
        tt.setStyle("-fx-background-color: yellow;" +
                    "-fx-text-fill: black;");
        Tooltip.install(sliceNode, tt);
    }
}
}

```

## Styling the PieChart with CSS

All properties, except the `data` property, defined in the `PieChart` class, can be styled using CSS as shown below.

```
.chart {
    -fx-clockwise: false;
```

```

        -fx-pie-label-visible: true;
        -fx-label-line-length: 10;
        -fx-start-angle: 90;
    }
}

```

Four style classes are added to each pie slice added to a pie chart:

- chart-pie
- data<i>
- default-color<j>
- negative

The <i> in the style-class name data<i> is the slice index. The first slice has the class data0, the second data1, the third data2, etc.

The <j> in the style-class name default-color<j> is the color index of the series. In a pie chart, you can think of each slice as a series. The default CSS (Modena.css) defines eight series colors. If your pie slice has more than eight slices, the slice color will be repeated. The concept of series in a chart will be more evident when I discuss two-axis charts in the next section.

The *negative* style-class is added only when the data for the slice is negative.

Define a style for *chart-pie* style-class-name if you want that style to apply to all pie slices. The following style will set a white border with 2px of background insets for all pie slices. It will show a wider gap between two slices as you have set 2px insets.

```

.chart-pie {
    -fx-border-color: white;
    -fx-background-insets: 2;
}

```

You can define colors for pie slices using the following styles. It defines colors for only five slices. Slices beyond the sixth one will use default colors.

```

.chart-pie.default-color0 {-fx-pie-color: red;}
.chart-pie.default-color1 {-fx-pie-color: green;}
.chart-pie.default-color2 {-fx-pie-color: blue;}
.chart-pie.default-color3 {-fx-pie-color: yellow;}
.chart-pie.default-color4 {-fx-pie-color: tan;}

```

## Using More Than Eight Series Colors

It is quite possible that you will have more than eight series (slices in a pie chart) in a chart and you do not want to repeat the colors for the series. The technique is discussed for a pie chart. However, it can be used for a 2-axis chart as well.

Suppose you want to use a pie that will display populations of ten countries. If you use the code for this pie chart, the colors for the ninth

and tenth slices will be the same as the colors for the first and second slices, respectively. First, you need to define the colors for the ninth and tenth slices as shown in [Listing 23-4](#).

### **Listing 23-4.** Additional Series Colors

```
/* additional_series_colors.css */
.chart-pie.default-color8 {
    -fx-pie-color: gold;
}

.chart-pie.default-color9 {
    -fx-pie-color: khaki;
}
```

The pie slices and the legend symbols will be assigned style-class names such as default-color0, default-color2... default-color7. You need to identify the nodes for the slices and legend symbols associated with data items with index greater than 7 and replace their default-color <j>style-class name with the new ones. For example, for the ninth and tenth slices, the style-class names are default-color0 and default-color1 as the color series number is assigned as (dataIndex % 8). You will replace them with default-color9 and default-color10.

The program in [Listing 23-5](#) shows how to change the colors for the slices and legend symbols. It adds ten slices to a pie chart.

The `setSeriesColorStyles()` method replaces the style-class names for the slice nodes for the ninth and tenth slices and for their associated legend symbols. [Figure 23-4](#) shows the pie chart. Notice the colors for “Germany” and “Indonesia” are gold and khaki as set in the CSS. Comment the last statement in the `start()` method, which is a call to the `setSeriesColorStyles()` and you will find that the colors for “Germany” and “Indonesia” will be the same as the colors for “China” and “India.”

### **Listing 23-5.** A Pie Chart Using Color Series up to Index 10

```
// PieChartExtraColor.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Side;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
```

```

public class PieChartExtraColor extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        PieChart chart = new PieChart();
        chart.setTitle("Population in 2000");

        // Place the legend on the left side
        chart.setLegendSide(Side.LEFT);

        // Set the data for the chart
        ObservableList<PieChart.Data> chartData
= PieChartUtil.getChartData();
        this.addData(chartData);
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        scene.getStylesheets()
            .add("resources/css/additional_series_colors.css"
);

        stage.setScene(scene);
        stage.setTitle("A Pie Chart with over 8 Slices");
        stage.show();

        // Override the default series color style-class-name
for slices over 8.
        // Works only when you set it after the scene is
visible
        this.setSeriesColorStyles(chart);
    }

    private void addData(ObservableList<PieChart.Data> data) {
        data.add(new PieChart.Data("Bangladesh", 138));
        data.add(new PieChart.Data("Egypt", 68));
        data.add(new PieChart.Data("France", 59));
        data.add(new PieChart.Data("Germany", 82));
        data.add(new PieChart.Data("Indonesia", 212));
    }

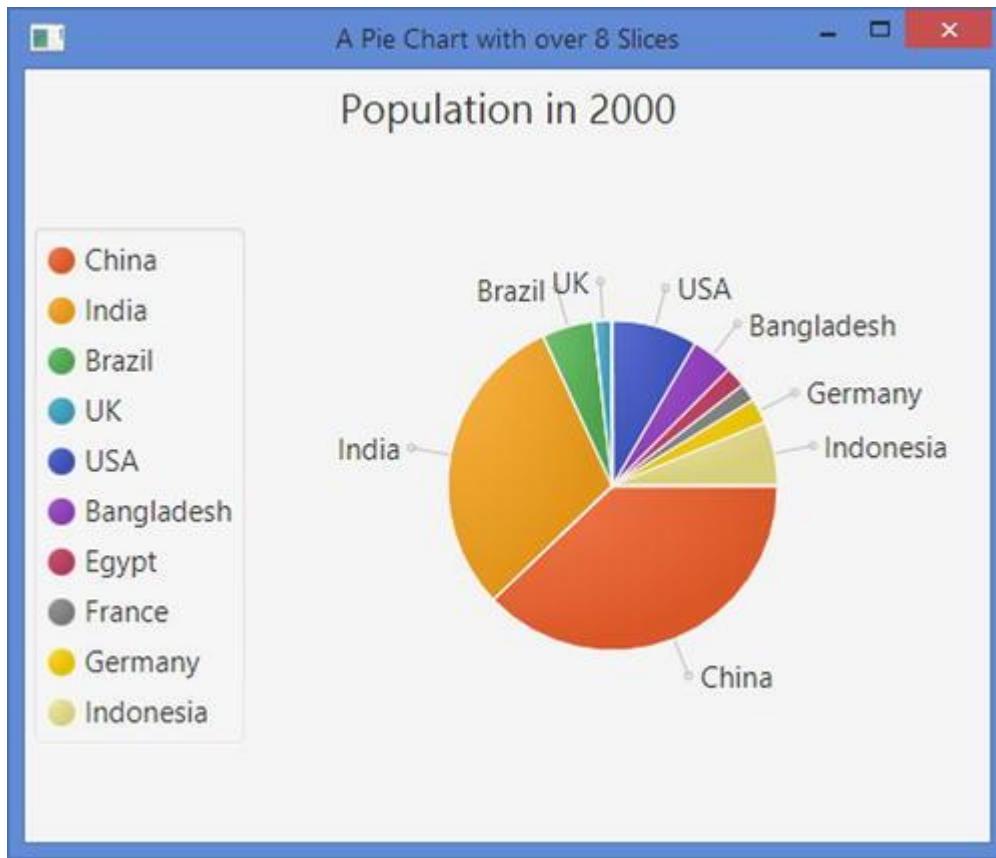
    private void setSeriesColorStyles(PieChart chart) {
        ObservableList<PieChart.Data> chartData
= chart.getData();
        int size = chartData.size();
        for (int i = 8; i < size; i++) {
            String removedStyle = "default-color" + (i
% 8);
            String addedStyle = "default-color" + (i
% size);

            // Reset the pie slice colors
    }
}

```

```
Node node = chartData.get(i).getNode();
node.getStyleClass().remove(removedStyle);
node.getStyleClass().add(addedStyle);

// Reset the legend colors
String styleClass = ".pie-legend-symbol.data"
+ i +
    ".default-color" + (i % 8);
Node legendNode = chart.lookup(styleClass);
if (legendNode != null) {
    legendNode.getStyleClass().remove(removed
Style);
    legendNode.getStyleClass().add(addedStyle
);
}
}
}
```



**Figure 23-4.** A pie chart using over 8 slice colors

## Using Background Images for Pie Slices

You can also use a background image in a pie slice. The following style defines the background image for the first pie slice.

```
.chart-pie.data0 {
    -fx-background-image: url("china_flag.jpg");
}
```

**Listing 23-6** contains the content of a CSS file named *pie\_slice.css*. It defines styles that specify the background images used for pie slices, the preferred size of the legend symbols, and the length of the line joining the pie slices and their labels.

### ***Listing 23-6.*** A CSS for Customizing Pie Slices

```
// pie_slice.css
/* Set a background image for pie slices */
.chart-pie.data0 {-fx-background-image: url("china_flag.jpg");}
.chart-pie.data1 {-fx-background-image: url("india_flag.jpg");}
.chart-pie.data2 {-fx-background-image: url("brazil_flag.jpg");}
.chart-pie.data3 {-fx-background-image: url("uk_flag.jpg");}
.chart-pie.data4 {-fx-background-image: url("usa_flag.jpg");}

/* Set the preferred size for legend symbols */
.chart-legend-item-symbol {
    -fx-pref-width: 100;
    -fx-pref-height: 30;
}

.chart {
    -fx-label-line-length: 10;
}
```

The program in **Listing 23-7** creates a pie chart. It uses the same data as you have been using in our previous examples. The difference is that it sets a CSS defined in a *pie\_slice.css* file.

```
// Set a CSS for the scene
scene.getStylesheets().addAll("resources/css/pie_slice.css");
```

The resulting window is shown in **Figure 23-5**. Notice that slices and legend symbols show the flags of the countries. It is important to keep in mind that you have matched the index of the chart data and the index in the CSS file to match countries and their flags.

**Tip** It is also possible to style the shape of the line joining the pie slices and their labels, labels for the pie slices, and the legend symbols in a pie chart.

### ***Listing 23-7.*** Using Pie Slices with a Background Image

```
// PieChartCustomSlice.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Side;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
```

```
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class PieChartCustomSlice extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

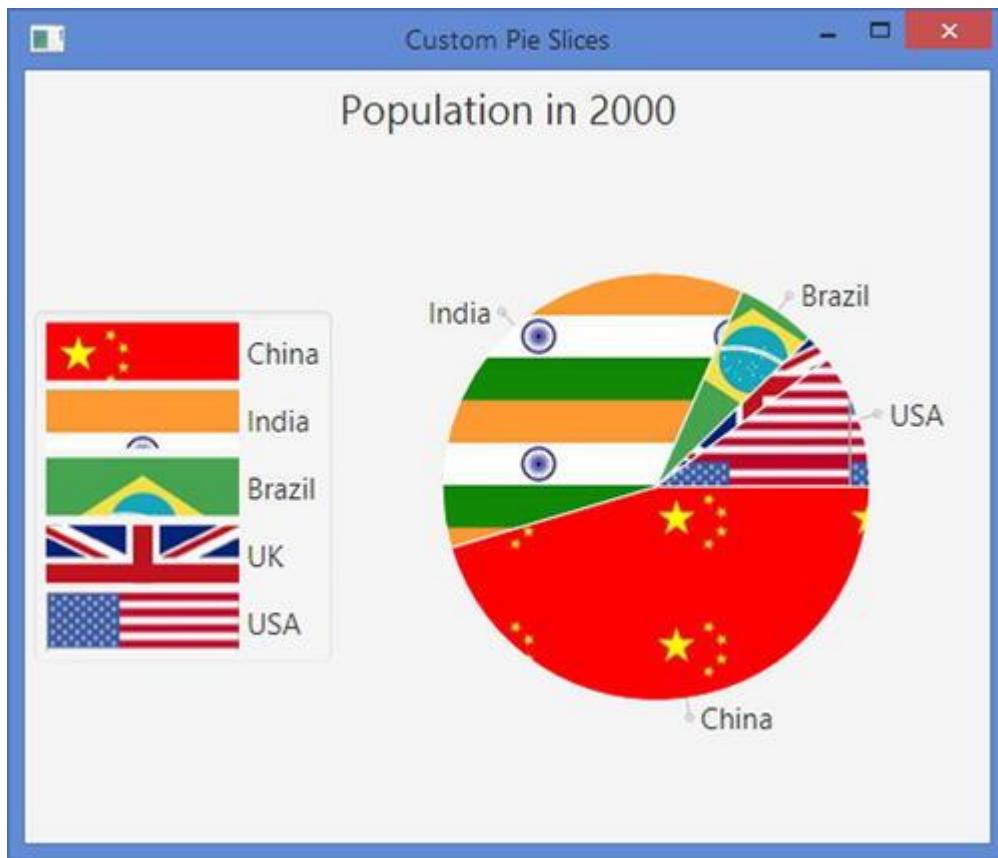
    @Override
    public void start(Stage stage) {
        PieChart chart = new PieChart();
        chart.setTitle("Population in 2000");

        // Place the legend on the left side
        chart.setLegendSide(Side.LEFT);

        // Set the data for the chart
        ObservableList<PieChart.Data> chartData
= PieChartUtil.getChartData();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);

        // Set a CSS for the scene
        scene.getStylesheets().addAll("resources/css/pie_slic
e.css");
        stage.setScene(scene);
        stage.setTitle("Custom Pie Slices");
        stage.show();
    }
}
```



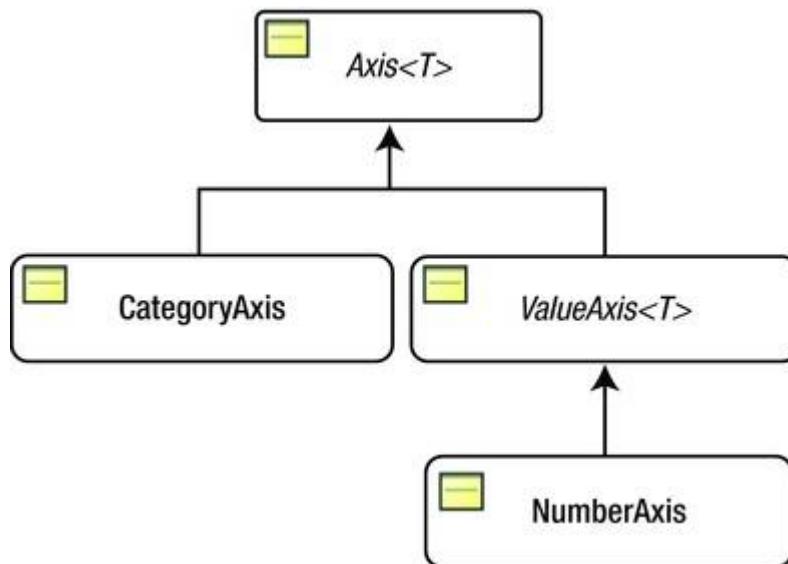
**Figure 23-5.** A pie chart using a background image for its slices

## Understanding the XYChart

An instance of a concrete subclass of the abstract `XYChart<X, Y>` class defines a two-axis chart. The generic type parameters `X` and `Y` are the data type of values plotted along x-axis and y-axis, respectively.

### Representing Axes in an XYChart

An instance of a concrete subclass of the abstract `Axis<T>` class defines an axis in the `XYChart`. **Figure 23-6** shows a class diagram for the classes representing axes.



**Figure 23-6.** A class diagram for classes representing axes in an *XYChart*

The abstract `Axis<T>` class is the base class for all classes representing axes. The generic parameter `T` is the type of the values plotted along the axis, for example, `String`, `Number`, etc. An axis displays ticks and tick labels. The `Axis<T>` class contains properties to customize the ticks and tick labels. An axis can have a label, which is specified in the `label` property.

The concrete subclasses `CategoryAxis` and `NumberAxis` are used for plotting `String` and `Number` data values along an axis, respectively. They contain properties specific to the data values. For example, `NumberAxis` inherits `ValueAxis<T>`'s `lowerBound` and `upperBound` properties, which specify the lower and upper bounds of the data plotted on the axis. By default, the range of the data on an axis is automatically determined based on the data. You can turn off this feature by setting the `autoRanging` property in the `Axis<T>` class to false. The following snippet of code creates an instance of the `CategoryAxis` and `NumberAxis` and sets their labels.

```

CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Country");
NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population (in millions)");
  
```

**Tip** Use a `CategoryAxis` to plot `String` values along an axis, and use a `NumberAxis` to plot numeric values along an axis.

## Adding Data to an *XYChart*

Data in an `XYChart` represents points in the 2D plane defined by the x-axis and y-axis. A point in a 2D plane is specified using the x and y

coordinates, which are values along the x-axis and y-axis, respectively. The data in an `XYChart` is specified as an `ObservableList` of named series. A series consists of multiple data items, which are points in the 2D plane. How the points are rendered depends on the chart type. For example, a scatter chart shows a symbol for a point whereas a bar chart shows a bar for a point.

An instance of the nested static `XYChart.Data<X, Y>` class represents a data item in a series. The class defines the following properties:

- `XValue`
- `YValue`
- `extraValue`
- `node`
- The `XValue` and `YValue` are the values for the data item along the x-axis and y-axis, respectively. Their data types need to match the data type of the x-axis and y-axis for the chart. The `extraValue` is an `Object`, which can be used to store any additional information for the data item. Its use depends of the chart type. If the chart does not use this value, you can use it for any other purpose: for example, to store the tooltip value for the data item. The `node` specifies the node to be rendered for the data item in the chart. By default, the chart will create a suitable node depending on the chart type.

Suppose both axes of an `XYChart` plot numeric values. The following snippet of code creates some data items for the chart. The data items are the population of China in 1950, 2000, and 2050.

```
XYChart.Data<Number, Number> data1 = new XYChart.Data<>(1950,
555);
XYChart.Data<Number, Number> data2 = new XYChart.Data<>(2000,
1275);
XYChart.Data<Number, Number> data3 = new XYChart.Data<>(2050,
1395);
```

An instance of the nested static `XYChart.Series<X, Y>` class represents a series of data items. The class defines the following properties:

- `name`
- `data`
- `chart`
- `node`

The name is the name of the series. The data is an ObservableList of XYChart.Data<X, Y>. The chart is a read-only reference to the chart to which the series belong. The node is a Node to display for this series. A default node is automatically created based on the chart type. The following snippet of code creates a series, sets its name, and adds data items to it.

```
XYChart.Series<Number, Number> seriesChina = new
XYChart.Series<>();
seriesChina.setName("China");
seriesChina.getData().addAll(data1, data2, data3);
```

The data property of the XYChart class represents the data for the chart. It is an ObservableList of XYChart.Series class. The following snippet of code creates and adds the data for an XYChart chart assuming the data series seriesIndia and seriesUSA exists.

```
XYChart<Number, Number> chart = ...
chart.getData().addAll(seriesChina, seriesIndia, seriesUSA);
```

How the data items for a series are displayed depends on the specific chart type. Every chart type has a way to distinguish one series from another.

You will reuse the same series of data items representing the population of some counties in some years several times. [Listing 23-8](#) has code for a utility class. The class consists of two static methods that generate and return XYChart data.

The getCountrySeries() method returns the list of series that plots the years along the x-axis and the corresponding populations along the y-axis. The getYearSeries() method returns a list of series that plots the countries along the x-axis and the corresponding populations along the y-axis. You will be calling these methods to get data for our XYCharts in subsequent sections.

### **[Listing 23-8.](#) A Utility Class to Generate Data Used in XYCharts**

```
// XYChartDataUtil.java
package com.jdojo.chart;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.chart.XYChart;

@SuppressWarnings("unchecked")
public class XYChartDataUtil {
    public static ObservableList<XYChart.Series<Number,
Number>> getCountrySeries() {
        XYChart.Series<Number, Number> seriesChina = new
XYChart.Series<>();
        seriesChina.setName("China");
```

```

seriesChina.getData().addAll(new XYChart.Data<>(1950,
555),
                               new XYChart.Data<>(2000,
1275),
                               new XYChart.Data<>(2050,
1395),
                               new XYChart.Data<>(2100,
1182),
                               new XYChart.Data<>(2150,
1149));

XYChart.Series<Number, Number> seriesIndia = new
XYChart.Series<>();
seriesIndia.setName("India");
seriesIndia.getData().addAll(new XYChart.Data<>(1950,
358),
                               new XYChart.Data<>(2000,
1017),
                               new XYChart.Data<>(2050,
1531),
                               new XYChart.Data<>(2100,
1458),
                               new XYChart.Data<>(2150,
1308));

XYChart.Series<Number, Number> seriesUSA = new
XYChart.Series<>();
seriesUSA.setName("USA");
seriesUSA.getData().addAll(new XYChart.Data<>(1950,
158),
                               new XYChart.Data<>(2000, 285),
                               new XYChart.Data<>(2050, 409),
                               new XYChart.Data<>(2100, 437),
                               new XYChart.Data<>(2150, 453));

ObservableList<XYChart.Series<Number, Number>> data =
FXCollections.<XYChart.Series<Number,
Number>>observableArrayList();
data.addAll(seriesChina, seriesIndia, seriesUSA);
return data;
}

public static ObservableList<XYChart.Series<String,
Number>> getYearSeries() {
    XYChart.Series<String, Number> series1950 = new
XYChart.Series<>();
    series1950.setName("1950");
    series1950.getData().addAll(new
XYChart.Data<>("China", 555),
                               new XYChart.Data<>("India",
358),
                               new XYChart.Data<>("Brazil",
54),
                               new XYChart.Data<>("UK", 50),
                               new XYChart.Data<>("Australia", 10));
}

```

```

        new XYChart.Data<>("USA",
158));
    XYChart.Series<String, Number> series2000 = new
XYChart.Series<>();
    series2000.setName("2000");
    series2000.getData().addAll(new
XYChart.Data<>("China", 1275),
new
XYChart.Data<>("India", 1017),
172),
new XYChart.Data<>("Brazil",
new XYChart.Data<>("UK", 59),
new XYChart.Data<>("USA",
285));
    XYChart.Series<String, Number> series2050 = new
XYChart.Series<>();
    series2050.setName("2050");
    series2050.getData().addAll(new
XYChart.Data<>("China", 1395),
new
XYChart.Data<>("India", 1531),
233),
new XYChart.Data<>("Brazil",
new XYChart.Data<>("UK", 66),
new XYChart.Data<>("USA",
409));
    ObservableList<XYChart.Series<String, Number>> data =
FXCollections.<XYChart.Series<String,
Number>>observableArrayList();
    data.addAll(series1950, series2000, series2050);
    return data;
}
}

```

## **Understating the BarChart**

A bar chart renders the data items as horizontal or vertical rectangular bars. The lengths of the bars are proportional to the value of the data items.

An instance of the `BarChart` class represents a bar chart. In a bar chart, one axis must be a `CategoryAxis` and the other a `ValueAxis/NumberAxis`. The bars are drawn vertically or horizontally, depending on whether the `CategoryAxis` is the x-axis or the y-axis.

The `BarChart` contain two properties to control the distance between two bars in a category and the distance between two categories:

- `barGap`

- categoryGap

The default value is 4 pixels for the barGap and 10 pixels for the categoryGap.

The BarChart class contains three constructors to create bar charts by specifying axes, data, and gap between two categories.

- BarChart(Axis<X> xAxis, Axis<Y> yAxis)
- BarChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data)
- BarChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data, double categoryGap)

Notice that you must specify at least the axes when you create a bar chart. The following snippet of code creates two axes and a bar chart with those axes.

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Country");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population (in millions)");

// Create a bar chart
BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);
```

The bars in the chart will appear vertically as the category axis is added as the x-axis. You can populate the chart with data using its setData() method.

```
// Set the data for the chart
chart.setData(XYChartDataUtil.getYearSeries());
```

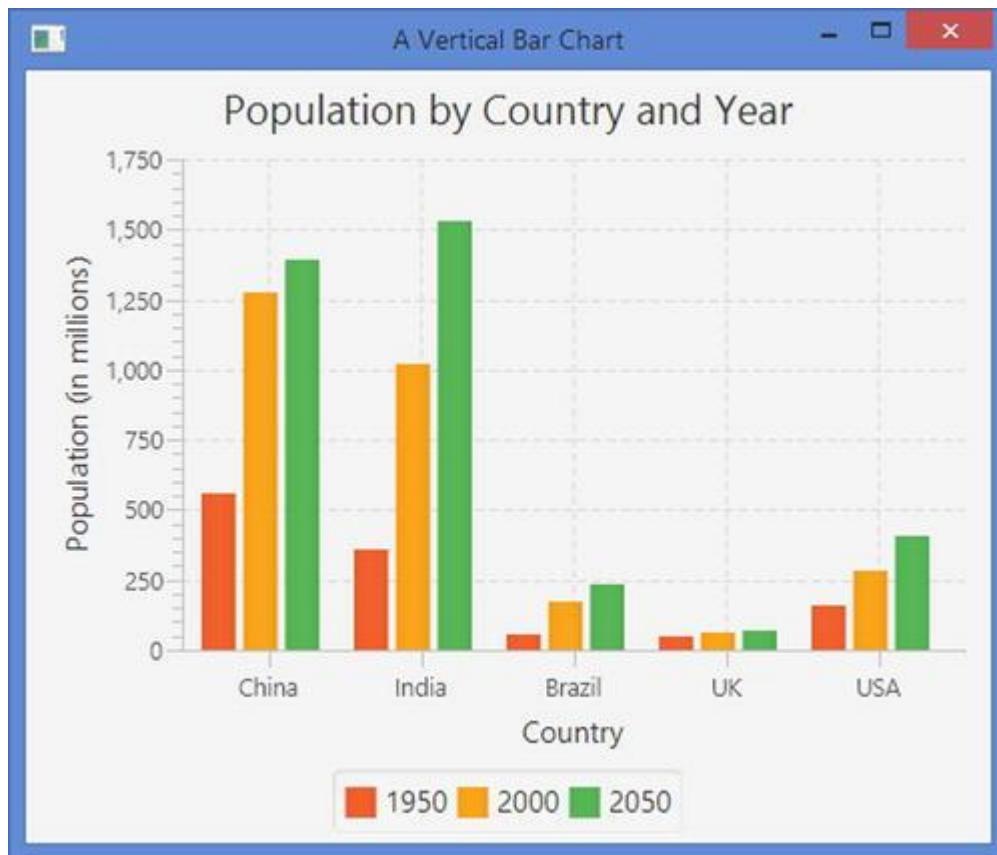
The program in [Listing 23-9](#) shows how to create and populate a vertical bar chart as shown in [Figure 23-7](#).

### **Listing 23-9.** Creating a Vertical Bar Chart

```
// VerticalBarChart.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
```

```
public class VerticalBarChart extends Application {  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
  
    @Override  
    public void start(Stage stage) {  
        CategoryAxis xAxis = new CategoryAxis();  
        xAxis.setLabel("Country");  
  
        NumberAxis yAxis = new NumberAxis();  
        yAxis.setLabel("Population (in millions)");  
  
        BarChart<String, Number> chart = new  
        BarChart<>(xAxis, yAxis);  
        chart.setTitle("Population by Country and Year");  
  
        // Set the data for the chart  
        ObservableList<XYChart.Series<String, Number>>  
        chartData = XYChartDataUtil.getYear  
        Series();  
        chart.setData(chartData);  
  
        StackPane root = new StackPane(chart);  
        Scene scene = new Scene(root);  
        stage.setScene(scene);  
        stage.setTitle("A Vertical Bar Chart");  
        stage.show();  
    }  
}
```



**Figure 23-7.** A vertical bar chart

The program in [Listing 23-10](#) shows how to create and populate a horizontal bar chart as shown in [Figure 23-8](#). The program needs to supply data to the chart in an `ObservableList<XYChart.Series<Number, String>`. The `getYearSeries()` method in the `XYChartDataUtil` class returns `XYChart.Series<String, Number>`. The `getChartData()` method in the program converts the series data from `<String, Number>` to `<Number, String>` format as needed to create a horizontal bar chart.

### **[Listing 23-10.](#) Creating a Horizontal Bar Chart**

```
// HorizontalBarChart.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
```

```

import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HorizontalBarChart extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Population (in millions)");

        CategoryAxis yAxis = new CategoryAxis();
        yAxis.setLabel("Country");

        // Use a category axis as the y-axis for a horizontal
        bar chart
        BarChart<Number, String> chart = new
        BarChart<>(xAxis, yAxis);
        chart.setTitle("Population by Country and Year");

        // Set the data for the chart
        ObservableList<XYChart.Series<Number, String>>
chartData =
            this.getChartData(XYChartDataUtil.getYear
        Series());
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Horizontal Bar Chart");
        stage.show();
    }

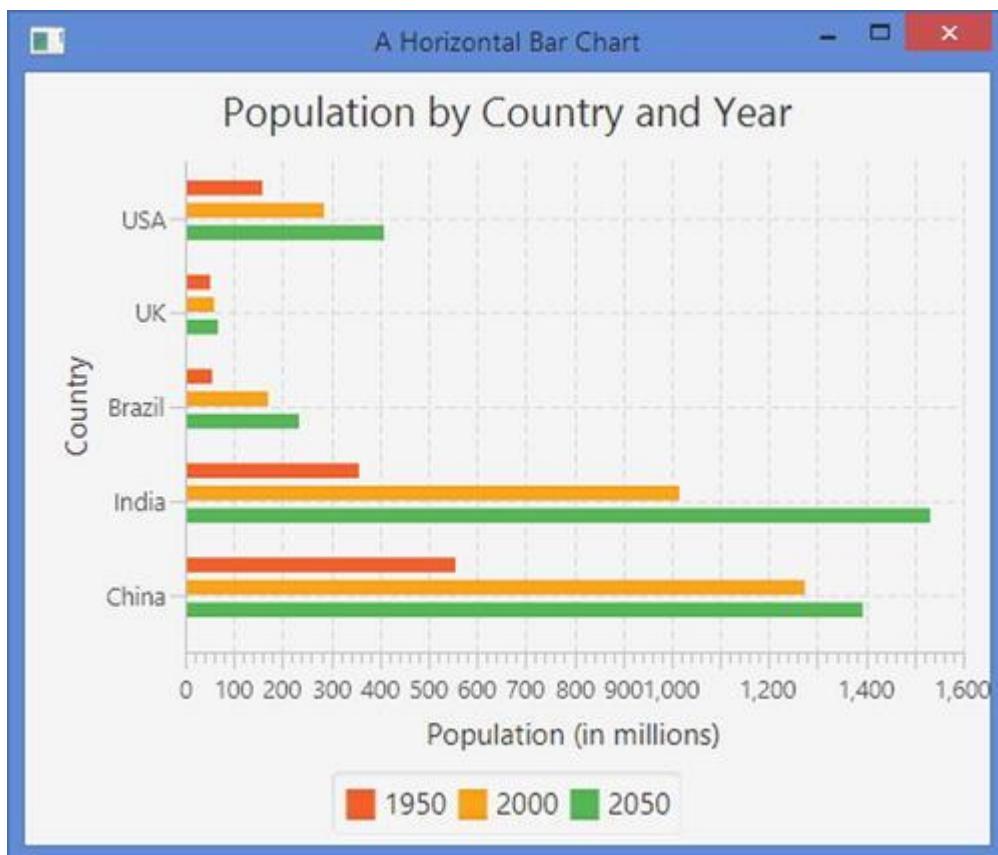
    private ObservableList<XYChart.Series<Number, String>>
getChartData(
        ObservableList<XYChart.Series<String, Number>>
oldData) {
        ObservableList<XYChart.Series<String, Number>>
newData =
            FXCollections.observableArrayList();

        // Read (String, Number) from old data and convert it
        into
        // (Number, String) in new data
        for(XYChart.Series<String, Number> oldSeries:
oldData) {
            XYChart.Series<Number, String> newSeries = new
            XYChart.Series<>();
            newSeries.setName(oldSeries.getName());

            for(XYChart.Data<String, Number> oldItem:
oldSeries.getData()) {

```

```
        XYChart.Data<Number, String> newItem =  
            new  
XYChart.Data<>(oldItem.getYValue(),  
                           oldItem.g  
etXValue());  
        newSeries.getData().add(newItem);  
    }  
    newData.add(newSeries);  
}  
return newData;  
}  
}
```



**Figure 23-8.** A horizontal bar chart

**Tip** Each bar in a bar chart is represented with a node. The user can interact with the bars in a bar chart, by adding event handlers to the nodes representing the data items. Please refer to the section on the pie chart for an example in which you added tooltips for the pie slices.

## Styling the BarChart with CSS

By default, a `BarChart` is given style-class names: `chart` and `bar-chart`.

The following style sets the default values for the `barGap` and `categoryGap` properties for all bar charts to `0px` and `20px`. The bars in the same category will be placed next to each other.

```
.bar-chart {
    -fx-bar-gap: 0;
    -fx-category-gap: 20;
}
```

You can customize the appearance of the bars for each series or each data item in a series. Each data item in a `BarChart` is represented by a node. The node gets four default style-class names:

- `chart-bar`
- `series<i>`
- `data<j>`
- `default-color<k>`
- `negative`

In `series<i>`, `<i>` is the series index. For example, the first series is given the style-class name as `series0`, the second as `series1`, etc.

In `data<j>`, `<j>` is the index of the data item within a series. For example, the first data item in each series gets a style-class name as `data0`, the second as `data1`, etc.

In `default-color<k>`, `<k>` is the series color index. For example, each data item in the first series will get a style-class name as `default-color0`, in the second series `default-color1`, etc. The default CS defines only eight series colors. The value for `<k>` is equal to `(i%8)`, where `i` is the series index. That is, series colors will repeat if you have more than eight series in a bar chart. Please refer to the pie chart section on how to use unique colors for series with index greater than eight. The logic will be similar to the one used for a pie chart, with a difference that, this time, you will be looking up the `bar-legend-symbol` within a series instead of a `pie-legend-symbol`.

The `negative` class is added if the data value is negative.

Each legend item in a bar chart is given the following style-class names:

- `chart-bar`
- `series<i>`
- `bar-legend-symbol`
- `default-color<j>`

In `series<i>`, `<i>` is the series index. In `default-color<j>`, `<j>` is the color index of the series. The legend color will repeat, as the bar colors do, if the number of series exceeds 8.

The following style defines the color of the bars for the all data items in series with series index 0, 8, 16, 24, etc., as blue.

```
.chart-bar.default-color0 {
    -fx-bar-fill: blue;
}
```

## Understating the StackedBarChart

A stacked bar chart is a variation of the bar chart. In a stacked bar chart, the bars in a category are stacked. Except for the placement of the bars, it works the same way as the bar chart.

An instance of the `StackedBarChart` class represents a stacked bar chart. The bars can be placed horizontally or vertically. If the x-axis is a `CategoryAxis`, the bars are placed vertically. Otherwise, they are placed horizontally. Like the `BarChart`, one of the axes must be a `CategoryAxis` and the other a `ValueAxis/NumberAxis`.

The `StackedBarChart` class contains a `categoryGap` property that defines the gap between bars in adjacent categories. The default gap is 10px. Unlike the `BarChart` class, the `StackedBarChart` class does not contain a `barGap` property, as the bars in one category are always stacked.

The constructors of the `StackedBarChart` class are similar to the ones for the `BarChart` class. They let you specify the axes, chart data, and category gap.

There is one notable difference in creating the `CategoryAxis` for the `BarChart` and the `StackedBarChart`. The `BarChart` reads the categories values from the data whereas you must explicitly add all category values to the `CategoryAxis` for a `StackedBarChart`.

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Country");

// Must set the categories in a StackedBarChart explicitly.
Otherwise,
// the chart will not show bars.
xAxis.getCategories().addAll("China," "India," "Brazil," "UK,"
"USA");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population (in millions)");

StackedBarChart<String, Number> chart = new
StackedBarChart<>(xAxis, yAxis);
```

The program in [Listing 23-11](#) shows how to create a vertical stacked bar chart. The chart is shown in [Figure 23-9](#). To create a horizontal stacked bar chart, use a CategoryAxis as the y-axis.

### **[Listing 23-11.](#) Creating a Vertical Stacked Bar Chart**

```
// VerticalStackedBarChart.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.StackedBarChart;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class VerticalStackedBarChart extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("Country");

        // Must set the categories in a StackedBarChart
        // explicitly. Otherwise,
        // the chart will not show any bars.
        xAxis.getCategories().addAll("China," "India,"
        "Brazil," "UK," "USA");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population (in millions)");

        StackedBarChart<String, Number> chart =
            new StackedBarChart<>(xAxis, yAxis);
        chart.setTitle("Population by Country and Year");

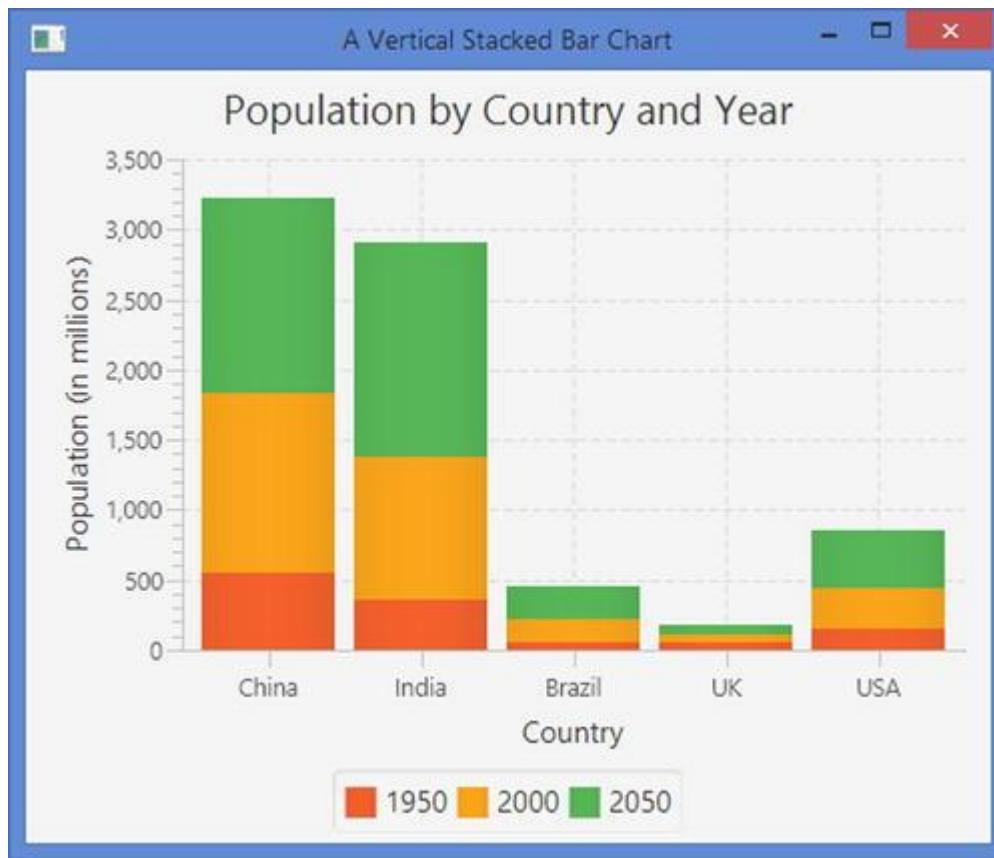
        // Set the data for the chart
        ObservableList<XYChart.Series<String, Number>>
chartData =
            XYChartDataUtil.getYearSeries();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Vertical Stacked Bar Chart");
        stage.show();
    }
}
```

```

    }
}

```



**Figure 23-9.** A vertical stacked bar chart

### Styling the StackedBarChart with CSS

By default, a `StackedBarChart` is given style-class names: `chart` and `stacked-bar-chart`.

The following style sets the default value for the `categoryGap` properties for all stacked bar charts to `20px`. The bars in a category will be placed next to each other.

```

.stackeds-bar-chart {
    -fx-category-gap: 20;
}

```

In a stacked bar chart, the style-class names assigned to the nodes representing bars and legend items are the same as that of a bar chart. Please refer to the section *Styling the BarChart with CSS* for more details.

### Understanding the ScatterChart

A bar chart renders the data items as symbols. All data items in a series use the same symbol. The location of the symbol for a data item is determined by the values on the data item along the x-axis and y-axis.

An instance of the `ScatterChart` class represents a scatter chart. You can use any type of `Axis` for the x-axis and y-axis. The class does not define any additional properties. It contains constructors that allow you to create scatter chart by specifying axes and data.

- `ScatterChart(Axis<X> xAxis, Axis<Y> yAxis)`
- `ScatterChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data)`

Recall that the `autoRanging` for an `Axis` is set to true by default. If you are using numeric values in a scatter chart, make sure to set the `autoRanging` to false. It is important to set the range of the numeric values appropriately to get uniformly distributed points in the chart. Otherwise, the points may be located densely in a small area and it will be hard to read the chart.

The program in [Listing 23-12](#) shows how to create and populate a scatter chart as shown in [Figure 23-10](#). Both axes are numeric axes. The x-axis is customized. The `autoRanging` is set to false; reasonable lower and upper bounds are set. The tick unit is set to 50. If you do not customize these properties, the `ScatterChart` will automatically determine them and the chart data will be hard to read.

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("Year");
xAxis.setAutoRanging(false);
xAxis.setLowerBound(1900);
xAxis.setUpperBound(2300);
xAxis.setTickUnit(50);
```

### **Listing 23-12.** Creating a Scatter Chart

```
// ScatterChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.ScatterChart;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ScatterChartTest extends Application {
    public static void main(String[] args) {
```

```
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Year");

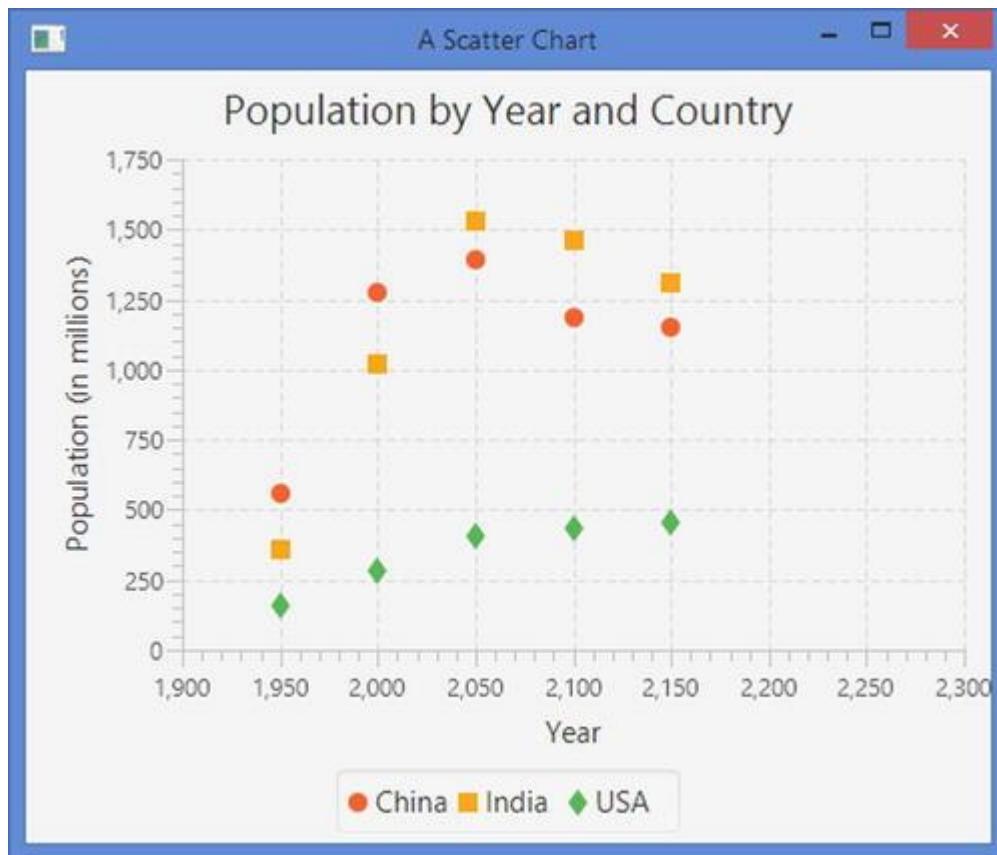
        // Customize the x-axis, so points are scattered
uniformly
        xAxis.setAutoRanging(false);
        xAxis.setLowerBound(1900);
        xAxis.setUpperBound(2300);
        xAxis.setTickUnit(50);

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population (in millions)");

        ScatterChart<Number,Number> chart = new
ScatterChart<>(xAxis, yAxis);
        chart.setTitle("Population by Year and Country");

        // Set the data for the chart
        ObservableList<XYChart.Series<Number,Number>>
chartData =
            XYChartDataUtil.getCountrySeries();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Scatter Chart");
        stage.show();
    }
}
```



**Figure 23-10.** A scatter chart

**Tip** You can use the `node` property for data items to specify symbols in a `ScatterChart`.

### Styling the ScatterChart with CSS

The `ScatterChart` is not assigned any additional style-class name other than `chart`.

You can customize the appearance of the symbols for each series or each data item in a series. Each data item in a `ScatterChart` is represented by a node. The node gets four default style-class names:

- `chart-symbol`
- `series<i>`
- `data<j>`
- `default-color<k>`
- `negative`

Please refer to the section *Styling the BarChart with CSS* for more details on the meanings of `<i>`, `<j>`, and `<k>` in these style-class names.

Each legend item in a scatter chart is given the following style-class names:

- chart-symbol
- series<i>
- data<j>
- default-color<k>

The following style will display the data items in the first series as triangles filled in blue. Note that only eight color series are defined. After that, colors are repeated as discussed at length in the section on the pie chart.

```
.chart-symbol.default-color0 {
    -fx-background-color: blue;
    -fx-shape: "M5, 0L10, 5L0, 5z";
}
```

## Understanding the LineChart

A line chart displays the data items in a series by connecting them by line segments. Optionally, the data points themselves may be represented by symbols. You can think of a line chart as a scatter chart with symbols in a series connected by straight line segments. Typically, a line chart is used to view the trend in data change over time or in a category.

An instance of the `LineChart` class represents a line chart. The class contains a `createSymbols` property, which is set to true by default. It controls whether symbols are created for the data points. Set it to false to show only straight lines connecting the data points in a series.

The `LineChart` class contains two constructors to create line charts by specifying axes and data.

- `LineChart(Axis<X> xAxis, Axis<Y> yAxis)`
- `LineChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X,Y>> data)`

The program in [Listing 23-13](#) shows how to create and populate a line chart as shown in [Figure 23-11](#). The program is the same as for using the scatter chart, except that it uses the `LineChart` class. The chart displays circles as symbols for data items. You can remove the symbols by using the following statement, after you create the line chart.

```
// Do not create the symbols for the data items
chart.setCreateSymbols(false);
```

### **Listing 23-13.** Creating a Line Chart

```
// LineChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
```

```
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class LineChartTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Year");

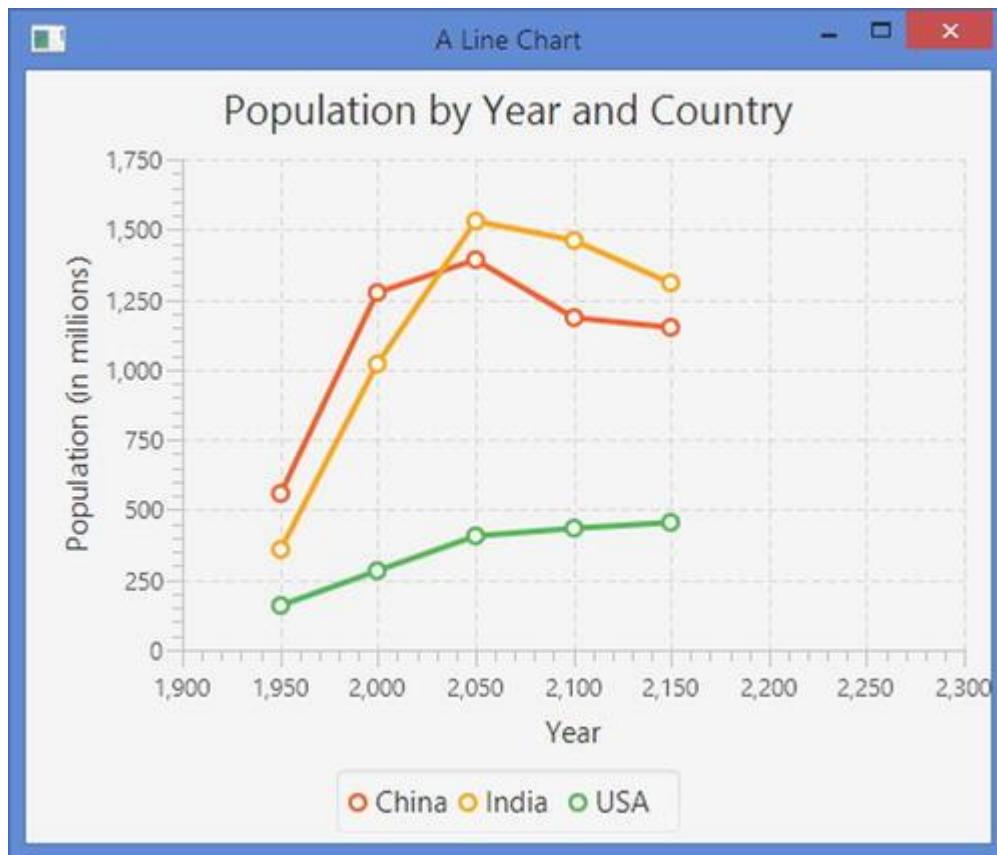
        // Customize the x-axis, so points are scattered
        uniformly
        xAxis.setAutoRanging(false);
        xAxis.setLowerBound(1900);
        xAxis.setUpperBound(2300);
        xAxis.setTickUnit(50);

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population (in millions)");

        LineChart<Number, Number> chart = new
LineChart<>(xAxis, yAxis);
        chart.setTitle("Population by Year and Country");

        // Set the data for the chart
        ObservableList<XYChart.Series<Number, Number>>
chartData =
            XYChartDataUtil.getCountrySeries();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Line Chart");
        stage.show();
    }
}
```



**Figure 23-11.** A line chart

### Styling the LineChart with CSS

The `LineChart` is not assigned any additional style-class name other than `chart`. The following style specifies that the `LineChart` should not create symbols.

```
.chart {
    -fx-create-symbols: false;
}
```

The `LineChart` creates a `Path` node to show the lines connecting all data points for a series. A line for a series is assigned the following style-class names:

- `chart-series-line`
- `series<i>`
- `default-color<j>`

Here, `<i>` is the series index and `<j>` is the color index of the series.

If the `createSymbols` property is set to true, a symbol is created for each data point. Each symbol node is assigned the following style-class name:

- chart-line-symbol
- series<i>
- data<j>
- default-color<k>

Here, <i> is the series index, <j> is the data item index within a series, and <k> is the color index of the series.

Each series is assigned a legend item, which gets the following style-class names:

- chart-line-symbol
- series<i>
- default-color<j>

The following styles set the line stroke for the color index 0 of series to blue. The symbol is for the series is also shown in blue.

```
.chart-series-line.default-color0 {
    -fx-stroke: blue;
}

.chart-line-symbol.default-color0 {
    -fx-background-color: blue, white;
}
```

## Understating the BubbleChart

A bubble chart is very similar to a scatter chart, except that it has the ability to represent three values for a data point. A bubble is used to represent a data items in series. You can set the radius of the bubble to represent the third value for the data point.

An instance of the `BubbleChart` class represents a bubble chart. The class does not define any new properties. A bubble chart uses the `extraValue` property of the `XYChart.Data` class to get the radius of the bubble. The bubble is an ellipse whose radii are scaled based on the scale used for the axes. Bubbles look more like a circle (or less stretched on one direction) if the scales for x-axis and y-axis are almost equal.

**Tip** The bubble radius is set by default, which is scaled using the scale factor of the axes. You may not see the bubbles if the scale factor for axes are very small. To see the bubbles, set the `extraValue` in data items to a high value or use a higher scale factors along the axes.

The `BubbleChart` class defines two constructors:

- `BubbleChart(Axis<X> xAxis, Axis<Y> yAxis)`

- `BubbleChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data)`

The program in [Listing 23-14](#) shows how to create a bubble chart as shown in [Figure 23-12](#). The chart data is passed to the `setBubbleRadius()` method, which explicitly sets the `extraValue` for all data points to 20px. If you want to use the radii of bubbles to represent another dimension of data, you can set the `extraValue` accordingly.

### ***[Listing 23-14](#). Creating a Bubble Chart***

```
// BubbleChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.BubbleChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class BubbleChartTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Year");

        // Customize the x-axis, so points are scattered
uniformly
        xAxis.setAutoRanging(false);
        xAxis.setLowerBound(1900);
        xAxis.setUpperBound(2300);
        xAxis.setTickUnit(50);

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population (in millions)");

        BubbleChart<Number, Number> chart = new
BubbleChart<>(xAxis, yAxis);
        chart.setTitle("Population by Year and Country");

        // Get the data for the chart
        ObservableList<XYChart.Series<Number, Number>>
chartData =
            XYChartDataUtil.getCountrySeries();
```

```
// Set the bubble radius
setBubbleRadius(chartData);

// Set the data for the chart
chart.setData(chartData);

StackPane root = new StackPane(chart);
Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("A Bubble Chart");
stage.show();
}

private void
setBubbleRadius(ObservableList<XYChart.Series<Number, Number>>
chartData) {
    for(XYChart.Series<Number, Number> series: chartData)
    {
        for(XYChart.Data<Number, Number> data
: series.getData()) {
            data.setExtraValue(20); // Bubble radius
        }
    }
}
```

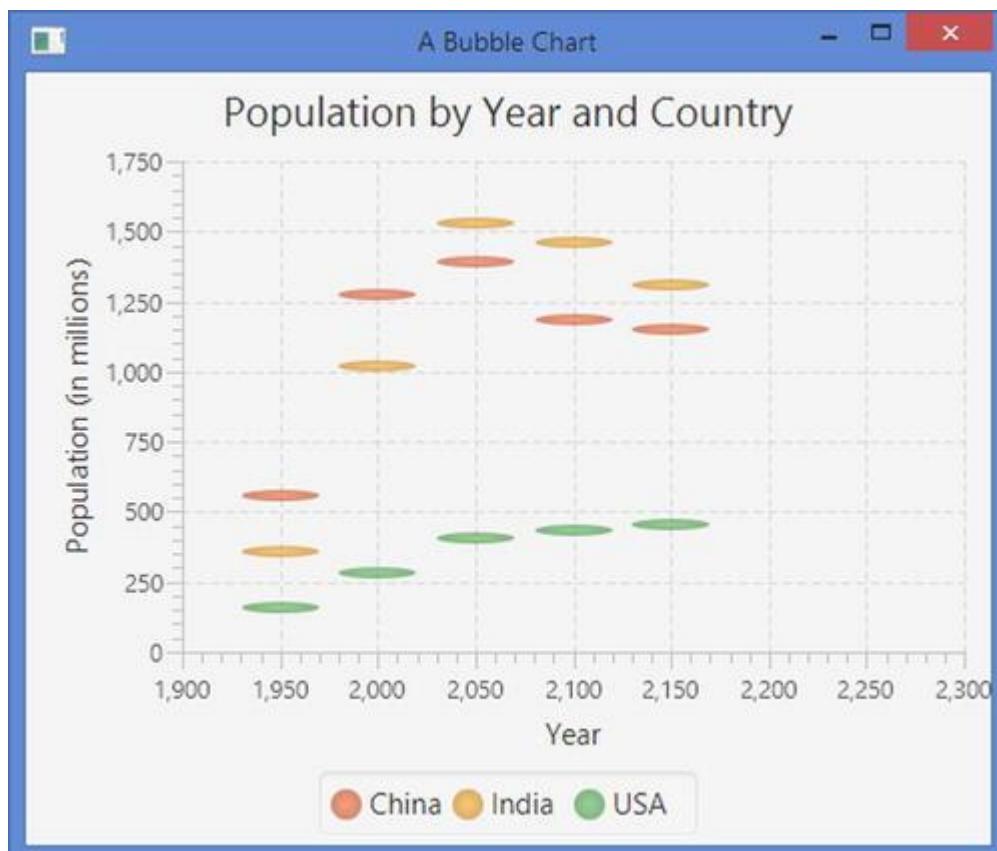


Figure 23-12. A bubble chart

## Styling the BubbleChart with CSS

The `BubbleChart` is not assigned any additional style-class name other than `chart`.

You can customize the appearance of the bubbles for each series or each data item in a series. Each data item in a `BubbleChart` is represented by a node. The node gets four default style-class names:

- `chart-bubble`
- `series<i>`
- `data<j>`
- `default-color<k>`

Here, `<i>` is the series index, `<j>` is the data item index within a series, and `<k>` is the color index of the series.

Each series is assigned a legend item, which gets the following style-class names:

- `chart-bubble`
- `series<i>`
- `bubble-legend-symbol`
- `default-color<k>`

Here, `<i>` and `<k>` have the same meanings as described above.

The following style sets the fill color for the series color index 0 to blue. The bubbles and legend symbols for the data items in the first series will be displayed in blue. The color will repeat for series index 8, 16, 24, etc.

```
.chart-bubble.default-color0 {
    -fx-bubble-fill: blue;
}
```

## Understating the AreaChart

The area chart is a variation of the line chart. It draws lines connecting all data items in a series and, additionally, fills the area between where the line and the x-axis is painted. Different colors are used to paint areas for different series.

An instance of the `AreaChart` represents an area chart. Like the `LineChart`, class, the class contains a `createSymbols` property to control whether symbols are drawn at the data points. By default, it is set to true. The class contains two constructors:

- `AreaChart (Axis<X> xAxis, Axis<Y> yAxis)`

- `AreaChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data)`

The program in [Listing 23-15](#) shows how to create an area chart as shown in [Figure 23-13](#). There is nothing new in the program, except that you have used the `AreaChart` class to create the chart. Notice that the area for a series overlays the area for the preceding series.

### ***[Listing 23-15.](#) Creating an Area Chart***

```
// AreaChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.AreaChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class AreaChartTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Year");

        // Customize the x-axis, so points are scattered
uniformly
        xAxis.setAutoRanging(false);
        xAxis.setLowerBound(1900);
        xAxis.setUpperBound(2300);
        xAxis.setTickUnit(50);

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population (in millions)");

        AreaChart<Number, Number> chart = new
AreaChart<>(xAxis, yAxis);
        chart.setTitle("Population by Year and Country");

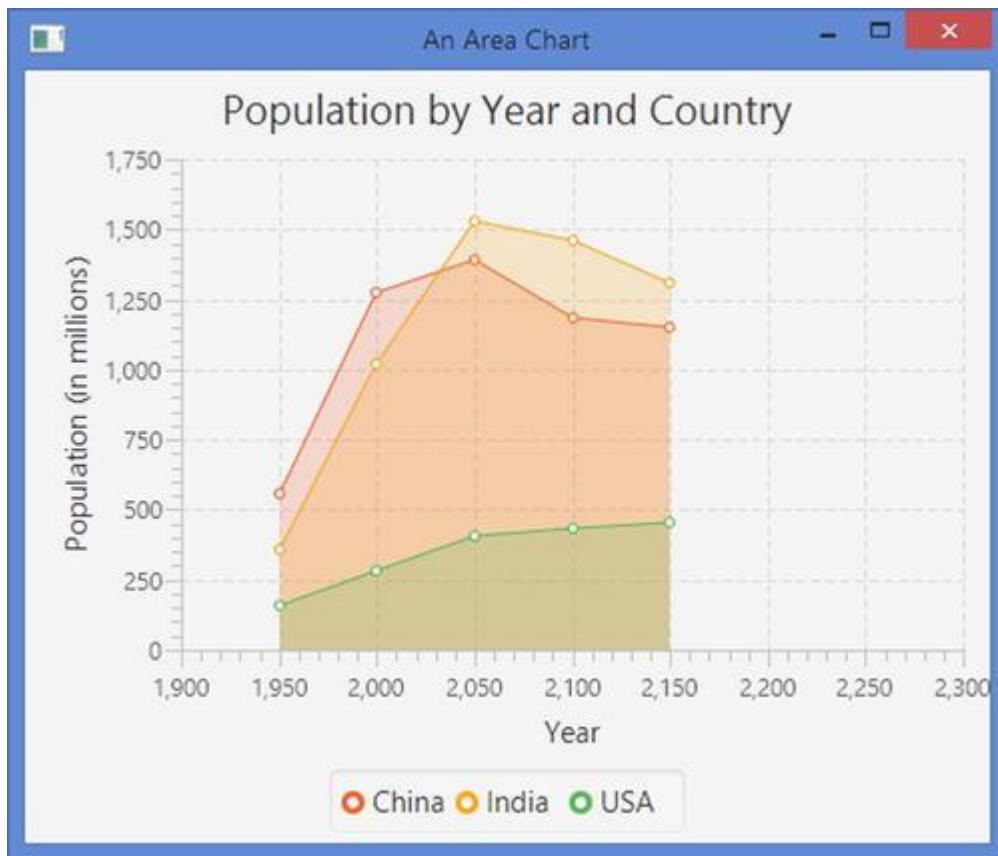
        // Set the data for the chart
        ObservableList<XYChart.Series<Number, Number>>
chartData =
            XYChartDataUtil.getCountrySeries();
        chart.setData(chartData);

        StackPane root = new StackPane(chart);
    }
}
```

```

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("An Area Chart");
        stage.show();
    }
}

```



**Figure 23-13.** An area chart

### Styling the AreaChart with CSS

The `AreaChart` is not assigned any additional style-class name other than `chart`. The following style specifies that the `AreaChart` should not create symbols for representing the data points.

```

.chart {
    -fx-create-symbols: false;
}

```

Each series in an `AreaChart` is represented by a `Group` containing two `Path` nodes. One `Path` represents the line segment connecting all data points in the series, and another `Path` represents the area covered by the series. The `Path` node representing the line segment for a series is assigned the following style-class names:

- `chart-series-area-line`

- `series<i>`
- `default-color<j>`

Here, `<i>` is the series index and `<j>` is the color index of the series.

The `Path` node representing the area for a series is assigned the following style-class names:

- `chart-series-area-fill`
- `series<i>`
- `default-color<j>`

Here, `<i>` is the series index and `<j>` is the color index of the series.

If the `createSymbols` property is set to true, a symbol is created for each data point. Each symbol node is assigned the following style-class name:

- `chart-area-symbol`
- `series<i>`
- `data<j>`
- `default-color<k>`

Here, `<i>` is the series index, `<j>` is the data item index within a series, and `<k>` is the color index of the series.

Each series is assigned a legend item, which gets the following style-class names:

- `chart-area-symbol`
- `series<i>`
- `area-legend-symbol`
- `default-color<j>`

Here, `<i>` is the series index and `<j>` is the color index of the series.

The following style sets the area fill color for the color index 0 for the series to blue with 20% opacity. Make sure to set transparent colors for the area fills as areas overlap in an `AreaChart`.

```
.chart-series-area-fill.default-color0 {  
    -fx-fill: rgba(0, 0, 255, 0.20);  
}
```

The following styles set the blue as the color for symbols, line segment, and legend symbol for the color index 0 for the series.

```
/* Data point symbols color */  
.chart-area-symbol.default-color0. {  
    -fx-background-color: blue, white;  
}
```

```

/* Series line segment color */
.chart-series-area-line.default-color0 {
    -fx-stroke: blue;
}

/* Series legend symbol color */
.area-legend-symbol.default-color0 {
    -fx-background-color: blue, white;
}

```

## Understanding the StackedAreaChart

The stacked area chart is a variation of the area chart. It plots data items by painting an area for each series. Unlike the area chart, areas for series do not overlap; they are stacked.

An instance of the `StackedAreaChart` represents a stacked area chart. Like the `AreaChart` class, the class contains a `createSymbols` property. The class contains two constructors:

- `StackedAreaChart(Axis<X> xAxis, Axis<Y> yAxis)`
- `StackedAreaChart(Axis<X> xAxis, Axis<Y> yAxis, ObservableList<XYChart.Series<X, Y>> data)`

The program in [Listing 23-16](#) shows how to create a stacked area chart as shown in [Figure 23-14](#). The program is the same as the one that created an `AreaChart`, except that you have used the `StackedAreaChart` class to create the chart.

### **Listing 23-16.** Creating a Stacked Area Chart

```

// StackedAreaChartTest.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.StackedAreaChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class StackedAreaChartTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}

```

```
@Override
public void start(Stage stage) {
    NumberAxis xAxis = new NumberAxis();
    xAxis.setLabel("Year");

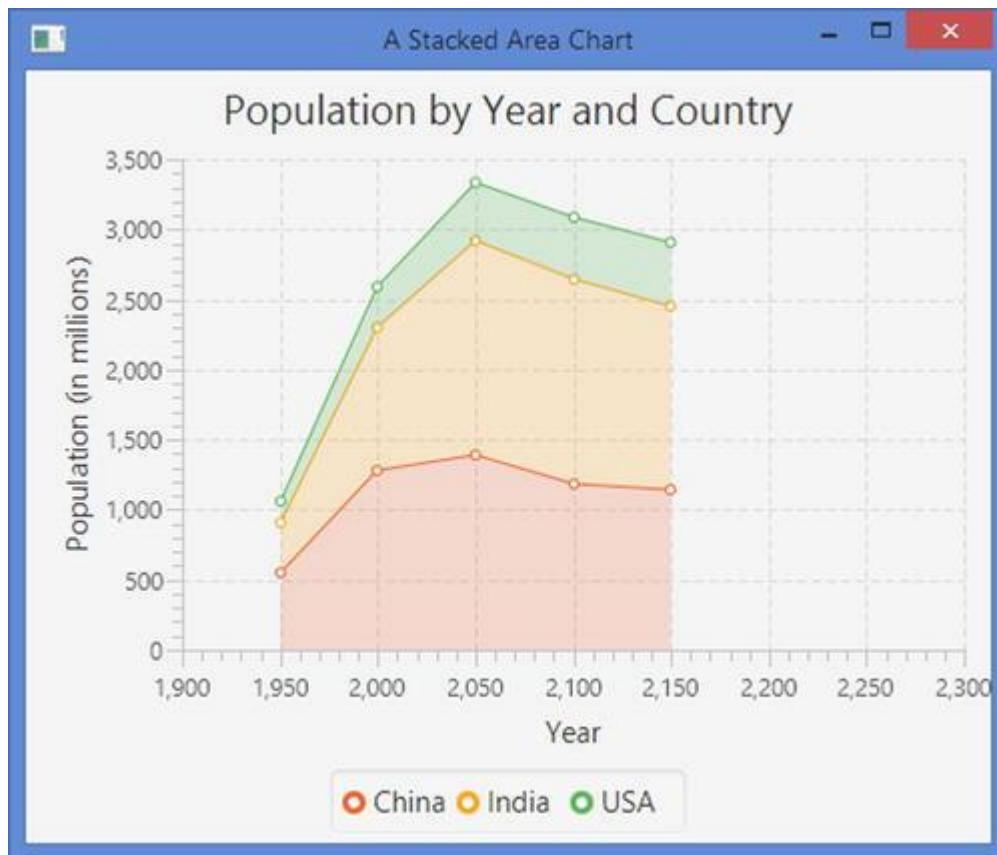
    // Customize the x-axis, so points are scattered
uniformly
    xAxis.setAutoRanging(false);
    xAxis.setLowerBound(1900);
    xAxis.setUpperBound(2300);
    xAxis.setTickUnit(50);

    NumberAxis yAxis = new NumberAxis();
    yAxis.setLabel("Population (in millions)");

    StackedAreaChart<Number,Number> chart = new
StackedAreaChart<>(xAxis, yAxis);
    chart.setTitle("Population by Year and Country");

    // Set the data for the chart
    ObservableList<XYChart.Series<Number,Number>>
chartData =
        XYChartDataUtil.getCountrySeries();
    chart.setData(chartData);

    StackPane root = new StackPane(chart);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("A Stacked Area Chart");
    stage.show();
}
}
```



**Figure 23-14.** A stacked area chart

### Styling the StackedAreaChart with CSS

Styling a StackedAreaChart is the same as styling an AreaChart. Please refer to the section *Styling the AreaChart with CSS* for more details.

### Customizing XYChart Appearance

You have seen how to apply chart-specific CSS styles to customize the appearance of charts. In this section, you will look at some more ways to customize XYChart plot and axes. The XYChart class contains several boolean properties to change the chart plot appearance:

- alternativeColumnFillVisible
- alternativeRowFillVisible
- horizontalGridLinesVisible
- verticalGridLinesVisible
- horizontalZeroLineVisible
- verticalZeroLineVisible

The chart area is divided into a grid of columns and rows. Horizontal lines are drawn passing through major ticks on the y-axis making up

rows. Vertical lines are drawn passing through major ticks on the x-axis making up columns.

### Setting Alternate Row/Column Fill

The `alternativeColumnFillVisible` and `alternativeRowFillVisible` control whether alternate columns and rows in the grid are filled. By default, `alternativeColumnFillVisible` is set to false and `alternativeRowFillVisible` is set to true.

As of time of this writing, setting the `alternativeColumnFillVisible` and `alternativeRowFillVisible` properties do not have any effects in JavaFX 8, which uses Modena CSS by default. There are two solutions. You can use the Caspian CSS for your application using the following statement:

```
Application.setUserAgentStylesheet(Application.STYLESHEET_CASPIAN);
```

The other solution is to include the following styles in your application CSS.

```
.chart-alternative-column-fill {
    -fx-fill: #eeeeee;
    -fx-stroke: transparent;
    -fx-stroke-width: 0;
}

.chart-alternative-row-fill {
    -fx-fill: #eeeeee;
    -fx-stroke: transparent;
    -fx-stroke-width: 0;
}
```

These styles are taken from Caspian CSS. These styles set the `fill` and `stroke` properties to null in Modena CSS.

### Showing Zero Line Axes

The axes for a chart may not include zero lines. Whether zero lines are included depends on the lower and upper bounds represented by the axes.

The `horizontalZeroLineVisible` and `verticalZeroLineVisible` control whether zero lines should be visible. By default, they are visible. Note that the zero line for an axis is visible only when the axis has both positive and negative data to plot. If you have negative and positive values along the y-axis, an additional horizontal axis will appear indicating the zero value along the y-axis. The same rule applies for values along the x-axis. If the range for an axis is set explicitly using its lower and upper bounds, the visibility of the zero line depends on whether zero falls in the range.

## Showing Grid Lines

The `horizontalGridLinesVisible` and `verticalGridLinesVisible` specify whether the horizontal and vertical grid lines are visible. By default, both are set to true.

## Formatting Numeric Tick Labels

Sometimes, you may want to format the values displayed on a numeric axis. You want to format the labels for the numeric axis for different reasons:

- You want to add prefixes or suffixes to the tick labels. For example, you may want to display a number 100 as \$100 or 100M.
- You may be supplying the chart scaled data to get an appropriate scale value for the axis. For example, for the actual value 100, you may be supplying 10 to the chart. In this case, you would like to display the actual value 100 for the label.

The `ValueAxis` class contains a `tickLabelFormatter` property, which is a `StringConverter` and it is used to format tick labels. By default, tick labels for a numeric axis are formatted using a default formatter. The default formatter is an instance of the static inner class `NumberAxis.DefaultFormatter`.

In our examples of `XYChart`, you had set the label for the y-axis to “Population (in millions)” to indicate that the tick values on the axis are in millions. You can use a label formatter to append “M” to the tick values to indicate the same meaning. The following snippet of code will accomplish this.

```
NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population");

// Use a formatter for tick labels on y-axis to append
// M (for millions) to the population value
yAxis.setTickLabelFormatter(new StringConverter<Number>() {
    @Override
    public String toString(Number value) {
        // Append M to the value
        return Math.round(value.doubleValue()) + "M";
    }

    @Override
    public Number fromString(String value) {
        // Strip M from the value
        value = value.replaceAll("M", "");
    }
});
```

```

        return Double.parseDouble(value);
    }
});

```

The `NumberAxis.DefaultFormatter` works better for adding a prefix or suffix to tick labels. This formatter is kept in sync with the `autoRanging` property for the axis. You can pass a prefix and a suffix to the constructor. The following snippet of code accomplishes the same thing as the above snippet of code.

```

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population");
yAxis.setTickLabelFormatter(new
NumberAxis.DefaultFormatter(yAxis, null, "M"));

```

You can customize several visual aspects of an `Axis`. Please refer to the API documentation for the `Axis` class and its subclasses for more details.

The program in [Listing 23-17](#) shows how to customize a line chart. The chart is shown in [Figure 23-15](#). It formats the tick labels on the y-axis to append “M” to the label value. It hides the grid lines and shows the alternate column fills.

### ***[Listing 23-17](#). Formatting Tick Labels and Customizing Chart Plot***

```

// CustomizingCharts.java
package com.jdojo.chart;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class CustomizingCharts extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Set caspian CSS to get alternate column fills
        // until modena CSS is fixed
        Application.setUserAgentStylesheet(Application.STYLES
HEET_CASPIAN);

        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("Year");

        // Customize the x-axis, so points are scattered
uniformly
        xAxis.setAutoRanging(false);
    }
}

```

```
        xAxis.setLowerBound(1900);
        xAxis.setUpperBound(2300);
        xAxis.setTickUnit(50);

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Population");

        // Use a formatter for tick labels on y-axis to
        // append
        // M (for millions) to the population value
        yAxis.setTickLabelFormatter(new
NumberAxis.DefaultFormatter(yAxis, null, "M"));

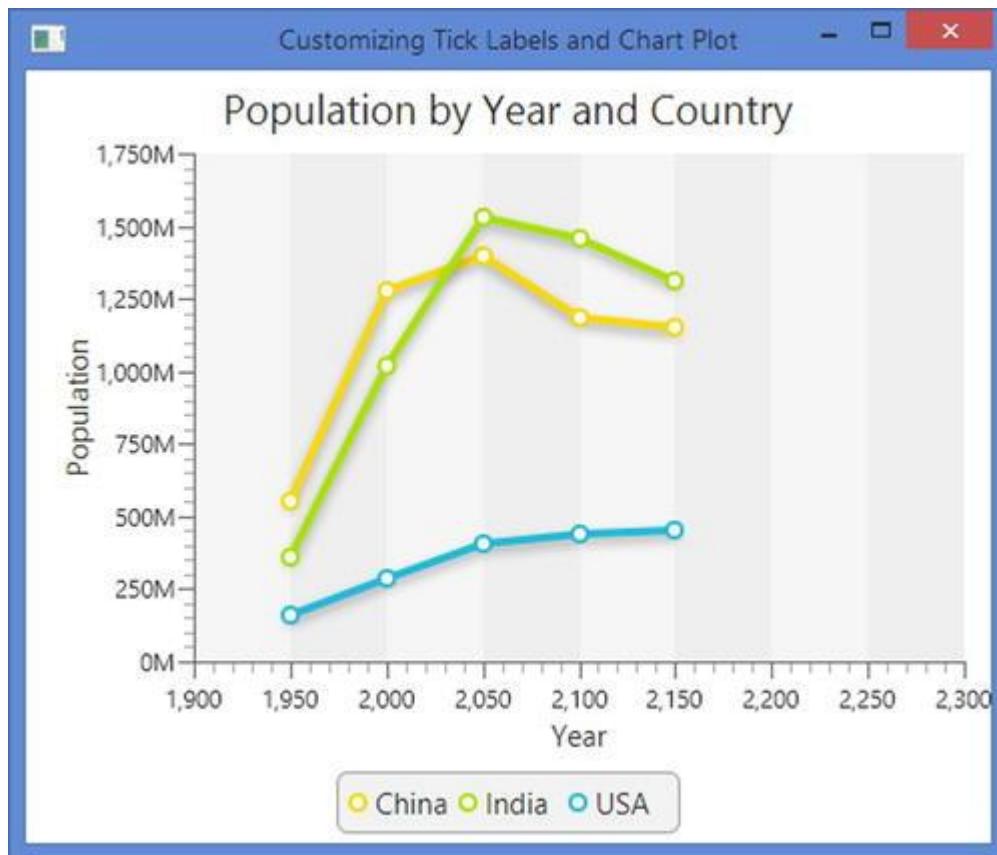
        LineChart<Number, Number> chart = new
LineChart<>(xAxis, yAxis);
        chart.setTitle("Population by Year and Country");

        // Set the data for the chart
        chart.setData(XYChartDataUtil.getCountrySeries());

        // Show alternate column fills
        chart.setAlternativeColumnFillVisible(true);
        chart.setAlternativeRowFillVisible(false);

        // Hide grid lines
        chart.setHorizontalGridLinesVisible(false);
        chart.setVerticalGridLinesVisible(false);

        StackPane root = new StackPane(chart);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Customizing Tick Labels and Chart
Plot");
        stage.show();
    }
}
```



**Figure 23-15.** A line chart with formatted tick labels and customized plot

## Summary

A chart is a graphical representation of data. Charts provide an easier way to analyze large volume of data visually. Typically, they are used for reporting purposes. Different types of charts exist. They differ in the way they represent the data. Not all types of charts are suitable for analyzing all types of data. For example, a line chart is suitable for understanding the comparative trend in data whereas a bar chart is suitable for comparing data in different categories.

JavaFX supports charts, which can be integrated in a Java application by writing few lines of code. It contains a comprehensive, extensible Chart API that provides built-in support for several types of charts. The Chart API consists of a number of predefined classes in the `javafx.scene.chart` package. Few of those classes are `Chart`, `XYChart`, `PieChart`, `BarChart` and `LineChart`.

The abstract `Chart` is the base class for all charts. It inherits the `Node` class. Charts can be added to a scene graph. They can also be styled with CSS as any other nodes. Every chart in JavaFX has three parts: a title, a legend, and data. Different types of charts define their

data differently. The `Chart` class contains the properties to deal with the title and legend.

A chart can be animated. The `animated` property in the `Chart` class specifies whether the change in the content of the chart is shown with some type of animation. By default, it is `true`.

A pie chart consists of a circle divided into sectors of different central angles. Typically, a pie is circular. The sectors are also known as *pie pieces* or *pie slices*. Each sector in the circle represents a quantity of some kind. The central angle of the area of a sector is proportional to the quantity it represents. An instance of the `PieChart` class represents a pie chart.

A bar chart renders the data items as horizontal or vertical rectangular bars. The lengths of the bars are proportional to the value of the data items. An instance of the `BarChart` class represents a bar chart.

A stacked bar chart is a variation of the bar chart. In a stacked bar chart, the bars in a category are stacked. Except for the placement of the bars, it works the same way as the bar chart. An instance of the `StackedBarChart` class represents a stacked bar chart.

A scatter chart renders the data items as symbols. All data items in a series use the same symbol. The location of the symbol for a data item is determined by the values on the data item along the x-axis and y-axis. An instance of the `ScatterChart` class represents a scatter chart.

A line chart displays the data items in a series by connecting them by line segments. Optionally, the data points themselves may be represented by symbols. You can think of a line chart as a scatter chart with symbols in a series connected by straight line segments. Typically, a line chart is used to view the trend in data change over time or in a category. An instance of the `LineChart` class represents a line chart.

A bubble chart is very similar to a scatter chart, except that it has ability to represent three values for a data point. A bubble is used to represent a data items in series. You can set the radius of the bubble to represent the third value for the data point. An instance of the `BubbleChart` class represents a bubble chart.

The area chart is a variation of the line chart. It draws lines connecting all data items in a series and, additionally, fills the area between where the line and the x-axis is painted. Different colors are used to paint areas for different series. An instance of the `AreaChart` represents an area chart.

The stacked area chart is a variation of the area chart. It plots data items by painting an area for each series. Unlike the area chart, areas for

series do not overlap; they are stacked. An instance of the `StackedAreaChart` represents a stacked area chart.

Besides using CSS to customize the appearance of charts, the Chart API provides several properties and methods to customize charts' appearance such as adding alternate row/column fills, showing zero line axes, showing grid lines, and formatting numeric tick labels.

The next chapter will discuss how to work with images in JavaFX using the Image API.