Chapter 5, Inheritance and GUIs

John M. Morrison

March 20, 2020

Contents

0 What is ahead?

So far, we have been programming "in the small." We have created simple classes that carry out fairly straightforward chores. Our programs have been little one or two class programs. One class has been the class you are writing, the other has been the interactions pane or a simple driver class with a `main` method in it. We created the `BigFraction` API, which allows a client programmer using it to do exact, extended-precision rational arithmetic.

So far, the relationship between classes has been a "has-a relationship." For example our `BigFraction` class has two `BigInteger`s, representing the numerator and denominator of our fraction object. We often use instances of classes that we attach to local variables inside of methods. This is a "uses-a" instead of a "has-a" relationship. Both of these relationships are `compositional`, since we are using them to compose, or build, our class. The compositional relationship is the most important and most common relationship between classes.

Java programs often consist of many classes, which work together to do a job. Sometimes we will create classes from scratch, sometimes we will aggregate various types of objects in a class, and sometimes we will customize existing classes using *inheritance*. We will also draw upon Java's vast class libraries. We will also see how to tie related classes together by using *interfaces*.

0.1 GUIs

The acronym GUI stands for *graphical user interface*. This is the means by which you interact with a modern computer. GUIs allow you to use a pointing device such as a mouse, touchpad, or touchscreen and the keyboard to interact

with your computer. They allow you to have several processes open at once and for each to occupy a different window on your desktop.

You use a windowing system; each window holds a program and displays what the program running in it is doing. These windows are containers; the windows themselves along with the visible features inside of them are called *widgets*. At the top of each window is a *title bar*, this often holds the name of the app you are using, some buttons that hide, maximize or dispose of the window, and possibly the name of a file you have open.

The main part of the window is called the *content pane*; this is where all of the "good stuff" goes. Within the content pane there may be *controls* that command the application to do things. Examples of these include widgets such as text areas, menus and menu items, scrollbars, buttons, labels, or checkboxes. If the application has menus, it will have a *menu bar* that holds the menus. It might also have a toolbar, which would hold other controls or a status bar that sends messages to the user.

A widget is a *top-level widget* if it can reside directly on the desktop. A top level window in `javafx` is called a *Stage*; stages are instances of the class `javafx.stage.Stage`. Stages can exist directly on your computer's desktop. Inside of a `Stage` goes one or more `scenes`. Scenes are containers that hold objects called *Nodes*. To properly use the `javafx` GUI framework, we first must begin by taking a look at the relationships that can exist between classes.

We will begin by looking at inheritance; this programming construct is essential to GUI programming. Heretofore, we have done object-based programming; we now enter the domain of object-*oriented* programming. Your careful attention to this material will be amply repaid when we begin creating GUIs.

1 Inheritance

Inheritance provides a mechanism by which we can customize the capabilities of existing classes to meet our needs. It can also be used as a tool to eliminate a lot of duplicate code which is a continuing maintenance headache. Finally, it will provide us with a means of enjoying the advantages of *polymorphism*, the ability of a variable to point at objects of a variety of different but related types.

Be wary, however, of the peril that the possession of a hammer makes everything look like a nail. Inheritance, as we shall see, is a tool that should be used judiciously. One reason you need to be careful is that any class (save for `Object`) has exactly one parent. Java does not support "multiple inheritance" that you can see in Python or C++. It has another mechanism called *interfaces* which does nearly the same thing, and which avoids a potentially serious source of intransigent programming bugs.

The new keyword you will see is `extends`; the relationship you create is an "is-a" relationship. We will create a small example by creating a suite of classes pertaining to geometric shapes.

Let us begin by creating a class for general shapes and putting method appropriate method stubs into it.

```
public class Shape
{
    public double area()
    {
        return 0;
    }
    public double perimeter()
    {
        return 0;
    }
    public double diameter()
    {
        return 0;
    }
}
```

Since we have no idea what kind of shape we are going to be working with, this seems the best possible solution. We will use it for now to get things going for now; later we will see smarter ways for managing a class such as this one.

Now we will create a `Rectangle` class. Note the use of the keyword `extends`. Note that `extends` creates an "is-a" relationship; a `Rectangle` is a `Shape`.

```
public class Rectangle extends Shape
{
    private double width;
    private double height;
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }
    public Rectangle()
    {
        this(0,0);
    }
    public double area()
    {
        return height*width;
```

```
    }
    public double perimeter()
    {
        return 2*(height + width);
    }
    public double diameter()
    {
        return Math.hypot(height, width);
    }
}
```

Next, we create a Circle class. Both these classes extend `Shape`, so they are *sibling* classes.

```
public class Circle extends Shape
{
    private double radius;
    public Circle(double radius)
    {
        this.radius = radius;
    }
    public Circle()
    {
        this(0);
    }
    public double area()
    {
        return Math.PI*radius*radius;
    }
    public double perimeter()
    {
        return 2*Math.PI*radius;
    }
    public double diameter()
    {
        return 2*radius;
    }
}
```

A square is indeed, a rectangle, so we will create a `Square` class by extending `Rectangle`. Note the use of the `super` keyword here. It calls the parent constructor. If you are going to use `super`, it must be in the first line of your constructor, or your program will fail to compile, and the compiler will send you a stinging reminder of this fact.

```
public class Square extends Rectangle
{
    private double side;
    public Square(double side)
    {
        super(side, side);
        side = side;
    }
}
```

So in our little class hierarchy here, we have the root class `Shape`. Then `Rectangle` and `Circle` are children of `Shape`. Finally, `Square` is a child of `Rectangle`.

Now you shall see that the type of variable you use is very important. Let us begin an interactive session. In this session we create a 6×8 rectangle and find its area, perimeter and circumference. The type of `r` is `Rectangle`.

```
> Rectangle r = new Rectangle(6,8);
> r.area()
48.0
> r.diameter()
10.0
> r.perimeter()
28.0
```

Now watch this.

```
> r = new Square(5);
> r.area()
25.0
> r.perimeter()
20.0
> r.diameter()
7.0710678118654755
>
```

We have been saying all along that a variable can only point at an object of its own type. But now we have a `Rectangle` pointing at a `Square`. Why can we do this?

The `Square` class is a child class of `Rectangle`, so that means a `Square` is a `Rectangle`! To wit, if you have a variable of a given type, it can point at any object of a descendant type. This phenomenon is a form of *polymorphism*. So, one of the benefits of inheritance is polymorphism. An easy way to remember this is, "Variables can point down the inheritance tree."

You might ask now, "Why not make everything an `Object` and save a lot of work?" Then all will look just like Python's duck-typing system. Let us try that here.

```
> Object o = new Square(5);
> o.diameter()
Error: No 'diameter' method in
    'java.lang.Object' with arguments: ()
> ((Square) o).diameter()
7.0710678118654755
>
```

We are quickly rebuked. Variables of type `Object` can only see the methods of the `Object` class. Remember, Java is a statically typed language, so methods visible to a variable must be known at compile time.

Since our `Square` has a method `diameter()`, we would have to cast the `Object` variable to a `Square` before calling `diameter`. That is really a graceless solution to the problem and a last resort. There is an important trade-off here: variables of more general type can see more types of objects, but at the same time, they may see fewer methods.

The moral of this fable is thus: Use variable types that are as general as you need to have the desired flexibility, but not too general. In our case, here, it would make sense to have `Shape` variables point at the various shapes.

Now let us have a `Shape` variable point at the different shapes and call their methods. Here we have a `Shape` variable pointing at all the different kinds of shapes. Notice how all of the methods work. Go ahead and test all three for each type.

```
> Shape s = new Circle(10);
> s.area()
314.1592653589793
> s = new Rectangle(12,5);
> s.diameter()
13.0
> s = new Square(10);
> s.perimeter()
40.0
>
```

Programming Exercise In this exercise, you will add to our little class hierarchy of shapes.

1. Create a class named `Triangle.java`. It should have three state variables

of type `double`, `side1`, `side2` `side3`. Create method stubs for `area`, `diameter` and `perimeter`. Create a constructor to initialize state.

2. Create a class named `EquilateralTriangle` extending `Triangle`.
3. Look up *Herrons Formula* for finding the area of a triangle from three sides. Use it to compute the area of a triangle.
4. Note that this formula has a square-root in it. If the radicand is negative, you have an illegal triangle (such as a 1-1-5 triangle). Make a private static method computing the radicand called

   ```
   public static double squaredArea(double a, double b, double c)
   {
       //return the value of the evaluated radicand
   }
   ```

   Now insert this code in your constructor

   ```
   if(squaredArea(side1, side2, side3) < 0)
   {
       throw new IllegalArgumentException();
   }
   ```

5. Compile these programs. Try in `jshell` to call `new Triangle(1,1,5);` and watch an exploding heart occur. Test your new classes and see that they play nicely with your existing shapes.

1.1 Abstract Classes: A First Pass

Does the `Shape` class need these [really stupid] method bodies? As of now, yes. To add insult to injury, we do know that it makes absolutely no sense at all to create an actual `Shape` object. So, how do we shuck the silly method stubs and keep order in the kingdom here?

We make the methods of our `Shape` class be `abstract`. This allows us to dispose of the method bodies and have only method headers, ending with semi-colons. This also forces any child class of `Shape` to implement those methods. Bear this in mind if you are extending an abstract class. You can have variables of abstract class type which can point down the inheritance tree, just as regular class variables do.

To achieve our goal we first mark our `Shape` class abstract as follows. Here we see the original code with the class marked `abstract`.

```
public abstract class Shape
{
    public double area()
    {
        return 0;
```

```
    }
    public double perimeter()
    {
        return 0;
    }
    public double diameter()
    {
        return 0;
    }
}
```

The second step consists of making all of the methods inside abstract and amputating their bodies. Hello Henry VII!

```
public abstract class Shape
{
    public abstract double area();
    public abstract double perimeter();
    public abstract double diameter();
}
```

Look how svelte and pretty our class is! Gone are its useless *pro forma* method bodies. You should go back and retry the earlier examples with our new code. It all works nicely!

Go ahead and try to make a `new Shape()` and watch the compiler spew its ire all over your attempt. You cannot make instances of an abstract class.

So here are the rules of the road for abstract classes.

1. If you remove a method's body in a class you *must* declare that method `abstract`.
2. If *any* method in a class is declared `abstract`, the class itself must be declared `abstract`.
3. You may declare any class you create `abstract`. By so doing, you prevent any instances of it from being created; do this if it make no sense for an instance of your class to be created.
4. You may not create instances of any abstract class.
5. You can create variables of abstract class type. They can point at any object of any descendant type.
6. If you extend an abstract class and the child class is not abstract, you must implement all abstract methods in the abstract class, as well as any abstract methods in ancestor classes of the abstract class.

1.2 Polymorphism, Delegation, and Visibility

How does this polymorphism thing work? Suppose we have `Shape` variable pointing at a 12×5 rectangle. When we said "`s.diameter()`," here is what happened. The variable `s` sent the message to its object, "compute your diameter." The actual job of computing the diameter is delegated to the object to which `s` is pointing. Since the object pointed at by `s` is is a `Shape` object, we can be confident it will know how to compute its diameter. In fact, at that point in the code, `s` was pointing at a `Rectangle`, so the `Rectangle` computes its diameter and returns it when commanded to do so.

The variable type determines what methods can be seen and the job of actually carrying out the method is delegated to the object being pointed at by the variable. Objects "don't care" about the type of variable pointing at them. When they are prompted to do so, they dutifully execute their methods.

We summarize here with two principles

- **The Visibility Principle** The type of a variable pointing at an object determines what methods are visible. Only methods in the variable's class may be seen. Variables can have regular or abstract class type, since variables do not actually have any responsibility for executing code. This is because Java is statically typed; visible methods must be known at compile time.
- **The Delegation Principle** If a variable is pointing at an object and a visible method is called, the object is responsible for executing the method. Regardless of a variable's type, if a given method in the object is visible, the object's method will be called. Remember objects are strongly aware of their type so you can do this.

1.3 Understanding More of the API Guide

Go to the JavaFX API guide and bring up `Button`. Here is the family tree for `Button`. It can be seen right near the top of the page for `Button`.

```
java.lang.Object
javafx.scene.Node
javafx.scene.Parent
javafx.scene.layout.Region
javafx.scene.control.Control
javafx.scene.control.Labeled
javafx.scene.control.ButtonBase
javafx.scene.control.Button
```

The `Button` class in the `javafx.scene.control` package extends the old

`ButtonBase` class in the same package. You can see that there are several layers of inheritance here going up to the root class `Object`.

You should note that the package structure of the java class libraries and the inheritance structure are two different structures. The two hierarchies are more or less independent from one another.

You are not limited to using the methods listed in the method summary for `Button`. Scroll down below the method summary. You will see links for all the methods inherited from `ButtonBase`. Below this, methods are listed for all ancestor classes right up to `Object`. You can click on any named method and view its method detail on its home API page from the ancestor class.

Also on this page, you will see a Field Summary. Fields can be either state variables or static variables. You will notice that many of these are in caps. It is a universally-observed convention to put a variable name in caps when the variable is a constant. Most static fields you see will be constants.

One new keyword you should know about is `protected`. This is an access specifier that says, "Descendants can see but nobody else." It allows descendant classes access to state variables in ancestor classes. Ideally, it is better to avoid `protected`, to make everything `private`. We have seen how to use `super` to initialize state variables in a parent class. You will see the `protected` keyword fairly often in the API guide.

How do I know if a class is abstract? Look in the Java API guide and find the class `AbstractList`; clearly it will be abstract. Go to the top of the page. You will see the fully-qualified name, the family tree, and then its implemented interfaces and direct descendants. Just below that you see this

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

The first line tells all: See the word `abstract`?

So, in summary, you can declare any class abstract and instances of it cannot be created. You can declare methods in a class abstract and they cannot have a method body. Any child class must override these methods unless, it too, is abstract. Any class containing an abstract method must be marked abstract. However, an abstract class is not required to have any abstract methods.

1.4 The `@Override` Annotation

When you override a method of a parent class, you have the option of using the `@Override` annotation. This tells the compiler to check that you are overriding a method in a parent class. The compiler will verify that you are using a correct

signature and return type for the method you are overriding. If you do not use the correct signature, you might accidentally *overload* the inherited method instead of overriding it. Here we show how to use the annotation.

```
public class Bar
{
    int count(int x)
    {
        return x;
    }
}

public class Foo extends Bar
{
    @Override
    int count(int x)
    {
        return 2*x
    }
}
```

You should always use this annotation when overriding `Object`'s `toString()` and `equals(Object o)` methods. You will see it used throughout the rest of the book.

One protection afforded by this annotation is that it can prevent you from accidentally *overloading* a method instead of overriding it. To override a method, the child method must have the same signature and and name as the parent method. If the sigs differ, you will have just overloaded the parent method and the parent method will still be in effect as is. This is a big reason to use `@Override`.

1.5 Why Not Have Multiple Inheritance?

Class designers often speak of the "deadly diamond;" this is a big shortcoming of multiple inheritance and can cause it to produce strange behaviors. Imagine you have these four classes, Root, Left, Right and Bottom. Suppose that `Left` and `Right` extend `Root` and that `Bottom` were allowed to extend `Left` and `Right`.

Before proceeding, draw yourself a little inheritance diagram. Graphically these four classes create a cycle in the inheritance graph (*which in Java must be a rooted tree*).

Next, imagine that both the `Left` and `Right` classes implement a method `f` with identical signature and return type. Further, suppose that `Bottom` does

not have its own version of `f`; it just decides to inherit it. Now imagine seeing this code fragment

```
Bottom b = new Bottom(....);
b.f(...)
```

There is a sticky problem here: Do we call the `f` defined in the class `Left` or `Right`? If there is a conflict between these methods, the call is not well–defined in our scheme of inheritance.

Shortly, we will see that Java has a clever alternative that is nearly as useful as inheritance with none of the error-proneness.

1.6 A C++ Interlude

There is a famous example of multiple inheritance at work in C++. There is a class `ios`, with children `istream` and `ostream`. The familiar `iostream` class inherits from both `istream` and `ostream`. Since the methods for input and output do not overlap this works well.

However, the abuse of multiple inheritance in C++ has lead to a lot of very bad errors in code. Java's creators decided this advantage was outweighed by the error vulnerabilities of multiple inheritance.

The One-Parent Rule Every class has exactly one parent, except for `Object`, which is the root class. When you inherit from a class, you "blow your inheritance." The ability to inherit is very valuable, so we should only inherit when it yields significant benefits.

We will see how to circumnavigate the one-parent rule and obtain the benefits of polymorphism by using Java's `interface` construct. Next, you can see inheritance at work; we are going to make our first GUI by inheriting from the `Application` class in the package `javafx.application`.

2 An Application of Inheritance: A Short GUI Program

We shall do a little exploration the Java GUI classes and illustrate how inheritance affects GUI programs. We will be able to make classes that create windows, graphics, menus and buttons. We will use the term *widget* for graphical objects of this sort. We will introduce many core ideas in the language using graphical objects.

Four packages will become important to us as we develop GUI technique.

- `javafx.application` This contains the `Application` class, which will manage the life-cycle of a javaFX app.
- `javafx.stage` This contains top-level windows, including things such as file choosers, pop-up windows, and `Stage`, which is the top-level container window for an application.
- `javafx.scene` This package and its descendants includes a panoply of things we will press into service, These items include buttons, text areas, menus and slider bars. This is the home of many of widgets.
- `javafx.event` This packages hold classes that are useful in responding to such things as keystrokes, mouse clicks, and the selection of menu items. Things in these packages make buttons and other widgets "live."

Let us begin with a little exercise, in which we produce a program that makes a window and puts a button in the window.

```
import javafx.application.Application;
public class SimpleGUI extends Application
{
}
```

Your attempt to run this is rewarded with an error message.

```
SimpleGUI.java:2: error: SimpleGUI is not abstract and does not
override abstract method start(Stage) in Application
public class SimpleGUI extends Application
       ^
1 error
```

Now look in the javafx API guide in the class `Application`. We notice a couple of things. Firstly, we notice this.

```
public abstract class Application
extends Object
```

The class `Application` is abstract. Secondly, we see that the method `start` is marked `abstract`. We must implement this method or the compiler will not proceed.

We will need to add an import so the import police do not come down on us. We do this so the class `Stage` is visible.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleGUI extends Application
```

```
{
    @Override
    public void start(Stage primaryStage)
    {
    }
}
```

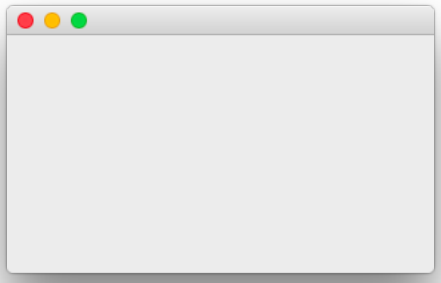
To execute this class we need a main. Here it is.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleGUI extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
    }
    public static void main(String[] args)
    {
    }
}
```

Run this and see nothing. Now for the body of the main. We are calling the static method `launch` in `Application` to launch the application.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleGUI extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

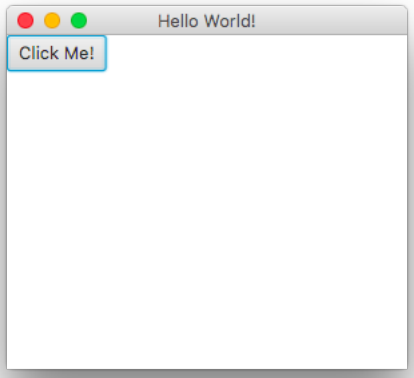
Run this and see an empty window.



Next, we add a button to the window.

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

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

    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Hello World!");
        Button aButton = new Button("Click Me!");
        Group root = new Group();
        root.getChildren().addAll(aButton);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Now when you run it, you will see a window pop up on your screen. In the title bar, you will see "Hello World!" The content pane will have a button in it labeled, "Click Me!"



If you are jaded and unimpressed, here is a look at Microsoft Foundation Classes using C++. Feast your eyes below and be appalled at the huge and puzzling program you have to write just to replicate the modest result here we just produced with four lines of code. What's worse is that we don't even get the button!

```
#include <afxwin.h>

class HelloApplication : public CWinApp
{
public:
  virtual BOOL InitInstance();
};

HelloApplication HelloApp;

class HelloWindow : public CFrameWnd
{
  CButton* m_pHelloButton;
public:
  HelloWindow();
};


BOOL HelloApplication::InitInstance()
{
  m_pMainWnd = new HelloWindow();
  m_pMainWnd->ShowWindow(m_nCmdShow);
```

```
  m_pMainWnd->UpdateWindow();
  return TRUE;
}

HelloWindow::HelloWindow()
{
  Create(NULL,
    "Hello World!",
    WS_OVERLAPPEDWINDOW|WS_HSCROLL,
    CRect(0,0,140,80));
  m_pHelloButton = new CButton();
  m_pHelloButton->Create("Hello World!",
      WS_CHILD|WS_VISIBLE,CRect(20,20,120,40),this,1);
}
```

Aren't you glad you saw that?

The `Application` manages the life-cycle of your app. The *primary stage* is the outer skin of the main window of your program. From this primary stage, you can create other stages.

The contents of the stage are kept in a `Scene`. The scene maintains the contents of the content pane. You also see a `Group`; this is the root of the *scene graph*, which allows us to create a hierarchy of elements placed by us in the scene. Let's look at this segment of code.

```
Button aButton = new Button("Click Me!");
Group root = new Group();
root.getChildren().addAll(aButton);
primaryStage.setScene(new Scene(root, 300, 250));
```

The first line creates a new button. The second is more mysterious. A group maintains a collection of children that are `Nodes` that go into the scene graph. So, we first create a group. We then add our button to its list of nodes using the group's `getChildren()` method. We finally set the scene (stuff in the content pane), specifying that the group `root` is to be placed in it and specifying dimensions for the content pane. The elements in the content pane are kept in the scene graph. The scene graph consists of a finite family of rooted trees.

Programming Exercises

1. Add two more buttons to the scene. Give them different labels. Run it and see what it looks like. Use the group's `addAll` method to add all three buttons at once.
2. Change the `Group` to an `HBox`. What does that do? Make sure you do the right import; look in the API guide.

3. Change the `Group` to a `VBox`. What does that do? Make sure you do the right import.

We just saw a practical example of inheritance at work; our application is an extension of `javafx.application.Application`. The classes `Group`, `HBox`, and `VBox` are all subclasses of the class `Parent`. The parent is a subclass of `Node`, which can be a widget itself or be a container that holds other widgets.

3 Nodes

The class `javafx.scene.Node` is the root class for all nodes in the scene graph. The scene graph in a `Stage` consists of several rooted trees; each is a different scene. There are several types of nodes in these trees. You can swap scenes in and out of a `Stage` using the method `setScene`.

1. **The Root Node** Every tree in the scene graph will have exactly one node without a parent; this node is called the *root node*.
2. **Parent Nodes** These include the root node and the interior nodes in the tree which can have children. The class `javafx.scene.Parent` is abstract. The `Group` class is a concrete implementation of `Parent`. All of the layout managers are subclasses of `Pane`, which is also a subclass of `Parent`.
3. **Non-Parent Nodes** These are nodes that cannot have children. Examples of these include such things as text boxes, graphical shapes, and image and media views.

Programming Exercises

1. Can you create an instance of a `Node`? Explain why or why not.
2. Can a variable of `Node` type point at at `Button`?
3. Look up the class `Rectangle`. Can you cause a `Rectangle` to appear on the screen?

4 The Full Skinny on Application Life Cycle

Here is a general application and exactly what happens when you run it.

```
import import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
public class Example extends Application
```

```
{
    public Example()
    {
    }
    @Override
    public void init()
    {
    }
    @Override
    public void start(Stage primary)
    {
    }
    @Override
    public void stop()
    {
    }
    public static void main(String[] args)
    {
        launch(args);
    }
```

In general when you create an application, the following occur.

1. You enter `java Example` at the command line.
2. The main method of the application runs, causing the application's static `launch` method to run.
3. The constructor executes.
4. The `init()` method executes. By default this does nothing.
5. The `init()` method returns and then `start()` starts to run.
6. The `start` method is actually a loop. This loop terminates if the user quits the program, if exception or memory error kills it, or if the `start` method returns.
7. If the event that quits the program calls `Platform.exit()` or if the go-away button is clicked, `stop()` is called. You can use this to save or close opened files, or any other housekeeping that should be done prior to shutting down. The `stop()` method by default does nothing.
8. The `stop()` method returns and the program's process is terminated. The kernel reclaims the program's memory and execution ends.

5 Interfaces

Now we dive back into the world of straight Java programming. We need to know: *How does this business with event handling actually work?*

The mechanism is built on *interfaces*, which are contracts stipulating methods that must be implemented by classes. You can think of an interface as being a "super-abstract" class, as we shall see soon.

Let us go back to the suite of `Shape` classes we created earlier. We blew our inheritance in the `Shape` class example. The big advantage yielded there was that a `Shape` variable could point at a `Rectangle`, `Circle`, or a `Square`. We could see obtain the diameter, perimeter or area of any such shape.

We got rid of the useless code in the `Shape` class by declaring it `abstract`. Since you cannot compute geometric quantities of a shape without first knowing what kind of shape it actually is you should make the `Shape` class abstract.

Let us now explore this new programming construct, the `interface`. We shall create an interface called `IShape` for handling shapes.

We decided that the essence of being a shape here is knowledge of your diameter, perimeter and area. Save this in a file named `IShape.java`.

```
public interface IShape
{
    public double area();
    public double perimeter();
    public double diameter();
}
```

What you see inside of the `IShape` interface is disembodied method headers; these are just abstract methods. Since you are programming in an interface, it is unnecessary to mark them, or the interface for that matter, `abstract`.

For now, you are not allowed to have any code inside of an interface. You may only place abstract method headers in it. An interface is just a named list of abstract method headers.

Java 8 Note Java 8 allows for the creation of interfaces whose methods have default implementations, but we will not address this at this early stage of the game. Many object-oriented programmers view this development askance and say it is only used for the sake of backward compatibility.

This is a debate that will shake out over time, but we do not need to worry about it for now. The principal benefit of these methods is to avoid breaking Java's backward compatibility.

An interface is an offer to sign a contract in Java. You know, for instance,

that a `Rectangle` should be a `IShape`. To sign the contract, modify the class header header to read

```
public class Rectangle implements IShape
```

You will see that, when you type the word `implements` into your text editor, it turns blue. (Note: forgetting the 's' on implements is a common error.) This indicates that `implements` is a language keyword. By saying you implement an interface, you warrant that your class will implement all methods specified in the interface, unless it is abstract and it passes this job off to its children. This contract is enforced by the compiler.

If you are creating a child class, you can use methods from ancestor classes to satisfy the requirements of implementing an interface. This rule is entirely consistent with the rule for abstract methods in abstract classes, so there is no surprise here.

An example of this construct from the standard libraries is the `Runnable` interface; this has only one method: `public void run()`. Look it up in the API guide; it lives in the package `java.lang`. A more complex example is that of the interface `List` in `java.util`. This interface has quite a few methods; note their names and look at the classes `ArrayList` and `LinkedList`; both of these classes implement `List`.

Interfaces are not classes. Because they contain abstract methods, you may not create an instance of an interface using the `new` keyword. This would make absolutely no sense, because none of an interface's methods have any code.

Because of the visibility principle, you *can* create variables of interface type. Such variables may point at any instance of any class implementing that interface. This works because the method's type is specified by its method header. It is the actual object that contains the code which executes. You can also use interfaces in signatures of methods as types for arguments. Objects passed to these arguments must implement the specified interface. This rule is enforced by the compiler.

Go back to the classes we created earlier that descended from `Shape`. Modify them to implement `IShape` instead, and polymorphism will work perfectly! Here is a driver program.

```
public class IShapeDriver
{
    public static void main(String[] args)
    {
        IShape s = new Rectangle(6,8);
        System.out.println("6X8 rectangle diameter = "
            + s.diameter());
        s = new Square(10);
```

```
        System.out.println("10X10 square area = " + s.area());
        s = new Circle(5);
        System.out.println("circle of radius 5 perimeter = "
            + s.perimeter());
    }
}
```

Run it and get this output.

```
> java IShapeDriver
6X8 rectangle diameter = 10.0
10X10 square area = 100.0
circle of radius 5 perimeter = 31.41592653589793
```

The variable of interface type pointed at all of the different shapes and the desired results were achieved. Now append this line to the code

```
IShape s = new IShape();
```

and see the angry yellow.

```
1 error found:
File: /home/morrison/Java/IShapeDriver.java  [line: 11]
Error: /home/morrison/Java/IShapeDriver.java:11:
    IShape is abstract; cannot be instantiated
```

This is the compiler's diplomatic yellow reminder that you cannot create instances of interfaces. Note the compiler's use of the term "abstract" for a bodyless method header.

Is there a one-parent rule for interfaces? Happily, no. Why is this true? We learned that the deadly diamond is triggered by multiple inheritance. If a child has two parents with two different methods with the same name and signature, there is a conflict

No such conflict exists for interfaces because their methods are abstract! There is no code to conflict. So if you have a class C that you want to have implement interfaces X, Y, and Z, you simply do this.

```
public class C implements X, Y, Z
{
    //code
}
```

This comma-separated list can have as many interfaces as you wish. Your class must have all of the methods specified by the interfaces you are implementing.

What if two interfaces have a method in common? It just needs to be present; you are simply killing two birds with one stone by having it present in your class.

If you are implementing interfaces, you can still extend a class. The usage looks like this

```
public class Child extends Parent implements X, Y, Z
{
    //code
}
```

5.1 A Java 8 Note: More about Default Methods

Some interfaces have default methods. This can cause the deadly diamond to rear its ugly head. However, if you implement several interfaces with conflicting default methods, you *must* override those default methods, or the compiler will issue forth with a nastygram, and compilation will halt.

6 Extending Interfaces

We have learned that we can extend classes; via this mechanism we can create a new class with all of the public interface of another class, then add new methods to it, or overriding methods in it to meet our needs.

Extending interfaces is simpler. The child interface inherits all of the methods specified in the parent; then you can add new methods. For example, in the `Shape` classes, we had the interface

```
public class IShape
{
    public double diameter();
    public double perimeter();
    public double area();
}
```

We could create a new interface `IPolygon` as follows.

```
public interface IPolygon extends IShape
{
    public int numSides();
}
```

We now revise our `Rectangle` class by implementing `IPolygon` and by adding this method.

```
public int numSides()
{
    return 4;
}
```

Note that `Square` will inherit the `IPolygon` interface; you need not change it at all. Note that `Circle` should only implement `IShape`.

Extending interfaces just adds new methods to an interface.

A Design Tip If all of the methods of an abstract class are abstract, make it an interface.

6.1 The API Guide, Again

Let us take a look at the `ArrayList` class. Right under the family tree you see an area listing all implemented interfaces. Here is a list.

1. Serializable
2. Cloneable
3. Iterable<E>
4. Collection<E>
5. List<E>
6. RandomAccess

Now visit the API pages. The interface `Serializable` is an interface with no methods. It is a *bundler* or *tagging* interface. You will meet it later when we store objects in files.

The `Iterable<E>` interface features one abstract method: `iterator` that returns an object called an *iterator*. It also had two mysterious default methods called `spliterator` and `forEach`. You will notice the `ArrayList<E>` class has a method called `Iterator`.

You will see that the interface `List<E>` is quite a complex interface containing many methods.

Each API page discloses which interfaces are implemented by a given class.

Let us take a trip to the `List<E>` interface. Look for the heading **All Known Implementing Classes**. Quite a number of classes implement this interface including

1. `AbstractList<E>`
2. `ArrayList<E>`
3. `LinkedList<E>`
4. `Stack<E>`
5. `Vector<E>`

All of these classes have something in common: they represent ordered sequences that you can index into using the method `get`.

7 Understanding Functional Interfaces

This is the key to understanding how event handling is done in `JavaFX`. To lead up to this, let us study a non-GUI example that is quite useful.

We now undertake a brief study of the interface `Comparator<E>`. Here is its code.

```
public interface Comparator<E>
{
    public int compare(E e1, E e2);
}
```

The design contract works like this. The set of elements `E` is given an ordering $\leq$ by this `compare` method. This ordering works as follows.

$$\texttt{e1} \leq \texttt{e2} \iff \texttt{compare}(\texttt{e1}, \texttt{e2}) \leq 0.$$

This ordering should satisfy these properties. For all `e1`, `e2` and `e3`,

1. If `e1`.equals(`e2`), then `compare`(`e1`, `e2`) returns 0.
2. if $\texttt{e1} \leq \texttt{e2}$ and $\texttt{e2} \leq \texttt{e3}$, $\texttt{e1} \leq \texttt{e3}$ (transitivity).
3. $(\texttt{e1} \leq \texttt{e2}) \vee (\texttt{e1} \leq \texttt{e2})$ is always true. In other words, any two elements of `E` are "related." Mathematically, this ordering is said to be *linear*.

We will do a modest example here, which will allow us to sort an entry from a concordance by frequency of appearance and secondarily alphabetically, or alphabetically and then by frequency. We begin by showing the basic class.

```
public class ConcordanceEntry
{
    String word;
    int times;
```

```
    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

Next, we add some two static elements of type `Comparator`; notice that these variable are of interface type. We have made them `final` because they are exposed to the client.

```
public class ConcordanceEntry
{
    public static final Comparator<ConcordanceEntry> byTimes;
    public static final Comparator<ConcordanceEntry> byFrequency
    static
    {
        byTimes = (e1, e2) -> {
            if(e1.times != e2.times)
            {
                return e1.times - e2.times;
            }
            return e1.word.compareTo(e2.word);
        };
        byAlpha = (e1, e2) -> {
            if(!e1.word.equals(e2.times))
            {
                return e1.word.compareTo(e2.word);
            }
            return e1.times - e2.times;
        };
    }
    String word;
    int times;

    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
```

```
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

What is happening here? How are these two new static members actual objects?

If you visit the API guide, you will see that the interface `Comparator` is a functional interface. It specifies one method,

```
public int compare(T e1, T e2);
```

Java performs a feat of type inference here. It creates an object of class type that implements the specified method and it assigns the lambdas we assigned to the static variables as the `compare` method of that object. No specific named class is ever created.

In pre-8 Java, you would need to implement these using an anonymous inner class. The code for byAlpha would look like this.

```
        byAlpha = new Comparator<ConcordanceEntry>(){
            public int compareTo(Concordance e1,
                ConcordanceEntry e2){
            if(!e1.times.equals(e2.times))
            {
                return e1.word.compareTo(e2);
            }
            return e1.times - e2.times;
        };
```

This new construct of lambdas eliminates a good bit of boilerplate code that adds no meaning to what we are doing. What we want to pass here is *behavior* that is to be carried out by our comparator objects.

So what is the benefit of this? How do we sort? Let us demonstrate that in a main method. To implement this method, we introduce the static method `sort` in the class `java.util.Collections`. There are two methods by this name. We use the `sort` method which has the signature `[List<E>, Comparator<E>]`; this method sorts according to our comparator.

```
    public static void main(String[] args)
    {
        ArrayList<ConcordanceEntry> al =
            new ArrayList<>();
```

```
        al.add(new ConcordanceEntry("cow", 5));
        al.add(new ConcordanceEntry("pig", 2));
        al.add(new ConcordanceEntry("zebra", 3));
        al.add(new ConcordanceEntry("cow", 5));
        al.add(new ConcordanceEntry("zebra", 5));
        al.add(new ConcordanceEntry("elephant", 6));
        al.add(new ConcordanceEntry("eland", 1));
        al.add(new ConcordanceEntry("coati", 2));
        System.out.println("Unsorted:");
        for(ConcordanceEntry e: al)
        {
            System.out.println(e);
        }
        Collections.sort(al, byAlpha);
        for(ConcordanceEntry e: al)
        {
            System.out.println(e);
        }
        System.out.println("Sorted by word:");
        Collections.sort(al, byTimes);
        for(ConcordanceEntry e: al)
        {
            System.out.println(e);
        }
        System.out.println("Sorted by word:");
        for(ConcordanceEntry e: al)
        {
            System.out.println(e);
        }
    }
```

Running this, here is the original list.

```
cow:  5
pig:  2
zebra:  3
cow:  5
zebra:  5
elephant:  6
eland:  1
coati:  2
```

Here it is, sorted first by word and second by frequency.

```
Sorted by word:
```

```
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

Now we sort first by frequency then by word.

```
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

Types and Subtypes We establish some terminology that will be quite useful and natural to us as we move along. Suppose that `S` and `T` represent classes (abstract or concrete) or interfaces. We say `T` is a *subtype* of `S` or that `S` is a *supertype* of `T`, if one of these applies.

- `S` and `T` are the same type.
- `S` and `T` are interfaces and `T` extends `S`.
- `S` and `T` are classes and `T` extends `S`.
- `S` is an interface,`T` is a class, and `T` implements `S`.

The totality of types is just the totality of interfaces and classes. This puts a partial ordering on the set of all types. Every class type is a subtype of `Object`. Note that there is no root interface type.

Variables can be of any type. Objects can only be of concrete class types. Variables can point at any object that is a subtype of the variable's type. This will provide us a compact language for expressing relationships between interface, class, and abstract class types.

You can use any type for a type parameter in a generic class such as `ArrayList`. You can also use any type for an argument in a method. You can pass objects to such a method that are subtypes of that parameter's type.

7.1 A Useful Interface for GUIs

An interface we shall soon use is `EventHandler`, which lives in package `javafx.event`. Click on it to view its documentation. You see at the top of the API page that it says

```
@FunctionlInterface
public interface EventHandler<T extends Event>
extends EventListener
```

If you look in the family tree for the `EventListner<T>` interface, you will see that it extends the interface `EventListener`. If you look in the documentation of this interface, `EventListener` contains no required methods. It is another example of a tagging interface that ties together related classes. All event listener interfaces we create must extend this interface.

Now you must be curious about the `FunctionalInterface` annotation. Quite a few interfaces in Java specify just a single method; we just saw that the `Comparator<T>` is an example of this. We also have the `Runnable` interface which requires `public void run()`, and the `EventListener<T>` interface requires `public void handle(T event)`, where the `T` stands for the event type.

These are examples of what are called *functional interfaces*. A functional interface must specify *exactly one* abstract (bodyless) method. As we shall see soon, functional interfaces have some very nice properties in Java, which can shorten and simplify your code. We will make use of these properties for handling simple action events.

Dangerous Bend Suppose interface B extends interface A. If interface A has one abstract method and interface B adds a new method, then interface B *cannot* be a functional interface, since B will, in fact, specify two methods. Some interfaces have no methods; for example the interfaces `java.io.Serializable` and `java.awt.event.EventListener` have no methods. Such an interface cannot be a functional interfaces.

You can make your own functional interfaces. If you do so, use the `@FunctionalInterface` annotation. It tells the compiler you intend to make a functional interface and flags an error if you fall victim to the dangerous bend we just described. It operates in a manner entirely similar to that of `@Override`.

Here we show a simple example

```
@FunctionalInterface
public interface RealFunction
{
    public double compute(double x);
}
```

For your edification, we show an example where the compiler flags an error.

```
@FunctionalInterface
public interface Clunker extends RealFunction
{
    public int clunk();
}
```

The interface `RealFunction` already specifies one method, `public double compute(double x)`. So this interface actually specifies two methods, `compute` and `clunk()`. We compile and get this friendly message.

```
$ javac Clunker.java
Clunker.java:3: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Clunker is not a functional interface
      multiple non-overriding abstract methods found in interface Clunker
      1 error
$
```

Remember, is is a far, far better thing that the compiler flag errors than you deal with them in a messy runtime situation. Use this annotation for your protection and to make your intent explicit.

Programming Exercises

1. Make your class `Triangle`, implement `IShape`.
2. Extend the interface `IShape` to a new interface `IPolygon`, which has an additional method

   ```
   public int numberOfSides();
   ```

   Decide which shapes should implement *IPolygon* and make the appropriate changes. The `extends` keyword is used for making child interfaces, just as it is used for making child classes.
3. Look in the package `java.util`. What interfaces in this package are functional interfaces?
4. Declare and `ArrayList<IShape>` and add shapes to it. Implement this method.

   ```
   public static double totalArea(ArrayList<IShape> al)
   {
       return 0;
   }
   ```

   Create an array list of `IShapes` and test it out.

8 Functional Interfaces and Event Handling

We now take a second look at one of our prior programs which we reproduce here.

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

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

    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Hello World!");
        Button aButton = new Button("Click Me!");
        Group root = new Group();
        aButton.setOnAction(e -> System.out.println("I've been clicked"));
        root.getChildren().addAll(aButton);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Let us step through the code. The object `primaryStage` is the captain window of your application. It is possible for it to spawn other stages. You begin by setting the window's title via the call to `setTitle()`. We next create a button named `aButton` with the text `"Click Me!"` emblazoned on it.

Next, we create a root for the first (and only tree) in the scene graph. Following this, we add the button to the scene graph.

A new `Scene` is then installed in the `primaryStage`, and we show it.

No, you are not crazy. We skipped this line. It's the one we do not understand. Let us now go on a treasure hunt and figure out how it works.

```
aButton.setOnAction(e -> System.out.println("I've been clicked"));
```

Begin by going to the API page for `Button` and looking up `setOnAction`. What

we see is that this method is inherited from the class `ButtonBase`. Here is its API method detail.

`setOnAction`

`public final void setOnAction(EventHandler<ActionEvent> value)`

Sets the value of the property onAction.

Property description:
`The button's action, which is invoked whenever the button is fired.`
`This may be due to the user clicking on the button with the mouse,`
`or by a touch event, or by a key press, or if the developer programmatically`
`invokes the fire() method.`

What we deduce here is that `setOnAction` takes an object of type `EventHandler<ActionEvent>` as its argument. Now it's time to visit this class. It specifies exactly one abstract method, `handle`.

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

What the heck is <T extends Event>?! The `EventHandler` class is a generic class. What you see here is a *type bound*; the type parameter we pass must be subtype of `Event`.

What kind of classes extend `Event`? Let us go to that page. There is a fair-sized list of them, but one of them is an `ActionEvent`. This is the type of event that is fired when a button is pushed. If an event handler is attached to the button via `setOnAction`, the code in its `handle` method is executed.

An object of type `EventHandler<ActionEvent>` is an event handler for an action event. It must have one public method,

```
public void handle(ActionEvent event)
{
    //code for the event
}
```

You see that Java is performing type inference in a manner identical to that of the example we did with the interface `Comparator` in the class `ConcordanceEntry`.

You can see that the secret to all of this working lies in two places, lambdas and type inference. We next turn to a discussion of lambdas in a little more detail than we have been using.

9 Lambdas

Let us make clear all the features of lambdas, which are anonymous functions. Recall that Python has lambdas. A typical Python lambda looks like this

```
lambda x : x*x
```

Lambdas, in both Python and Java, are function literals. The Python lambda depicted here is the squaring function. Python lambdas are assignable. You can do this

```
f = lambda x: x*x
print f(5)
```

and the output `25` will be put to `stdout`. You can create lambdas like these

```
lambda : print("foo")    #this has an empty sig and prints "foo"
                         #Tacit return of None.
lambda x,y,z: x*y*z      #this returns the product of three numbers
```

In Python, lambdas are primarily meant for short functions. If you are doing something complex, you really need to fall back to the `def` mechanism. Lambdas in Python have a tacit return.

In Java, it's a little different. The simplest Java lambdas look like this

```
e -> System.out.println("foo")  //same as lambda e : print("foo")
() -> System.out.println("foo") //same as lambda (): print("foo")
(x,y,z) -> x*y*z                //same as lambda x,y,z : x*y*z
```

You can also specify types in the signature. For example, you can make a lambda like this.

```
(int n, String s) -> n*s.length()
(int x, int y, int z) -> x*y*z
```

In all of these one-line lambdas, there is a tacit return, just as there is in Python. Note that `e -> System.out.println("foo")` has `void` return type.

In Java, if your lambda has more than one line in its body, you can format it like so.

```
e ->
{
```

```
        //line 1
        //line 2
        //line 3
    }
```

In this case, there is no tacit return. You must put in a `return` statement if you wish to return a value from a lambda whose body is enclosed in curly braces.

Consider this example

```
x -> x*x;
```

This accepts a numerical argument and it returns the square of the number passed it. This lambda has a tacit return, which is the expression to the right of the arrow. It is best to use this only for very simple functions.

Now we put our little lambda to work. The `apply` method causes the function to be evaluated. Make sure you put the `.0` on your number or the compiler will whine.

```
import java.util.function.Function;
public class Lambdas
{
    public static void main(String[] args)
    {
        Function<Double, Double> f = x -> x*x;
        System.out.println(f.apply(5.0));
    }
}
```

Run this and `25.0` is put to `stdout`. Do not put a `return` statement in a simple lambda such as this or you will get this utterly useless error message.

```
2 errors found:
File: /Users/morrison/book/java/javaCode/j5FX/Lambdas.java  [line: 6]
Error: illegal start of expression
File: /Users/morrison/book/java/javaCode/j5FX/Lambdas.java  [line: 6]
Error: not a statement
```

Now wasn't that informative?

Also note you have met a new interface

```
java.util.function.Function
```

It has one specified method and looks like this

```
@FunctionalInterface
public interface Function<S, T>
{
    public T apply(S arg);
}
```

Here we show a lambda that spans several lines. In such a function, you can declare local variables and call other functions. If you wish to return something, remember you must explicitly use a `return` statement.

```
import java.util.function.Function;
public class Lambdas
{
    public static void main(String[] args)
    {
        Function<Double, Double> f = x -> x*x;
        System.out.println(f.apply(5.0));
        Function<String, Integer> foo = s  -> {
            int q = s.length();
            return q*q*q;
        };
        System.out.println(foo.apply("flibbertygibbet"));
    }
}
```

The output is unsurprising.

```
25.0
3375
```

You can restrict types of arguments in a lambda. We do this in our little program. Try changing the lambda `foo` as follows. Note the use of parentheses.

```
Function<String, Integer> foo = (String s)  -> {
    int q = s.length();
    return q*q*q;
};
```

Note that a lambda can have several arguments; just separate them with commas in a list as we did in our example with comparators. You can restrict types for all of the arguments if you wish.

We do a little example here to repeat a string. You will now meet a new interface, `BiFunction`.

```
@FunctionalInterface
public interface BiFunction<S, T, U>
{
    public U apply(S arg1, T arg2);
}
```

Here is our full program.

```
import java.util.function.Function;
import java.util.function.BiFunction;
public class Lambdas
{
    public static void main(String[] args)
    {
        Function<Double, Double> f = x -> x*x;
        System.out.println(f.apply(5.0));
        Function<String, Integer> foo = (String s)  -> {
            int q = s.length();
            return q*q*q;
        };
        System.out.println(foo.apply("flibbertygibbet"));
        BiFunction<String, Integer, String> repeat = (String s,  Integer n) ->
        {
            StringBuffer sb = new StringBuffer();
            for(int k = 0; k < n; k++)
            {
                sb.append(s);
            }
            return new String(sb);
        };
        System.out.println(repeat.apply("*", 50));

    }
}
```

Running this produces the following output.

```
25.0
3375
**************************************************
```

Programming Exercises

1. Look up the interface `BinaryOperator`. Use it to make a lambda that multiplies two integers.

10 And Now Back to Event Handling

The central interface to handling events in JavaFX is the interface

`javafx.event.EventHandler`

It looks like this

```
@FunctionalInterface
public interface EventHandler<T>
{
    public void handle(T event);
}
```

The type parameter is the event type. We have met one event type in creating our live button, `ActionListener`. The button's `setOnAction` method expects an object of type `EventHandler<ActionListener>`. Consider this code fragment.

```
Button b = new Button("Click Now");
b.setOnAction( e -> System.out.println("Bah, spam!");
```

Java's type inference says that the lambda is an object of type `EventHandler<ActionListener>`, and that is why the button responds when clicked.

11 Fun with Mohammed Ali: How to Achieve the Desired Layout

The package `javafx.scene.layout` contains classes that control the arrangements of widgets in the content pane or a portion thereof. We have met two already in the exercises, the `VBox` which puts things in a vertical column and `HBox` which arranges them in a horizontal row.

We are now going to meet some of their brethren which will give us a decent palette for creating our own layouts. These classes all descend from `javafx.scene.layout.Pane`. If you go into the JavaFX API guide, you will see these direct descendants of `Pane`. All reside in package `javafx.scene.layout`.

1. AnchorPane
2. BorderPane
3. DialogPane

4. FlowPane
5. HBox
6. PopControl.CSSBridge
7. StackPane
8. TextFlow
9. TilePane
10. VBox

We begin with the border pane. We start by creating a minimal javafx application.

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

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        Scene s = new Scene(bp, 500, 500);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Next, we make a `Scene` with a `BorderPane` it and embed it in our `Stage`.

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

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
```

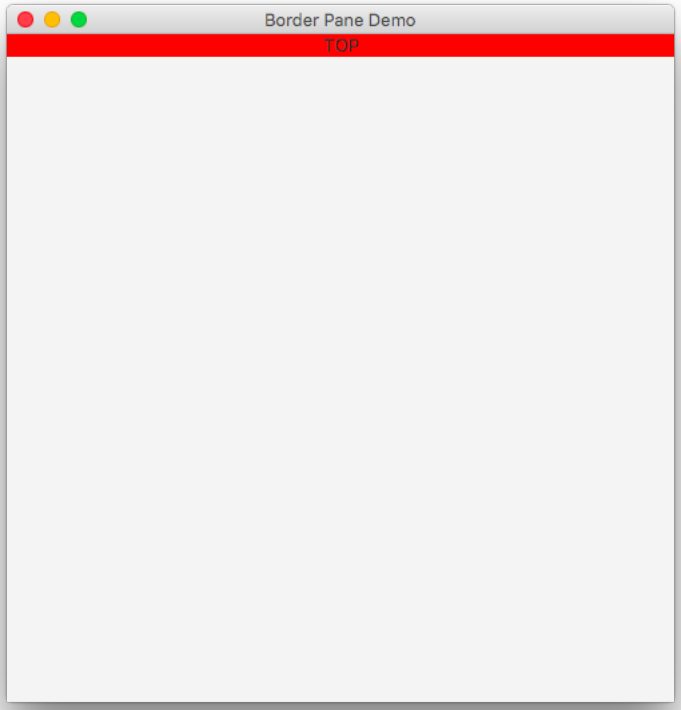
```
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Now we go to work on the top. We make an `HBox` and put a `Label` in it marked `TOP`. We add the label to the `HBox`. We set the `HBox`'s alignment to center, and set its background color to red.

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

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);

        Label topLabel = new Label("TOP");
        HBox topBox = new HBox();
        topBox.setAlignment(Pos.CENTER);
        topBox.getChildren().add(topLabel);
        topBox.setStyle("-fx-background-color:red");
        bp.setTop(topBox);


        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
```

```
    {
        launch(args);
    }
}
```

The result can be seen here.



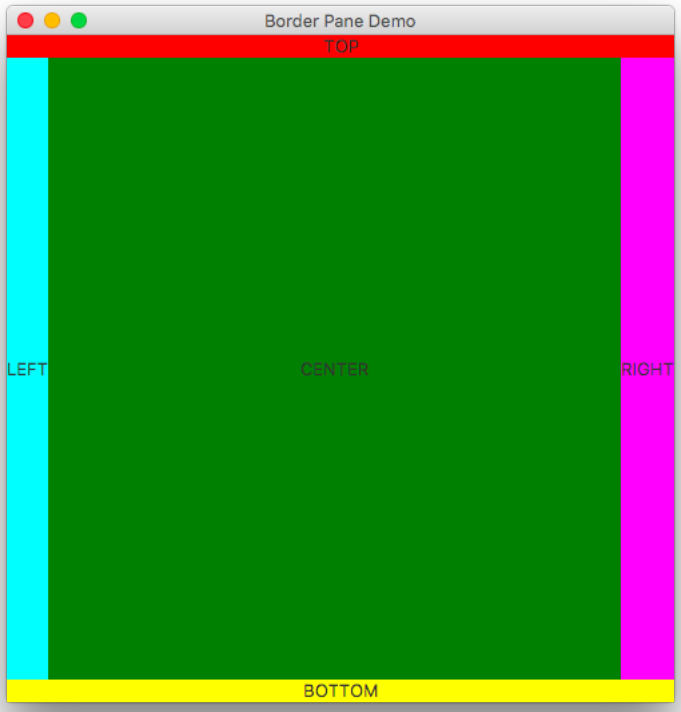
The red field shows the size of the top border pane. Now we add in the rest of the panes, giving them backgrounds of various colors.

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

```
public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);

        Label topLabel = new Label("TOP");
        HBox topBox = new HBox();
        topBox.setAlignment(Pos.CENTER);
        topBox.getChildren().add(topLabel);
        topBox.setStyle("-fx-background-color:red");
        bp.setTop(topBox);

        Label bottomLabel = new Label("BOTTOM");
        HBox bottomBox = new HBox();
        bottomBox.setAlignment(Pos.CENTER);
        bottomBox.getChildren().add(bottomLabel);
        bottomBox.setStyle("-fx-background-color:yellow");
        bp.setBottom(bottomBox);

        Label leftLabel = new Label("LEFT");
        HBox leftBox = new HBox();
        leftBox.setAlignment(Pos.CENTER);
        leftBox.getChildren().add(leftLabel);
        leftBox.setStyle("-fx-background-color:cyan");
        bp.setLeft(leftBox);

        Label rightLabel = new Label("RIGHT");
        HBox rightBox = new HBox();
        rightBox.setAlignment(Pos.CENTER);
        rightBox.getChildren().add(rightLabel);
        rightBox.setStyle("-fx-background-color:magenta");
        bp.setRight(rightBox);

        Label centerLabel = new Label("CENTER");
        HBox centerBox = new HBox();
        centerBox.setAlignment(Pos.CENTER);
        centerBox.getChildren().add(centerLabel);
        centerBox.setStyle("-fx-background-color:green");
        bp.setCenter(centerBox);

        primary.setScene(s);
```

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

Run this and see the result. You now know what a `BorderPane` looks like.

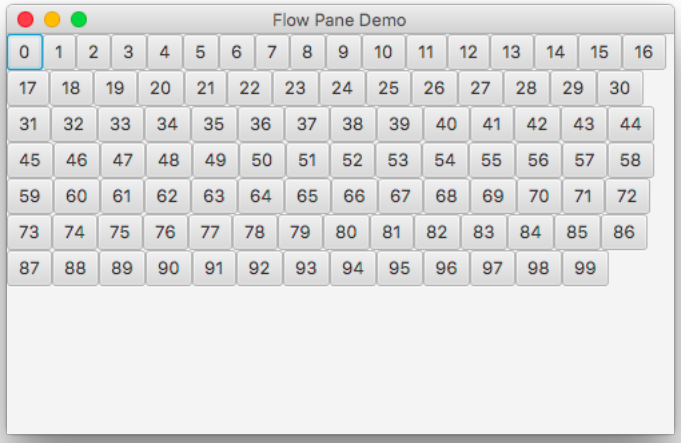


Now let us consider a `FlowPane`. This has a "jimmybuffetesque" layout policy. Here is a program that puts 100 buttons into `FlowPane`.

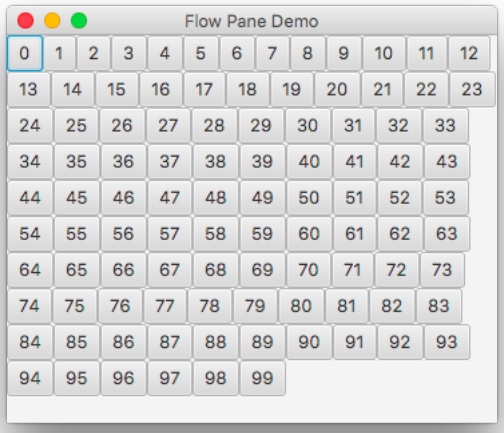
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.FlowPane;
```

```
import javafx.scene.control.Button;
public class Flow extends Application
{
@Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
                FlowPane fp = new FlowPane();
        Scene s = new Scene(fp, 500, 500);
                for(int k = 0; k < 100; k++)
                {
                        fp.getChildren().add(new Button("" + k));
                }
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Run the program and you will see this.



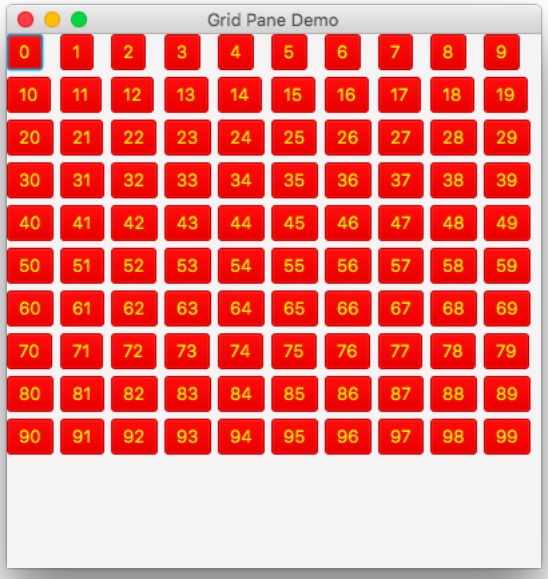
Now resize the window and watch the buttons re-corral themselves amicably. Everybody seems to get along.



Finally, we will do a sample program with a `GridPane`. This will put 100 buttons in a neat little 10×10 grid.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.GridPane;
import javafx.scene.control.Button;
import javafx.scene.paint.Color;
public class Grid extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Grid Pane Demo");
                GridPane gp = new GridPane();
        gp.setHgap(5);
        gp.setVgap(5);
                for(int k = 0; k < 10; k++)
                {
            for(int l = 0; l < 10; l++)
            {
                Button b = new Button("" + (10 *k + l));
                b.setStyle("-fx-base:red;-fx-text-fill:yellow");
                            gp.add(b, l, k);
            }
                }
```

```
        Scene s = new Scene(gp, 400, 400);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```



A Partial list of Layout Managers	
`Pane`	This just sizes all components to their preferred size. This is the parent class of all of the layout managers.
`StackPane`	This places it children in a back to front stack atop one another.
`FlowPane`	It enforces a "Jimmy Buffet" policy in which widgets go with the flow.
`BorderPane`	This has fields for TOP, LEFT, RIGHT, BOTTOM, and CENTER. The CENTER field is "piggy" and will devour the entire content pane if no other portions of the pane are occupied.
`VBox`	This positions its child widgets vertically.
`BBox`	This positions its child widgets horizontally.
`GridPane`	This positions its child widgets in a rectangular grid.

12 Examining Final

The keyword `final` pops up in some new contexts involving inheritance. Let us begin with a little sample code here

```
public class APString extends String
{
}
```

We compile this, expecting no trouble, and we get ugliness from the compiler in the form of this error message.

```
1 error found:
File: /home/morrison/book/texed/Java/APString.java  [line: 1]
Error: /home/morrison/book/texed/Java/APString.java:1:
    cannot inherit from final java.lang.String
```

The `String` class is a `final` class, and this means that you cannot extend it. Why do this? The creators of Java wanted the `String` class to be a standard. Hence they made it `final`, so that every organization under the sun does not decide that it would like to create (yet another annoying....) implementation of the `String` class. An example of this undesirable phenomenon existed during the days of the AP exam in C++. Subclasses of the string and vector classes were created for the the exam. Near the top of the API page for the `String` class, you will see it says

```
public final class String extends Object
```

Look here on any API page to see if a given class is final. Methods in classes can also be declared `final`, which prevents them from being overridden. We present a table with all of the uses of `final`, including a new context in which we mark the argument of a method `final`. Note the all of the wrapper classes are `final`.

Also, strings are pooled in Java. A descendant class of mutable strings could admit its instances to the pool. Via these instances and via the reflection library, it could be a huge security risk. Overridden methods in the child class could also perpetrate malicious activity.

`final` **Exam!**	
`primitive`	When a variable of primitive type is marked final, it is constant, since it cannot be assigned a new value.
`Object`	When a variable of object type is marked final, it can never point at an object other than the object with which it is initialized. Mutator methods, however can change the state of an object being pointed at by a final variable. What is immutable here is the pointing relationship between the identifier marked `final` and its object. Note that finality is a property of a variable, and not an object.
`class`	When a class is marked `final`, you cannot extend it.
`method`	When a method is marked `final`, you cannot override it in a descendant class.
`argument`	When an argument of a method is marked `final`, it is treated as final local variable within its function.

13 Terminology Roundup

We have blasted through a lot of important ideas here, so we will make a tidy list of all of the new terms we have encountered.

- **abstract class** An abstract class cannot be instantiated. Any class containing an abstract method must be marked `abstract`. However, any class can be marked `abstract` to prevent instances of it from ever being made.
- **abstract method** Abstract methods are bodyless method headers. They appear in both abstract classes and interfaces.
- **compositional relationship** This is a has-a relationship, the most common in object-oriented programming. One class uses instances of other classes for state. For example, a `BigFraction` has two `BigIntegers`. A second type of compositional relationship is a uses-a relationship; this occurs when an instance of a class is used as a local variable in a method.

- **delegation principle** This is the principle that says that an object is responsible for executing method code, regardless of the type of variable pointing to it. Just remember: *Objects know nothing about the variables pointing at them. They just execute their behavior.*
- **event dispatch thread** This is the thread through which all GUI events are passed and processed. It is a single-file thread that executes events in order in which they are received.
- **extends** If you have two classes A and B and if A extends B, then A inherits all of the public non-static methods of B. This is how we signify inheritance in Java.
- **final method** This is a method is a class that cannot be overridden by descendant classes.
- **final class** This is a class that cannot be extended.
- **final variable** This is a variable that, once initialized, cannot be reassigned. Note that a mutable object can have its state changed by a final variable. Note that finality is a property of variables, not objects.
- **functional interface** This is an interface specifying exactly one abstract method.
- **@FunctionalInterface** This annotation causes the compiler to check that an interface you are making is, in fact, functional. A compiler error will be generated if your interface is not functional.
- **inheritance** This is an is-a relationship between classes. If class A inherits from class B, then A is a B.
- **interface** An interface is a named list of abstract methods.
- **implements** This keyword indicates a class is signing the implementation contract of an interface.
- **lambda** A lambda is an anonymous function. It is not affiliated with any class. You can use functional interface types for variables that point at lambdas.
- **model-view-controller design pattern** This is the general anatomy of a GUI application. It consists of the view, which is the graphical visual portion, the model, which is the state embodying the business logic, and the controller, which consists of the event handlers.
- **node** This is an element in a scene graph.
- **override** When a child class re-implements a method of the parent class, this takes primacy and supersedes the parent method.
- `@Override` This is a request to the compiler to verify that we are overriding a method from an ancestor class correctly.
- **polymorphism** This refers to the ability of variables of interface type to point at any object whose class implements the variable's interface type, or the ability of variables of class type to point down the inheritance tree.

- **scene graph** This is a data structure holding the elements of a scene. It consists of a finite set of rooted trees.
- **sibling classes** Two classes are sibling if they have the same parent class.
- **subtype** `T` is a subtype of `S` if any of these hold.
  1. `S` and `T` are the same type.
  2. `S` and `T` are interfaces and `T` extends `S`.
  3. `S` and `T` are classes and `T` extends `S`.
  4. `S` is an interface `T` is a class and `T` implements `S`.
- **super** Used in the first line of a constructor, `super` can be used to call one of the parent constructors. It is a compiler error to use it in a constructor after the first line. You can also use `super` to call parent class methods. The usage is `super.`*method*. This will call the parent's overridden method.
- **supertype** `S` is a supertype of `T` if `T` is a subtype of `S`.
- **thread** A thread is an independent sub-process of your java code that has its own call stack.
- **type** This is the name of a class or an interface; the set of all types is the totality of all classes and interfaces.
- **visibility principle** This is the principle that says that a variable's type determines the methods that are visible to it.
- **widget** This is the general term for graphical items that appear on the screen. A top-level widget such as a `Stage` can contain an entire application. A container widget such as a `Scene` or a can have other widgets added to it.