# 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. We created the `BigFraction` class, which allows a client programmer using it to do exact, extended-precision rational arithmetic. Another programmer could use this class for his own fractional calculations.

So far, the relationship between classes has been *compositional*, or a "has-a relationship." For example our *BigFraction* class has two `BigInteger`s, representing the numerator and denominator of our fraction object. This 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 see how to tie related classes together by using *interfaces*.

One of the major goals of this portion of the book is to introduce the reader to the world of event-driven program in a graphical environment. To get started in this exploration, we will first create a modest GUI program that places a button in a window on your screen. We will discuss what is happening in some detail, so you will be able to see why inheritance and interfaces are important and how they can help you develop surprisingly sophisticated applications in a small program. By the end of this chapter, you will create a simple GUI program that reacts to user interaction.

## 1 A Prelude to Inheritance: A Short GUI Program

We shall do a little exploration the Java GUI classes and illustrate how inheritance affects GUI programs. Quickly, 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.

- `java.awt` This is the "old brain" of Java GUIs. It contains quite a few graphical widgets, and such things as color and graphics.
- `javax.swing` This is the "new brain" of Java GUIs. This includes a panoply of things we will press into service, including frames, which hold applications, menus, buttons, and slider bars. This is the home of many of Java's widgets.
- `java.awt.event` and `javax.swing.event` These 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 use the interactions pane to produce a program that makes a window and puts a button in the window. Start by entering this code. When you are done, you will see a window pop up on your screen. In the title bar, you will see "My First GUI." Notice that the window will not appear until you enter `f.setVisible(true)`.

```
> import javax.swing.JFrame;
> JFrame f = new JFrame("My First GUI");
> f.setSize(400,400);
> f.setVisible(true);
```

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.

```
#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?

Keep your DrJava session open; we will now add to it.

A `JFrame` is a container that holds other graphical elements that appear in an applications; you can think of it as outer skin for your app. A `JFrame` is a *container widget* because other widgets can reside inside it. It is also a *top-level* widget, because it can contain an entire application.

Now let us make a button.

```
> JButton b = new JButton("Panic");
```

We have made a button, but we have not yet placed it in the `JFrame`. The content of a frame lives, logically enough, in the frame's content pane. The `JFrame` class has a method called `getContentPane()`, which returns a pointer to the content pane of the frame, allowing you to add widgets to it. Let us now get the button in the window.

```
> f.getContentPane().add(b);
```

This is quite a hairy–looking construct, but if we dissect it, we will see it is very understandable. Look in the API guide. The call

```
f.getContentPane()
```

returns a pointer to the content pane of our `JFrame f`. The content pane is an instance of the class `Container`, which lives in package `java.awt`. You can see that because `Container` is the return type of `getContentPane()`. In the API guide, click on the return type link, `Container`. Go to `Container`'s method summary. The first method is

```
Component add(Component comp)
```

This is the method that is adding the `JButton` to the content pane. Now look at `JButton`'s API page. If you look up the family tree, you will see that, directly below `java.awt.Object`, there is `java.awt.Component`. What we learn here is that `JButton` *is a* `Component`. Thus, the `add` method in `Container` will happily accept a `JButton`.

Finally, we make the button appear in the window. The trick to doing this from the interactions pane is to set the frame to be invisible, then to be visible again. This will charm the button into appearing.

```
> f.setVisible(false);
> f.setVisible(true);
```

Your frame should have a big, fat button occupying the entire content pane. Click on the button. You will see it blinks in response to the click, but the button does not trigger any action in the program. This is no surprise, because we have just told the button to appear, not to do anything.

We just saw a practical example of inheritance at work; the `JButton` we added to the content pane is a `Component`, so we can add it to the content pane. There are others. Notice the call to `setSize` on the `JFrame`. If you look in the API guide, you will see that `JFrame` has no `setSize` in its method summary! This is inherited from the grandpappy class `Window`. The same is true of `setVisible`.

It seems here we are encountering a whole new layer of interest and complexity in Java. Now let us take a step back look at the idea of inheritance in general. Understanding inheritance is an absolute must in the world of Java GUI and event-driven programming. We will then return to GUIs and apply inheritance principles to them.

## 2 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 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.

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)
    {
        width = _width;
        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)
    {
        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 it, 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 $6 \times 8$ 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. 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 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
>
```

## 2.1 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 semicolons. This also forces any child class of `Shape` to implement those methods. 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.

```
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 spill angry yellow all over your attempt. You cannot make instances of an abstract class.

Here are a couple of things you should know. You can declare *any* class abstract. If you do, instances of it cannot be created. If you give any class an abstract method, you must declare the entire class abstract, and instances of it cannot be created.

## 2.2 Polymorphism, Delegation, and Visibility

How does this polymorphism thing work? We had the `Shape` variable pointing at a $12 \times 5$ 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.

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.
- **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.

## 2.3 Understanding More of the API Guide

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

```
java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame
```

The `JFrame` class in the `javax.swing` package extends the old `Frame` class in java.awt, the Abstract Window Toolkit package. We now now each class in the list above extends the one above it.

Notice 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 `JFrame`. Scroll down below the method summary. You will see links for all the methods inherited from `Frame`. 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 static field we will commonly use with the `JFrame` class is

```
JFrame.EXIT_ON_CLOSE
```

which we will use to tell an app to quit when its go–away box is clicked. Otherwise, your app remains running in memory; it just is not visible.

The quantity `JFrame.EXIT_ON_CLOSE` is actually an integer; type it into the interactions pane to see what integer it is.

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. 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 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.

## 2.4 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 method when overriding `Object`'s `toString()` and `equals(Object o)` methods. You will see it used throughout the rest of the book.

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

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.

## 2.6 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. We will introduce this when we start doing event handling, which makes widgets in a window perform actions when they are activated by the user. First, we will deal with the `final` keyword and then we will see how to position widgets in a window, before turning to interfaces.

## 3 Interfaces

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 a new type of programming construct, the `interface` construct. 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 `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.

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 DrJava, 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.

An example of this construct from the standard libraries is the 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`.

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 its 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 have arguments for methods of interface type and pass any object whose class implements that interface as an argument to the method.

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.

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

## 3.1 The API Guide, Again

The Java libraries contain an abundance of interfaces; these serve as a means for organizing classes. If you look in the class window, you can tell an item listed is an interface if it is italicized. An example of this, which we shall soon use is `ActionListener`. Click on it to view its documentation. It has a superinterface called `EventListener`. Interfaces can be extended in the same manner as classes. A child interface simply adds more method headers. A superinterface is a parent interface. `ActionListener` has a subinterface called `Action`. Note that interfaces can be exteneded in a manner entirely similar to that of classes. Extending an interface just creates a new interface with new methods not specified by they parent.

Next, you will see a list of all classes that implement `ActionListener`; it is quite large. `ActionListener` has one method,

```
public void actionPerformed(ActionEvent e);
```

so any implementing class must have a method with this header. Click on some of the implementing classes and hunt for their `actionPerformed` methods.

Consider its parent interface. It must have no methods! Let us explore `EventListener`. Indeed, it is devoid of methods. It is simply a "bundler" interface that ties a bunch of classes together with a common bond. There are several interfaces like this in the Java standard libraries.

Quite a few interfaces in Java specify just a single method. For example the `Runnable` interface requires `public void run()`, and the `ActionListener` interface requires `public void actionPerformed (ActionEvent e)`. 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 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 interfaces 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.

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.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
@FunctionalInterface
public interface Clunker extends ActionListener
{
    public int clunk();
}
```

The interface `ActionListener` already specfies one method, `public void actionPerformed(ActionEvent e)`. So this interface actually specifies two methods, `actionPerformed` 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. Create a new class `Triangle`, which implements `IShape`. Look up *Herron's formula* to find the area of a triangle from its three sides. Remember, the diameter of a shape is the greatest distance between any two points in the shape. This should make computing the diameter of a `Triangle` simple.
2. Create a new class `EquilateralTriangle`. From whom should it inherit?
3. 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.

4. Look in the package `java.util`. What interfaces in this package are functional interfaces?

## 4 A Framework for our GUI Programs

We are going to create a simple framework for creating all of our GUI classes. As you are about to see, this is necessary to keep your applications running sanely and cleanly.

What we have not discussed so far is that Java has a capability called *threading* built into it. A *thread* in Java is a sub-process launched within a Java program; you can have several threads running concurrently within any given program. Threads run almost like independent programs within your program's process; in fact, Linux regards them as full-blown processes in their own right. Having several threads running at once is called *multithreading.* Every thread has its own call stack which is independent of those in the other threads.

Whether you know it or not, all Java programs of any size at all are multi-threaded. The garbage collector runs it its own thread, monitoring your the heap section of your program's memory for orphaned objects and deallocating their memory after they get orphaned. The method `main` starts the *main thread*, so a Java program always has the garbage collector and main threads running concurrently.

When you interact with a modern GUI program, your communications to the computer come in the form of *events.* Events include such things as keystrokes, mouse clicks, and button pushes. These are all initiated by the user of the computer.

Java manages events with a data structure called a *queue.* Queues work just like lines in a cafeteria. You enter the line at the end, and are *enqueued* in the line. You go through the line to the *service end*, where you get what is needed (food and paying), at which time you are *dequeued*, and leave the queue, having got what you were seeking. This completely severs your relationship with the queue and as far as the queue is concerned, you are gone for good.

In Java there is a queue called the *event dispatch queue* or event queue for short. In a GUI, there is a separate thread for the event queue, called logically enough, the it event dispatch thread. As your program processes events, they are placed on a queue, which is a first in, first out data structure. The events go into the queue, wait to be processed, and then are processed and are removed from the queue. We want to ensure that the events from our GUI enter the event queue in an orderly fashion. If we do not ensure this, strange things can happen to your GUI that are patently undesirable. In particular, you want events to be processed in the order in which the user causes them to occur. This way, your program will not have sudden magical, nonsensical behavior. The event dispatch thread is a single-file line that executes the code for events in the order in which they are received.

First, you must `implements Runnable` as shown in the class `Pfooie.java` here.

```
public class Pfooie implements Runnable
{
//constructors and other methods and instance variables
    public void run()
    {
        //your run code.  Build your GUI here.
    }
    public static void main(String[] args)
    {
        Pfooie pf = new Pfooie(anyArgumentsYouNeedIfAny);
        javax.swing.SwingUtilities.invokeLater(pf);
    }
}
```

Secondly, other thing required is for you to have a `run` method that looks like this.

```
   public void run()
   {
   }
```

What is that ugly stuff in `main`? On the first line, you are making an instance of your class named `pf`. You then pass this instance to the static method

```
    javax.swing.SwingUtilities.invokeLater
```

This function runs your `run` method so that the event queue behaves itself. Later, when we discuss interfaces in full, you will see that "implements Runnable" is just a promise you will implement the method

```
public void run()
```

Below we furnish a quick summary of what to do.

1. Implement the Runnable interface.
2. Implement a `public void run()`, as is required by the Runnable interface. Use this method to run your GUI. If the GUI is large, you can call other methods from `run`. This method should orchestrate the entire activity of your GUI.

3. Run your GUI by using the static method invokeLater(Runnable r) in `main` as shown above. This method lives in the class `SwingUtilities`. Since you only use it once, just use the fully-qualified class name and no import statement is needed.

Your events will now join the event queue in an orderly fashion and they will execute in the order in which the user creates them. You will see this basic format used throughout this book.

**Programming Exercise** In the `run` method insert these lines.

```
JFrame f = new JFrame();
f.setSize(500,500);
f.setVisibile(true);
```

Compile and run. You should see a window appear on your screen.

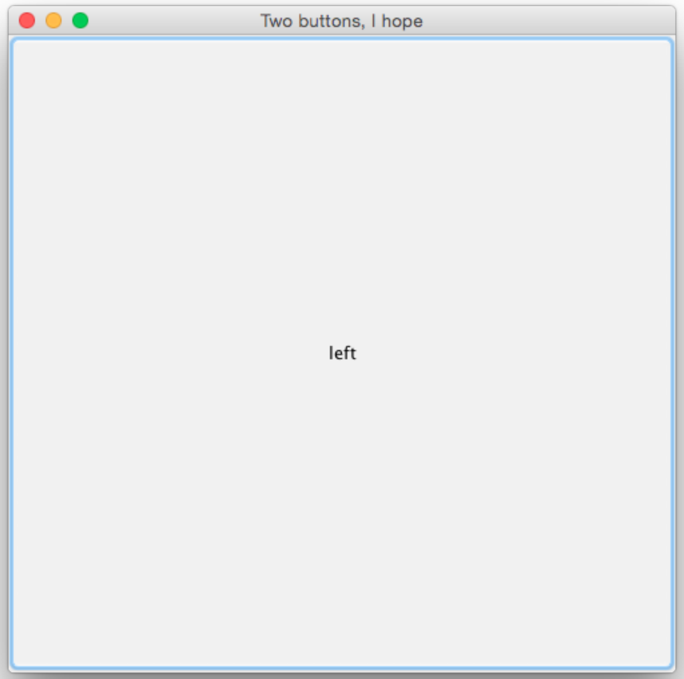
## 5 Fun with Mohammed Ali: How to Achieve the Desired Layout

Let us begin naïvely and wind up in a bind.

Open the interactions pane for an enlightening session in which we try to place two buttons in a window via its content pane. We begin by building the frame and adding the left button.

```
> import javax.swing.JFrame
> import javax.swing.JButton
> JFrame f = new JFrame("Two buttons, I hope");
> f.setSize(500,500);
> JButton left = new JButton("left");
> JButton right = new JButton("right");
> f.getContentPane().add(left);
> f.setVisible(true);
```

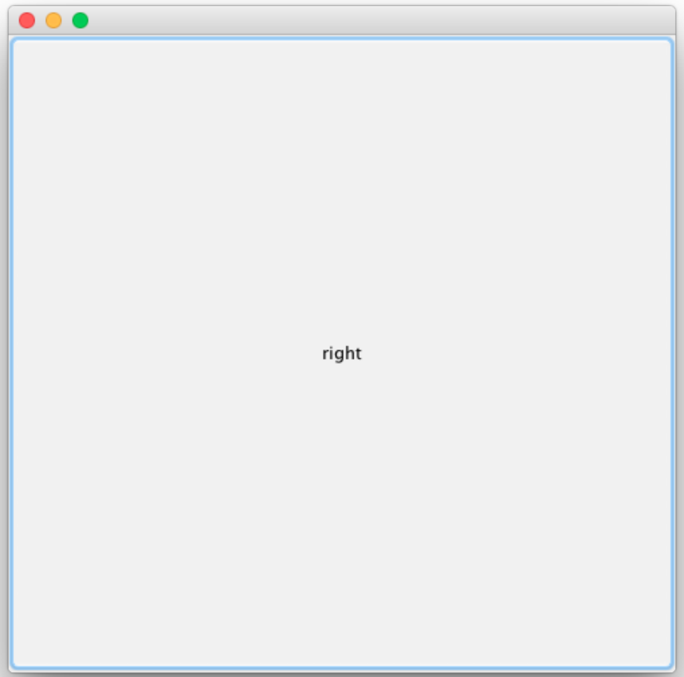
You should see a frame with the title “Two buttons, I hope” in the title bar. It features a button with `"left"` emblazoned on it. All is calm and irenic. Here it is.



Now let us try to add the right button. As a concession to reality, we know we have to make the frame invisible, add the button and make it visible again. We now do so.

```
> f.setVisible(false)
> f.getContentPane().add(right);
> f.setVisible(true);
```

Whoa! we only see one button in the window.



It reminds us of one of those nature shows where the biggest chick in the nest kills its siblings so it gets all of the food. This is a problem. How do we get these (childish) widgets to play nicely? To create a decent application, we will need to have several widgets occupying the content pane together.

**Quick, Call the cops, or at least attract the attention of Mrs. Wormwood and her lethal ruler!** The content pane appears to be some kind of Wild West scene, replete with fratricidal widgets. So the question is: How do we achieve the desired layout of components in the content pane? To get warmed up, try this little exercise. It will help you to get accustomed to our new GUI programming format and give you a sneak preview of things upcoming.

### Do-Now Programming Exercises

1. Make a copy of the GUI framework and call it `TwoButtons.java`. In addition to implementing `Runnable`, have it extend `JFrame`. Do so by making the class header read as follows.

```
public class TwoButtons extends JFrame implements Runnable
```

Enter the code you used in the interactions pane into the `run` method. There is an important difference: since we are extending `JFrame` we do not need to make a `JFrame`. Instead of saying `f.setSize(500,500)`, just say `setSize(500,500)`. You can get rid of the `f.`s. Add the two buttons to the content pane of the frame.

2. Add this line to your code before adding the buttons to the content pane.

   ```
   getContentPane().setLayout(new FlowLayout);
   ```

   Then import `java.awt.FlowLayout`. What happens? Can you add more buttons?

The tool we need to control the position of components in a container isis called a *layout manager.* A layout manager imposes a layout policy on a container, telling the widgets added to it how to fill the available space. This table provides a brief summary of the most commonly-used layout managers in Java. You are encouraged to forge ahead and perform experiments. The class `Container` has a method called `setLayout` that allows you to set the layout manger in that container.

| **Layout Managers** | |
|---|---|
| `null` | This is an absence of a layout manager. You manually position components by using `setLocation` and size them with `setSize`. |
| `GridLayout` | The constructor of the `GridLayout` accepts as its first argument a number of rows, then a number of columns. It places widgets in which it is the law of the land in a grid. |
| `FlowLayout` | It enforces a "Jimmy Buffet" policy in which widgets go with the flow. It has several constructors that give it additional guidance. In a `JPanel`, this is the default layout manager. |
| `BorderLayout` | This has fields for NORTH, SOUTH, EAST, WEST and CENTER. The CENTER field is "piggy" and will devour the entire content pane. The other fields occupy the edges of the container. This is the default layout in a `Container`. If you simply add something to a container, by default it adds to CENTER, which hogs all the space. |
| `BoxLayout` | This positions widgets vertically in a Jimmy Buffetesque fashion. |

There is one other layout manager, called a `GridBagLayout`, which gives precise control over the placement of widgets. This, however, is usually used by front-end programs such as NetBeans, that generate GUIs. You do not want to work manually with these.

**So, why did only one button show up?** The content pane of a `JFrame` is a `Container`, which by default, comes with the layout manager `BorderLayout`. By default, if you add a widget it is added to the center of the container. If you add one thing to the center, then another, then the second thing will cover the first.

Let us make a simple example that shows a how to use the border layout correctly. Begin by making this shell code.

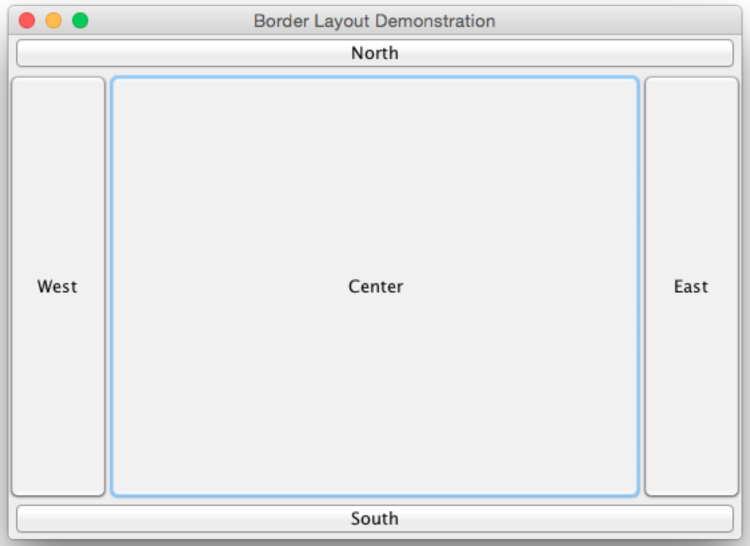
```
import javax.swing.JFrame;
import java.awt.Container;
import java.awt.BorderLayout;
public class BorderLayoutDemo extends JFrame implements Runnable
{
    public void run()
    {
        setSize(550,400);
        setTitle("Border Layout Demonstration");
        setVisible(true);
    }
    public static void main(String[] args)
    {
        BorderLayoutDemo bld = new BorderLayoutDemo();
        javax.swing.SwingUtilities.invokeLater(bld);
    }
}
```

Now compile and run; a JFrame should appear on the screen entitled "Border Layout Demonstration." Next add five JButtons to the content pane using the static `BorderLayout` constants as follows.

Begin by importing `JButton` at the top of your class file. Then place these lines in the `run` method just before the `setVisible` line.

```
    Container c = getContentPane();
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton center = new JButton("Center");
    c.add(BorderLayout.NORTH, north);
    c.add(BorderLayout.SOUTH, south);
    c.add(BorderLayout.EAST, east);
    c.add(BorderLayout.WEST, west);
    c.add(BorderLayout.CENTER, center);
```

You will now see this. The exact appearance of your window will vary, depending on your operating system. This one was generated on a Mac.



Let us now discuss the working of this example. These two lines set things up for the button labeled "North."

```
JButton north = new JButton("North");
c.add(BorderLayout.NORTH, north);
```

The first line is obvious; it just creates the button. In the second line, notice the use of `add`. We first use an integer, the static constant `NORTH` of the class `BorderLayout`, then the second argument is the button. This places the buttton on the north side of the content pane. Note that in the line

```
c.add(BorderLayout.CENTER, center);
```

we could have instead have written

```
c.add(center);
```

since a `BorderLayout` by default places things in the center of the container it rules.

## 6 Can I have Layouts Within Layouts?

Happily, yes.

Java supplies a class called a `JPanel` that is an ideal tool for corralling a group of related graphical widgets. You can add `JPanels` to the content pane. Take note of the fact that a `JPanel`, by default uses a `FlowLayout`. When using a JPanel, you can specify a layout manager either of these two ways. You can pass a new layout manager to the constructor

```
JPanel p = new JPanel(new BorderLayout());
```

or you can do this

```
JPanel p = new JPanel();
p.setLayout(new BorderLayout());
```

There is nothing special about the border layout. You can use any of the layout managers in this fashion. To use a null layout, just pass `null` to `setLayout`.

You can create `JPanels`, and impose a layout manager on each. You can then add these, using layout managers for other panels and containers. Using the basic layout managers and this principle, you have huge latitude. This phenomenon may be used recursively.

## 7 Sur l'Carte: What's on the Menu

Since we have shown you how to create buttons, let's see how to use their close cousins, menus. If you look in at typical application's menus, you will see the following things.

1. There is a menu bar, this is where the menu items live. It is a containter.
2. There are menus; these are the headings. Click on one and you expose...
3. the menu items. These behave like buttons. When you release on a menu item, a specific action occurs.

This section consists of a lab exercise in which you put menus in a window. At this time, you are just constructing a view. Whenever you create menus you need "The Trinity" of swing classes which include

- `javax.swing.JMenuBar` This is a container that holds menus. It can also hold other items as well, such as buttons or textfields.

- `javax.swing.JMenu` This is the class that makes menus. It is a container that can hold menu items. You can add other things to it a well.
- `javax.swing.JMenuItem` This is the class that makes menu items. Releasing on a menu item triggers an action identical to that of clicking a button.

## 7.1 Step 1, Create a Frame and insert a menu bar.

Let us agree to begin with this class. Note that the trinity is imported here.

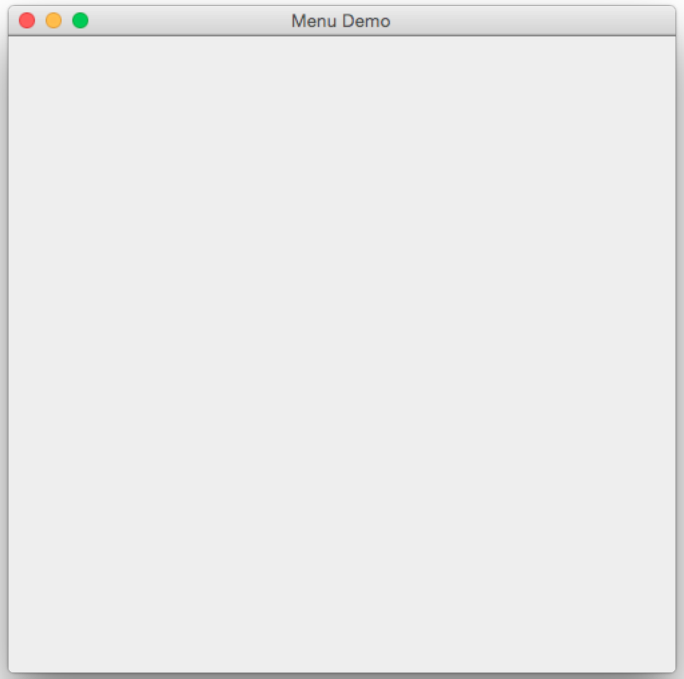
```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;

public class MenuDemo extends JFrame implements Runnable
{
    public void run()
    {
        setSize(500,500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        MenuDemo md = new MenuDemo();
        javax.swing.SwingUtilities.invokeLater(md);
    }
}
```

Now comes your job. Put your code in the `run` method before `setVisible`.

1. Look in the `JMenuBar` documentation. How do you make an empty JMenuBar?
2. Look in the `JFrame` documentation. What do you do to put the menu bar you just created into the `JFrame`?
3. Right after `setSize`, add this code, `setTitle("Menu Demo");`

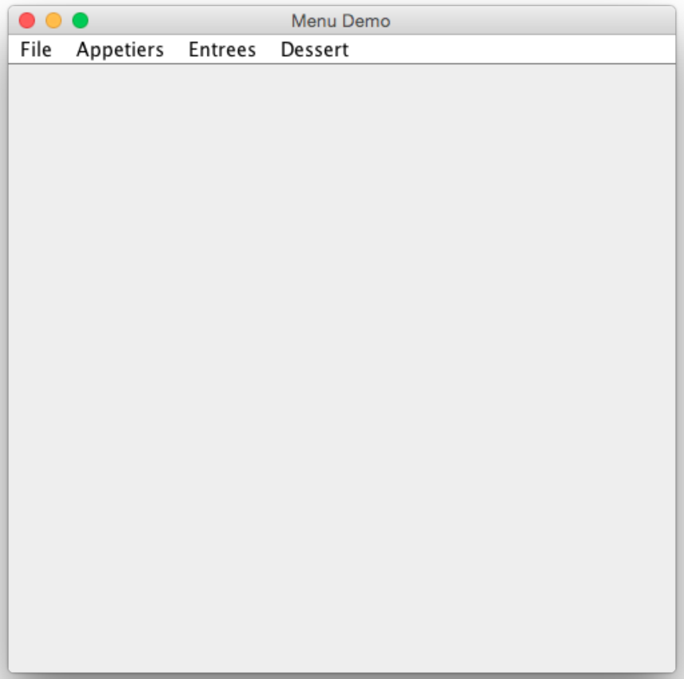
If you complete this correctly, here is what you will see



If you look *really* closely, you can see a one pixel high menu bar. It's small because it's empty.

**Step 2, Add Some Menus** It's time for you to write some more code.

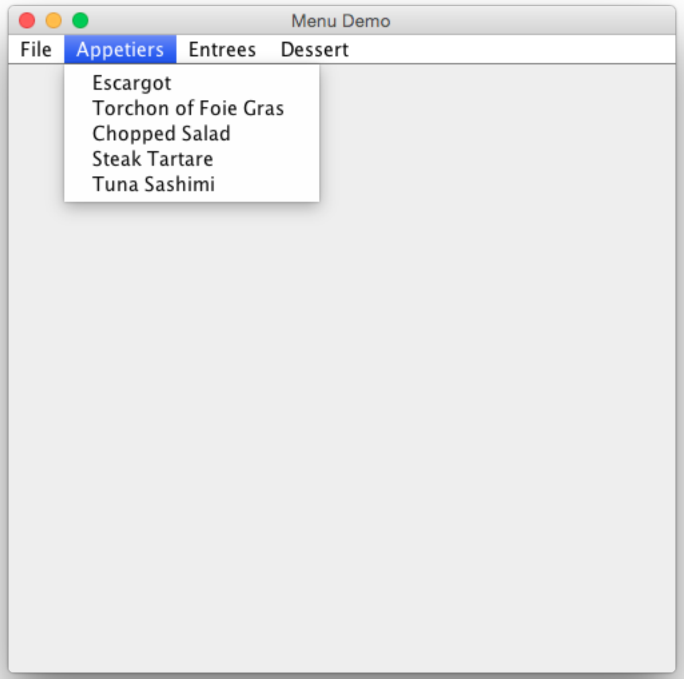
1. Make four `JMenus` named File, Appetizer, Entree, and Dessert.
2. Get them in into the menu bar. Note that the method for doing this is an inherited one. How did we put buttons in a container? Expect some parallelism.



**Step 3, Populate the Menus** Now you will add menu items to the menus as specified.

1. Add a Quit menu item to File
2. To the Appetizer Menu, add Escargot, Torchon of Foie Gras, Chopped Salad, Steak Tartare, and Tuna Sashimi menu items.
3. To the Entree menu add these menu items: Steak au Poivre, Seared Yellowfin Tuna, Roast Quail, Pork Tenderloin, and Rack of Lamb.
4. To the dessert menu add these items: Bananas Foster, Molten Chocolate Cake, Bourbon Pecan Pie, House Made Ice Creams

If you do this correctly, you will be able to click on a menu, see its contents, and select one.



## 8 Creating a Complex View

In this section, we will create a view for a calculator. We will create a panel to hold the a $4 \times 3$ number pad that looks like this.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 0 | . | (-) |

The (-) button is for changing the sign of a number. To its right we might have a vertical panel of operator buttons that includes $+$, -, *, $/$ and $=$. We shall do this, but first let us attend to an important matter. Begin by entering this code into the DrJava code window.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class Calculator extends JFrame implements Runnable
{
    public void run()
    {
    }
    public static void main(string[] args)
    {
        Calculator c = new Calculator();
        javax.swing.swingutilities.invokelater(c);
    }
}
```

Compile right away to ensure you have entered it correctly. We have several import statements which will be the ones we will need as we develop this application. Now we add more code. We need two JPanels to hold the number keys and the op keys. We need to set layouts for each panel and then, in turn, add them to the content pane.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class Calculator extends JFrame
{
    JPanel numberPanel;
    JPanel opPanel;
    public Calculator()
    {
        numberPanel = new JPanel(new GridLayout(4,3));
        opPanel = new JPanel(new GridLayout(5,1));
    }
    public void run()
    {
        getContentPane().add(BorderLayout.CENTER, numberPanel);
        getContentPane().add(BorderLayout.EAST, opPanel);
    }
    public static void main(string[] args)
    {
        Calculator c = new Calculator();
        javax.swing.swingutilities.invokelater(c);
    }
}
```

So far, nothing is visible, so let us select a size and make it visible. We also set a default close operation so the app quits when its go-away button is clicked.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class Calculator extends JFrame implements Runnable
{
    JPanel numberPanel;
    JPanel opPanel;
    public Calculator()
    {
        numberPanel = new JPanel();
        opPanel = new JPanel();
    }
    public void run()
    {
        setSize(500,400);
        opPanel.setLayout(new GridLayout(5,1));
        numberPanel.setLayout(new GridLayout(4,3));
        getContentPane().add(BorderLayout.CENTER, numberPanel);
        getContentPane().add(BorderLayout.EAST, opPanel);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Calculator c = new Calculator();
        javax.swing.SwingUtilities.invokeLater(c);
    }

}
```

Now let us add a little code to the `run` method so the application will be visible when it runs.

When you compile this, next hit F2 to run it. An empty, title-less window will appear on your screen. Go to the first line of the constructor and place the line at the beginning of the constructor.

```
super("Calculator Demo");
```

Your window will get a title in the title bar. Recall that the `super` keyword launches a call to the parent constructor. This constructor causes a title to be placed in the title bar. Our calculator, however, is still bereft of buttons. Let us create these next. Add two new state variables

```
JButton [][]numberKeys;
JButton [] opKeys;
```

These are arrays of buttons. The `numberKeys` array is a two-dimensional array, i.e. a grid of buttons. Think of the first number as specifying the number of rows and the second as specifying the number of columns. It makes sense to use an array since we are keeping related things all in one place, and the collections we are keeping are of a fixed size. It will next be necessary to initialize the buttons in the constructor. First we will take care of the number keys. Add this code to the constructor.

```
numberKeys = new JButton[4][3];
//get in digits 1-9 with a dirty trick
for(int k = 0; k < 3; k++)
{
    for( int l = 0; l < 3; l++)
    {
        numberKeys[k][l] = new JButton("" + (3*k + l + 1));
        numberPanel.add(numberKeys[k][l]);
    }
}
//fill in the rest of the number keys manually
numberKeys[3][0] = new JButton("0");
numberPanel.add(numberKeys[3][0]);
ys[3][1] = new JButton(".");
numberPanel.add(numberKeys[3][1]);
numberKeys[3][2] = new JButton("(-)");
numberPanel.add(numberKeys[3][2]);
```

Let us explain some of the things occurring here. The line

```
numberKeys = new JButton[4][3];
```

directs that we create an object capable of pointing at an array of `JButtons` with four rows and three columns. Next comes a `for` loop for getting each entry of the array to point at an actual button. It then causes that button to be added to the panel of number keys. Do you see how this dirty trick got the digits 1-9 in their proper places? Notice the exploitation of lazy evaluation as well.

```
for(int k = 0; k < 3; k++)
{
    for( int l = 0; l < 3; l++)
    {
        numberKeys[k][l] = new JButton("" + 3*k + l + 1);
        numberPanel.add(numberKeys[k][l]);
    }
}
```

After the `for` loop, we just added the remaining buttons in one–by–one.

Now compile and run; you will see a numerical keyboard occupying the content pane. Since we haven't put anything in the op panel, it does not yet appear. We shall now create the op panel. Append these pieces of code to the constructor. Compile and check after you add each one. We begin by creating all of the op buttons. They live in an array with five elements.

```
opKeys = new JButton[5];
opKeys[0] = new JButton("+");
opKeys[1] = new JButton("-");
opKeys[2] = new JButton("*");
opKeys[3] = new JButton("/");
opKeys[4] = new JButton("=");
```

This handy little `for` loop finishes the job.

```
for(int k = 0; k < 5; k++)
{
    opPanel.add(opKeys[k]);
}
```

Compile and run and you will see the completed product. You can see that we can integrate various containers into the content pane, each with a different layout manager to achieve professional–looking effects.

Let us conclude by showing the entire program.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class Calculator extends JFrame implements Runnable
{
```

```
JPanel numberPanel;
JPanel opPanel;
JButton [][]numberKeys;
JButton [] opKeys;
public Calculator()
{
    super("Calculator Demo");
    numberPanel = new JPanel();
    opPanel = new JPanel();
}
public void run()
{
    setSize(500,400);
    opPanel.setLayout(new GridLayout(5,1));
    numberPanel.setLayout(new GridLayout(4,3));
    getContentPane().add(BorderLayout.CENTER, numberPanel);
    getContentPane().add(BorderLayout.EAST, opPanel);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    numberKeys = new JButton[4][3];
    //get in digits 1-9 with a dirty trick
    for(int k = 0; k < 3; k++)
    {
        for( int l = 0; l < 3; l++)
        {
            numberKeys[k][l] = new JButton("" + (3*k + l + 1));
            numberPanel.add(numberKeys[k][l]);
        }
    }
    //fill in the rest of the number keys manually
    numberKeys[3][0] = new JButton("0");
    numberPanel.add(numberKeys[3][0]);
    numberKeys[3][1] = new JButton(".");
    numberPanel.add(numberKeys[3][1]);
    numberKeys[3][2] = new JButton("(-)");
    numberPanel.add(numberKeys[3][2]);
    opKeys = new JButton[5];
    opKeys[0] = new JButton("+");
    opKeys[1] = new JButton("-");
    opKeys[2] = new JButton("*");
    opKeys[3] = new JButton("/");
    opKeys[4] = new JButton("=");
    for(int k = 0; k < 5; k++)
    {
        opPanel.add(opKeys[k]);
    }
    setVisible(true);
    }
    public static void main(String[] args)
    {
        Calculator c = new Calculator();
        javax.swing.SwingUtilities.invokeLater(c);
    }
}
```

**Programming Exercises**

1. Look up the class `JTextField` in the API guide. Modify the calculator code to place a `JTextField` with a white background on the top of the calculator. This is preparation for producing a display in which to show numbers. Cause the `JTextField` to display some text.
2. Look up the class `Font` in the API guide. Look in the `JButton` class and see if you can set the font to be bold and to have size 36 numbers on the buttons.
3. Make the text in the `JTextField` right-justified. You may need to look in fields or methods from parent classes.
4. Make the background of the `JTextField` black and the type red, as you might see on an old-fashioned calculator.

## 9 Model-View-Controller

So far, we have confined ourselves to creating the graphical faces, or views, of applications. They have pretty faces but do no useful work. The next job is to enable the graphical features we place in a GUI app.

The basic anatomy of a GUI application will be created along the lines of the *Model-View-Controller* design pattern. The view is simply the graphical interface of the application. The model is the application's state, or business logic. The controller mediates between the model and the view. When a button is pushed, the state of the application (the model) is updated, and the view receives any necessary updates as well. The controller in Java GUI programs consists of *listener classes* that "hear" events and execute code in reaction to them.

Both buttons and menu items broadcast an `ActionEvent` when they are selected or pushed. Note that a menu item, fundamentally, is just a button; when it is selected, it broadcasts an `ActionEvent`. So, here is how the whole thing goes.

1. The user pushes a button or selects a menu item.

2. The widget broadcasts an `ActionEvent`.
3. An `ActionListener` attached to the widget executes the code in its method

   `{actionPerformed(ActionEvent e)}`

   This code may carry out actions or change object state. This causes the widget to be "live" in the sense that the program reacts when the widget is activated.

Notice that the listener must have a method called

```
public void actionPerformed (ActionEvent e)
{
}
```

in its public interface. How is this enforced? It is enforced via the mechanism of interfaces. We shall now turn our attention to building the controller.

## 10 Making a `JButton` live with `ActionListener`

A We have created a GUI with buttons, and in the exercises, you made an application with menus in the exercises. If you haven't done that, now is a good time for you to go back and create that. So far, what we have accomplished is the creation of a view for an application. However, these pretty things *do nothing*.

A button requires a class that implements `ActionListener` to make it live.

We begin by creating the graphical shell with the button.

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

public class LiveButton extends JFrame implements Runnable
{
    private JButton b;
    public LiveButton()
    {
        super("Live Button Demo");
        b = new JButton("Panic");
    }
    public void run()
    {
        getContentPane().add(b);
        setSize(300,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args )
    {
        LiveButton l = new LiveButton();
        javax.swing.SwingUtilities.invokeLater(l);
    }
}
```

Compile and run; you will have a window with a button in it. Next, create another class called `ButtonListener` that implements `ActionListener`. Note the necessary import statements.

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

public  class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("AAAAAAAAAAAAAAAAAAAAAHHHH!!");
    }
}
```

Now add the following line of code to `run` inside of `LiveButton`.

```
b.addActionListener(new ButtonListener());
```

When you click on the button, it *broadcasts* an `event`, telling your program, "I have been pushed." When a button is not live, no one is listening. Now we create an instance of a `ButtonListener`. That is like buying a radio allowing you to listen for `ActionEvents`, which are broadcast by pushed buttons. When you do `b.addActionListener(new ButtonListener())`, you are now telling that `ButtonListener` to tune in on `b` and to execute its `actionPerformed` method each time the button `b` is clicked. For any given button, you may attach as many `ActionListeners` as you wish, subject of course, to limits on memory.

For your convenience, we display the entire program here

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

public class LiveButton extends JFrame implements Runnable
{
    private JButton b;
    public LiveButton()
    {
        super("Live Button Demo");
        b = new JButton("Panic");
    }
    public void run()
    {
        getContentPane().add(b);
        setSize(300,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        b.addActionListener(new ButtonListener());
        setVisible(true);
    }
    public static void main(String[] args )
    {
        LiveButton l = new LiveButton();
        javax.swing.SwingUtilities.invokeLater(l);
    }
}
```

You will need this, too, in the same directory.

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

public  class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("AAAAAAAAAAAAAAAAAAAAAHHHH!!");
    }
}
```

## 11 Lambdas and Functional Interfaces

We will now look at another route to doing event handling for buttons using the fact that `ActionListener` is a functional interface. Here we introduce a new construct, a *lambda*, which is an anonymous function not belonging to any class. Lambdas are also found in Python.

Consider this example

```
    e -> System.out.println("AAAAAAAAAAAAAAAAAAAAAHHHH!!");
```

This is an anonymous function which has one argument, `e` and which has body `System.out.println("AAAAAAAAAAAAAAAAAAAAAAHHHH!!");`. Note the arrow in the syntax: it is formed by a minus sign followed by a greater-than sign. It is a token; whitespace within it will break it. So, note that `- >` will reap you some really ugly compiler errors.

We will now modify our class `LiveButton` to take advantage of this. We will then see how it all works.

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

public class LiveButton extends JFrame implements Runnable
{
    private JButton b;
    public LiveButton()
    {
        super("Live Button Demo");
        b = new JButton("Panic");
    }
    public void run()
    {
        getContentPane().add(b);
        setSize(300,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        b.addActionListener(
        e -> System.out.println("AAAAAAAAAAAAAAAAAAAAAAHHHH!!");
        );
        setVisible(true);
    }
    public static void main(String[] args )
    {
        LiveButton l = new LiveButton();
        javax.swing.SwingUtilities.invokeLater(l);
    }
}
```

You no longer need the `ButtonListener` class.

It is time to ask, "How did this work?" The change we made in our code all lies in the line where we call `b.addActionListener(...)`. In the original version, we passed an object of type `ButtonListener` to it. Since `ButtonListener` implements the `ActionListener` interface, it must have an `actionPerformed(ActionEvent e )` method. When the button is pushed `actionPerformed` is called and the message gets put to `stdout`.

In the new version, we passed the lambda

```
e -> System.out.println("AAAAAAAAAAAAAAAAAAAAAHHHH!!");
```

Recall we said that `ActionListener` is a functional interface: it has exactly one abstract method. The Java8 compiler performs a feat of type inference here: it says, after putting on its Sherlock Holmes hat and firing up its pipe, "Gee, this lambda has a single argument and a void return type. It is being passed to the method `b.addActionListener`, which requires an action listener. Therefore, I must deduce that this lambda, is, in fact, the required `actionPerformed` method!"

Sneaky, eh? This is a benefit that solely inheres to functional interfaces. You can happily exploit it for both buttons and menu items. Later, we will meet `MouseListener`, which requires five methods. You won't be able to use this nifty trick there. At least not without some serious trickery, which in fact can be managed.

## 11.1 Lambdas in General

Let us make clear all the features of lambdas. 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 : print("foo")
(x,y,z) -> x*y*z                //same as lambda x,y,z : x*y*z
```

You can also specify types in the sig. 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 1
    //line 2
    }
```

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.

Can a variable point at a lambda? The answer is yes. The key is provided by using a variable of functional interface type. Recall that `ActionListener` is a functional interface which specifies one method, public void actionPerformed(ActionEvent e). You can do the following.

```
JButton duck = new JButton("Event Emitter");
ActionListener quack = e -> System.out.println("I emit an event.");
duck.addActionListener(quack);
```

How does this work? It occurs via the magic of type inference. The `ActionListener` interface specifies exactly one method. We create a variable of interface type, duck. Since Java is strongly typed, it knows duck's type. We now assign a lambda with a void return type with sig `[ActionListner]`. Java 8 now says, "I know that this must be the actionPerformed method, since the lambda's sig and return type match this functional interface's sole specified method. Bam! You have created an instance of an anonymous class.

Since this is an ActionListener, it can be passed to the `JButton`'s `addActionListener` method.

**Programming Exercise** Get out your code for `MenuDemo.java`.

1. Use lambdas to have menu items print out "You ordered `xxx` for a `yyy`" where `xxx` denotes the name of the item, and `yyy` can be "appetizer," "entree," or "dessert." You can do this becasue `JMenuItems` emit `ActionEvents` when they are selected.

## 12 Inheritance and Graphics

We will begin by creating the standard graphical shell.

```
import javax.swing.JFrame;

public class DrawFrame extends JFrame implements Runnable
{
    public void run()
    {
        setSize(500,500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        DrawFrame df = new DrawFrame();
        javax.swing.SwingUtilities.invokeLater(df);
    }
}
```

To draw, we do the following. We create a class that inherits from `JPanel`. We then override the method `public void paintComponent(Graphics g)`.

This method is called automatically by the OS whenever the window refreshes. You can also call it by using the `repaint()` method. Windows refresh when they are maximized, minimized or re-sized. They may also refresh at other times when the OS sees fit to do so. Repainting any component causes all of the components inside of it to repaint recursively.

So, to get started we create a class such as `DrawPanel.java` shown here. Note the use of the `@Override` annotation.

```
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.Color;

public class DrawPanel extends JPanel
{
    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);//always do this.
        //Place instructions here to draw all of the
        //stuff that goes in this window.
    }
```

Now add one of these panels to the `DrawFrame`. Just add this line in `run()` method.

```
getContentPane().add(new DrawPanel());
```

That embeds the `DrawFrame` inside of your app. You can impose a layout manager on the content pane and add several panels for drawing if you wish.

What is a `Graphics`? It is a combined pen and paintbrush that has 16,777,216 colors. You should visit the API page and experiment with the methods. We will show a few examples here. Note that you can set the color of the pen by using `g.setColor()`.

Place these lines in `paintComponent()`.

```
g.setColor(Color.BLACK);
g.fillRect(0,0,getWidth(), getHeight());
```

This will fill the pen with black pixels and then paint the entire window black. Try re-sizing the window. The entire window will be filled with black. Why? When the window is re-sized, the window repaints. A `JPanel` knows its width and height; we obtain these with the accessor methods `getWidth()` and `getHeight()`. Now augment the `paintComponent` method with these lines.

```
System.out.printf("height = \%s\n", getHeight());
System.out.printf("width = \%s\n", getWidth());
```

Re-size the window and watch the `stdout` window. As the window re-sizes, updates the height and width to the terminal. Experiment with this and watch its behavior.

Now let's make a Carolina blue rectangle in the window. Just add this.

```
g.setColor(new Color(0xabcdef));
g.fillRect(100,100,75,75);
```

Here is a little magic; we can make a Wolfpack red rectangle that re-sizes with the window.

```
g.setColor(Color.RED);
g.fillRect(getWidth()/4,getHeight()/4,
    getWidth()/2,getHeight()/2);
```

Now make some colorful circles. Add this.

```
g.setColor(Color.BLUE);
g.fillOval(200, 200, 50, 50);
```

```
    g.setColor(Color.YELLOW);
    g.drawOval(250,250,100,100);
```

Finally put it all in a green jail.

```
    g.setColor(Color.GREEN);
    for(int k = 0; k < getHeight(); k+= 20)
    {
        g.drawLine(0, k, getWidth(), k);
    }
```

You can see that drawing in a window is a simple process. To recap you do the following.

1. Write a class that extends `JPanel`
2. Override the parent method

   `public void paintComponent(Graphics g){}`
3. In the first line of this method, put

   `{super.paintComponent(g);}`

   for safety; this clears the screen gracefully before repainting.
4. Place an instance of this class in the content pane of your `JFrame`.

## 13 Abstract Classes: Second Pass

Suppose you have a closely related group of classes. You are remembering the eleventh commandment, *"Thou shalt not maintain duplicate code!"* This animadversion reminds us that, to maintain programs, we sometimes need to change code. When we do, we do not want to be ferreting out identical code segments in a group of classes and making the same edit on all of them. That is folly-filled nonsense to be avoided at any cost.

Such a thing, does, indeed exist and we have already seen it: the abstract class. We already know that, like an interface, an abstract class cannot be instantiated. Like an interface, you can create variables of abstract class type that can point at any descendant class. The abstract class is the item that lies between the empty–looking interface and the fully furnished class.

An abstract class can have zero or more abstract methods. Abstract methods are just bodyless method headers. Any non-abstract class inheriting from your abstract class must have all of the method specified by your abstract class.

We shall create an example of a related group of classes. Suppose you are Old MacDonald and you have a farm. You are going to write code to keep track of the many things that are on your farm. There are several broad categories you might have: `Animal`, `Implement`, `Building` and `Crop` might be some of these categories. These sorts of things are good choices for being abstract classes or interfaces.

The main thing that motivates you to use abstract classes is that you might actually have classes that share code. This is when you use abstract classes. If the classes merely share functionality, you might want to use an interface.

We will build small *class hierarchy*. All classes live in a family tree. All of our farm classes descend from the root class `FarmAsset`.

```
public abstract class FarmAsset
{
    private String name;
    public FarmAsset(String _name)
    {
        name = _name;
    }
    public String getName()
    {
        return name;
    }
}
```

Since this class is marked `abstract`, it cannot be instantiated. You must produce a new descendant class to make an actual instance, but you can create variables of `FarmAsset` type Now let us make an `Animal` class. Some class designers would make their variables `protected` for convenience, but we make them `private` and initialize them via calls to `super`. This is consistent with the design principle that we make our internal working of our classes private. Observe that `Animal` inherits `getName` from its parent.

```
public abstract class Animal extends FarmAsset
{
    private String noise;
    private String meatName;

    public Animal(String _name, String _noise, String _meatName)
    {
        super(_name);
        noise = _noise;
        meatName = _meatName;
    }
    public String getNoise()
    {
        return noise;
    }
    public String getMeatName()
    {
        return meatName;
    }
}
```

Next we create a class for crops.

```
public abstract class Crop extends FarmAsset
{
    private double acreage;
    public Crop(String _name, double _acreage)
    {
        super(_name);
        acreage = _acreage;
    }
}
```

Finally, we create a concrete (non-abstract) class which we can instantiate. We shall begin with the honorable pig. Notice the brevity of the code. What we did here was to push the common features of farm assets as high up the tree as possible. You do not need to create the `getName`, `getMeatName` and `getNoise` for each animal. We added a `toString` method for `Pig` so it would print nicely.

```
public class Pig extends Animal
{
    public Pig(String _name)
    {
        super(_name, "Reeeeet! Snort! Snuffle!", "pork");
    }
    @Override
    public String toString()
    {
        return "Pig named " + getName();
    }
}
```

```
> Pig p = new Pig("Wilbur");
> p.getNoise()
"Reeeeet! Snort! Snuffle!"
> p.getName()
"Wilbur"
> p.getMeatName()
```

```
"pork"
> p
Pig named Wilbur
```

Here is another simple example. Suppose you run a school and are in charge of keeping track of all people on campus. You might have a class called `Employee`, with an abstract method `computePay()`. You know that all employees are paid, so you place this line in your *Employee* class.

```
public abstract double computePay(double hoursWorked);
```

Your school likely has hourly and salaried employees. A salaried employee's paycheck is fixed each pay period. An hourly employee's pay is computed by multiplying the hours worked by the hourly rate of pay, and adding in the legally required time-and-a-half for overtime. You would likely have two classes extending the abstract *Employee* class, `HourlyEmployee` and `SalariedEmployee`. All employees have many things in common: these go into the parent class. You have to know their social security number, mail location, and department. Your `Employee` class might have a parent class `Person`, which would keep track of such details common to everyone on a school campus, including, name, address, and emergency contact information. From `Person`, you might have child class `Student`. A `Student` should know his locker number, class list, and grade.

Using classes, we model the school in a "real-life" way. We create a hierarchy of classes, some of which are abstract. We look at various bits of information germane to each class, and we keep that information as high as possible in the class hierarchy; for instance, the name of an individual is really a property of `Person`, so this class should have a `getName()` method. All employees have a paycheck, so we create the abstract `computePay()` method so that every class of employee we ever create is required to to have the `computePay()` method. That requirement is enforced by the compiler, just as it is for interfaces. It confers an additional benefit. A variable of type `Employee` can call the `computePay()` method on the object it points to, and that object will compute its pay, regardless of the type of employee it represents.

## Programming Exercises

1. Create a class for `Cow` and `Goat`. Instantiate these and have a variable of type `FarmAsset` point at them. What methods can you call successfully? Have a variable of type `Animal` point at them. What methods can you call now?
2. Have you ever noticed that the meat name for fowl is the same as the animal's name. For instance, we call chicken meat "chicken" and duck meat "duck." Create a new abstract class `Fowl` that exploits this. Then create classes `Chicken`, `Goose` and `Duck`. Point at these with an `Animal` variable, and see what methods you can call.

3. Create a new abstract class `Implement` to encompass farm implements such as tractors, combines, or planters. Make some child classes for farm implements.

## 14 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 angry yellow, along with 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`.

| `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 inherit from 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 a `final` local variable inside of the method. |

### 14.1 `final` Classes and Performance

In the days of yore, people used the `final` keyword to improve performance. When a method is overridden, the selection of the code to be executed is done dynamically, i.e. at run time. In old versions of java, this came at some cost to performance. Today that is no longer really true. Use finality to enforce design intent and to keep the structure of any hierarchy of classes you produce rational. Do not feel obliged to use it for performance reasons. We will now return to the world of GUIs.

## 15 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 `BigInteger`s.

- **delegation principle** This is the principle that says that an object is responsible for executing method codes, regardless of the type of variable pointing to it.
- **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.
- **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.
- **listener classes** are classes that "hear" events and respond to them by executing code. So far, we have met the `ActionListner` interface for classes whose objects respond to button pushes and menu item selections.
- **model-view-controller design pattern** This is the general anatomy of a GUI app. 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 listeners.
- **override** When a child class re-implements a method of the parent class, this takes primacy and supercedes 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.
- **sibling classes** Two classes are sibling if they have the same parent class.
- **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. For example in the method `paintComponent` in a `JPanel`, you call `super.paintComponent(g)` to properly clear the panel between repaintings.
- **thread** A thread is an independent sub-process of your java code that has its own call stack.
- **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 `JFrame` can contain an entire application. A container widget such as a `Container` or a `JPanel` can have other widgets added to it.