Programming in Java

John M. Morrison

March 8, 2013

Contents

Chapter 0

Getting Started

0.0 Introduction

In this chapter you will learn how to get Java up and running, and understand how compile a program. This chapter will deal purely with nitty-gritty mechanical matters. Getting these out of the way will pave the path for the more interesting matters awaiting us in Chapter 1.

Let us begin by getting Java running on your machine. You will need to connect your box to the Internet or you will need to write certain materials to a CD or DVD so they may be installed on your box.

0.1 Getting a Java Development Kit

To build Java programs on your box, you will need a *Java Development Kit*. This piece of software is available for all major computing platforms. We will step through the process for the Windoze, Mac and Linux platforms.

- **Windoze** Go to `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. and you can get the most current JDK. Begin by hitting the "Download JDK" button. You will be asked the platform you are using. Agree to the license agreement and hit Continue. You will need to know if you are runnig 32 or 64 bit windoze. You will download a file something like `jdk-6u24-windows-586.exe`. This name may vary whether you are using 64 bit or 32 bit windows. The number after the `jdk-` is version number.
  When the download is done, double–click on the download's icon to launch it. A Windoze install shield will come up. Agree to the license agreement

and click through the boxes to do the install. When you finish, reboot. Your machine is now fully Java-enabled.

- **Mac** Macs come with the JDK installed.
- **Linux** In Ubuntu, You will need `openjdk` or `sun-javaX-jdk`, where `X` is the version number. You will need to get into the extended repositories for Ubuntu to get the package sun-java6. You should not have to reboot when this process is complete. You can also obtain the binaries from Oracle. All major distributions of Linux will have repositories will also have JDK binaries.

If you are building a CD Place the JDK installer in a folder you will later copy to the CD. Go to the DrJava site, and download the DrJava documentation and the DrJava App for your platform and the JAR file. Place these on your CD so you can finalize it and get it ready for your (offline) machine to read. All of these items are freely available for you to use and to share with your friends, colleagues and classmates.

If you are going to work offline, you should also download the Java API documentation and install it on your machine. This documentation is a free "Encyclopaedia of Java" that will be extremely helpful.

0.2 Getting DrJava

We will use the DrJava integrated development environment (IDE) in this book. Its unique features will allow us to approach the language correctly and coherently. It represents an ideal compromise between ease of use and powerful features for beginning–to–intermediate Java programmers. It has an interactive mode similar that of Python that allows you to "talk" to Java interactively. We will make a lot of use of this feature. You can obtain DrJava from

```
http://www.drjava.org
```

This site has complete instructions on downloading and configuring DrJava, including how to download and install the Java development kit. Take a little time to browse this very useful documentation.

If you are a Windoze or Mac user, download the App. On LINUX machines, get the `.jar` file. This file is fairly small; it is a few megabytes.

To launch DrJava on a Mac or in Windoze, double-click on its icon.

In Linux, open a terminal navigate to the directory containing DrJava. Change its name to `drjava.jar` (for convenience) and enter

```
$ java -jar drjava.jar
```

at the UNIX command line. In a few seconds, the DrJava window will appear on the screen. If you are asked to locate `tools.jar`, skip to the next section immediately and follow the instructions there.

The DrJava window is divided into three portions. On the bottom you will see a tabbed window with tabs named Interactions, Console, and Compiler Output. Click on the interactions tab. On top there is a skinny window on the left that will contain the text

```
(Untitled)
```

and the big window on the right that is the *code window*; you will enter your Java code in the code window. Enter the following text in the code window

```
public class Foo
{
}
```

DrJava has a toolbar underneath its menu bar. Most things you need to do in DrJava can be run from either. To run the powerful program we just created, you can do any of the following.

- Hit the F5 button on your keyboard.
- Hit the Compile button in the toolbar.
- Click on the Tools menu and select the Compile All Documents or Compile This Document item.

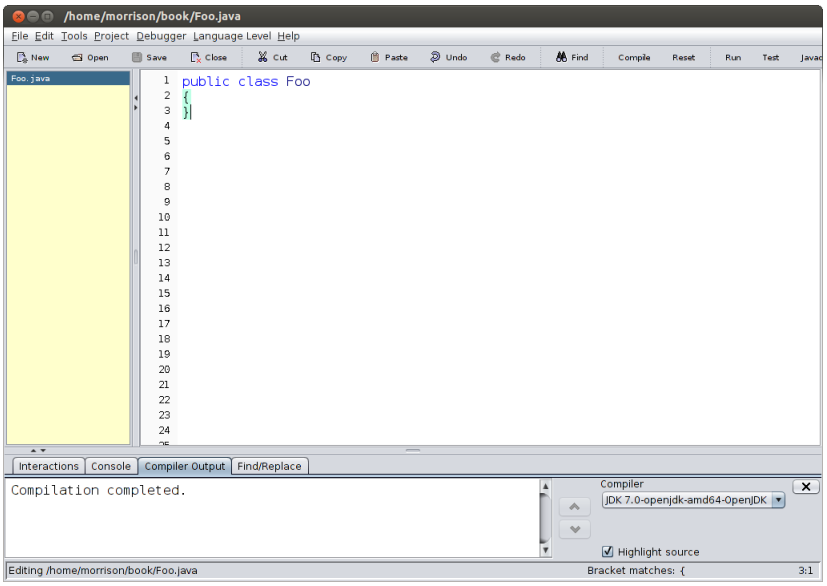
You may be asked to locate your `tools.jar` file at this time. If so, skip to the next section, do that, then return here.

You will be prompted to save your file. Java requires you name it `Foo.java`; the IDE will make this the default choice. Save your program. It is a good idea for you to make a directory to hold your Java programs and save it there.

Click on the Compiler Output tab on the bottom window. You will see the reassuring text

```
Compilation Completed
```

Here is what you will see.



As advertised, you can see the file window on the left. It wll display all of the files you are currently editing. You can switch between files by clicking on the file names. The big window is the code window. You can obtain the line numbers by folling the directions in the next chapter. If you are impaitent, click control-; and see the preferences. There is a check box for show line numbers. At the bottom you see a triple-tabbed window on the left. It offers you a choice of the interactions pane, Console (`stdout`), Compiler output, or if you have clicked on Find/Replace, that shows up too, as has happened here.

0.3 What the heck is `tools.jar`?

When you first use DrJava, you may be asked to locate the `tools.jar` file in your Java development kit. The instructions on the DrJava site will help you locate this in Windoze. Owing to the poor search features on Windoze, we cannot recommend you search for this file. You can easily find `tools.jar` file on a Mac; just search for it in Finder.

In LINUX, you can find the file using `locate`. Before you do this enter

```
$ sudo updatedb
```

to update the index of your hard drive. You should do this periodically as a matter of maintenance on your UNIX box to make searching easy. If you have never done this, it may take a few minutes for this procedure to execute. Then type

```
$ locate tools.jar
```

at the command line. Once you know the path to your `tools.jar` file, navigate to it and DrJava will be ready to roll.

If the sample program shown in the last section compiles for you, the goal of this chapter is met. We will turn to actually creating programs that do useful stuff in the next chapter.

Exercise Open the Dr. Java documentation and look through it. Perform some customization you might want such as font size (this is important for teachers who use their PC attached to a projector), background color, or colors in general.

Chapter 1

Introducing Java

1.0 How does Java Work on a Mechanical Level?

We will begin by looking at the mechanics of producing a program. We will then sketch a crude version of what actally happens during the process and refine it as we go along. Here is a simplified life-cycle for a Java program.

1. **Edit** You begin the cycle by entering code in the code window and saving it. Each file of Java will have a `public` class in it. The class is the fundamental unit of Java code; all of your Java programs will be organized into classes. This classes are similar to those in Python; later we will compare them.
   The name of the class must match the name of the file; otherwise you will get a nastygram from the compiler. As you saw in the example at the end of the last section, the file containing `public class Foo` must be be named `Foo.java`; failure to adhere to this convention will be punished by the compiler.
   Deliberately trigger this error by changing the name of your class from `Foo` to something else. Hit F5; your new class name will be highlighted in angry yellow. The compiler enforces this convention. An optional but nearly universal convention is to capitalize class names. You should adhere to this rule in the name of wise consistency. This is done by other Java programmers; uncapitlized class names just confuse, annoy and vex others.
2. **Compile** Java is an example of a *high–level language*. A complex program called a *compiler* converts your program into an executable form.
   If your program contains syntactical errors that make it unintelligible to the compiler, the compilation will abort. When this happens, nothing executes and no executable file is generated. In contrast, in the Python language, the program stops running when an syntactical error is encoun-

tered; in Java the program does not run at all unless it compiles successfully. There is no compile time in Python.
If your program does not compile, you will get one or more error messages. You can click on the Compiler Output tab of the bottom window to see these. You will see "angry yellow" in the code window at the scene of the crime. You will need to re–edit your code to stamp out these errors before it will compile.
Java compiles programs in the machine language of the *Java Virtual Machine*; this machine is a virtual computer that is built in software. Its machine language is called *Java byte code*. In the `Foo.java` example, successful compilation yields a file `Foo.class`; this file consists of Java byte code. Your JVM takes this byte code and converts it into machine language for your particular platform, and your program will run. Java is not the only language that compiles to the JVM. Others include Clojure, a Lisp dialect, Processing, an animation language created in the MIT media lab, and Groovy, a scripting language. There is even a JVM implementation of Python called Jython.

3. **Run** If your compilation succeeds, you will be able to run your program. For now, we will run our programs in the interactions pane. You can see the interactions pane by clicking on the Interactions tab in the bottom window. You will run your program and see if it works as specified. If not... back to the step **Edit**. You can have errors at run time, too. These errors result in an "exploding heart;" these ghastly things are nasty error messages that are printed in dark red in the compiler output window. You can also have logic errors in your program; in this case, the program will reveal some unexpected behavior you did not want.

You will often hear the terms "compile time" and "run time" used. These are self-explanatory. Certain events happen when your program compiles, these are said to be compile time events. Others happen at run time. These terms will often be used to describe errors.

1.0.1 Customizing DrJava

There are several things you can do to make DrJava easier to use. Choose the Edit menu and select Preferences. Under Display Options, you can set the font size. Choose a size that is easy for you to read. We strongly recommend a Monospaced font.

Go to the Colors under Display Options. Make your background color `0xFFFFCC`, or red: 255, green: 255, blue 208. You can do this by clicking on the RGB tab. This color background is much easier on your eyes than the conventional white. The rest of the defaults are OK. Go under Miscellaneous. We recommend indent level: 4. Go into Display Options and check "Show all Line Numbers." The compiler specifies errors by line number, so having them

out is convenient. When you are done, hit the Apply button and close the preference window. You should adjust the font if you find it too large or small. We strongly suggest you use a monospace font; it makes programs easier to look at and it also will match the appearance of code examples in this book.

Next, we take a brief tour of Python classes.

1.1 Python Classes and Objects

Python classes have a very simple structure; we will take a quick look at these before wading into Java classes. You can create a Python class with two lines of code.

```
class Simple(object):
    pass
```

A class is a blueprint for creating objects; this principle works the same in Python and Java. Making an object from this class is ... simple. Just do this.

```
>>> class Simple(object):
...     pass
...
>>> s = Simple()
>>> t = Simple()
>>>
```

We have created two objects `s` and `t` that are *instances* of this class.

We have learned that objects have state, identity and behavior. Recall that state is what an object knows, behavior is what an object does, and identity is what an object is (a hunk of memory). Since the body of our class is empty, `Simple` objects no nothing and do nothing. The can, however do some basic stuff. They can be represented as strings, and they can be checked for equality. Here we see this.

```
>>> s
<__main__.Simple object at 0x12e3250>
>>> t
<__main__.Simple object at 0x12e3290>
>>> s == t
False
>>>
```

In Python, you can attach things to objects. Watch what is happening here.

```
>>> s.x = "I am x."
>>> s.y = "I am y."
>>> s.x
'I am x.'
>>> t.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Simple' object has no attribute 'x'
>>>
```

Now the object `s` knows its `x` and `y`, but `t` still knows nothing. We can make all instances of our class know `x` and `y` as follows.

```
>>> class Simple:
...     x = "I am x."
...     y = "I am y."
...
>>> s = Simple()
>>> t = Simple()
>>> s.x
'I am x.'
>>> t.x
'I am x.'
>>> s.y
'I am y.'
>>> t.y
'I am y.'
>>>
```

So far, our classes have fixed state. Objects of type `Simple` all have the same `x` and `y`. This is not terrible useful. Suppose we want to make a `Point` class to represent points in the plane with integer coördinates. When we create a `Point`, we might want to specify its coördiantes. To do this, we will use a special method called `__init__`, which runs immediately after the object is created.

Let us now consider this program.

```
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

p = Point()
print ("p = ({0}, {1})".format(p.x, p.y))
q = Point(3,4)
print ("q = ({0}, {1})".format(q.x, q.y))
```

Now run this program and see the following.

```
$ python3 Point.py
p = (0, 0)
q = (3, 4)
```

We see a lot of new stuff here, so let us go through it with some care. We know that the `__init__` method runs immediately after a Point object is created. Its argument list is `(self, x, y)`. The purpose of the `x` and `y` seem clear: they furnish coördiantes to our `Point` object.

We also see this `self`. What is this? When you program in the `Point` class, you *are* a `Point`. So `self` is you. In the statement `self.x = x`, you are attaching the value `x` sent by the caller to yourself. The quantities `self.x` and `self.y` constitute the state of an instance of this class. This is how a `Point` knows its coördinates. The symbols `self.x` and `self.y` have scope inside of the entire class body. Take note that all argument lists of methods in a Python class must begin with `self`.

Now let us make a `Point` capable of doing something. Modify your `Point.py` program as follows.

```
import math
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def distanceTo(self, other):
        return math.hypot(self.x - other.x, self.y - other.y)

p = Point()
print ("p = ({0}, {1})".format(p.x, p.y))
q = Point(3,4)
print ("q = ({0}, {1})".format(q.x, q.y))
print ("p.distanceto(q) = {0}".format(p.distanceTo(q)))
```

Now run this program.

```
$ python Point.py
p = (0, 0)
q = (3, 4)
p.distanceto(q) = 5.0
$
```

The class mechanism enables us to create our own new types of objects. Python supports the class mechanism, and object-oriented programming in general.

Java goes even further: all code in Java must appear in a class.

1.2 Java Classes and Objects

A Java program consists of one or more classes. All Java code that is created is contained in classes. So far you have created an empty class called `Foo`. Since the class is devoid of code, it is of limited use.

In Python, you created programs that consisted of functions, one of which was the "main routine," which lived outside of all other functions. Your programs had variables that point at objects and functions that remember procedures. Java has these features but it works somewhat differently. Let us begin by comparing the time–honored "Hello, World!" program in both languages. In Python we code the following

```
#!/usr/bin/python
print "Hello, World!"
```

A Java vs. Python Comparison Python has classes but their use is purely optional. In Java, all of your code *must* be enclosed in classes. Throughout you will see that Java is more "modest" than Python. No executable code can be naked; all code must occur within a function that is further clothed in a class, with the exception of certain initializations of variables that still must occur inside of a class.

Also, we should remember we have two types of statements in Python, worker statements and boss statements. In Python, boss statements are grammatically incomplete sentences. Worker statements are complete sentences. All boss statements in Python end in a colon (:), and worker statements have no mark a the end. All boss statements own a block of code consisting of one or more lines of code; an empty block can be created by using Python's *pass* keyword.

Java uses a similar system of boss and worker statements. In Java, boss statements have no mark at the end. Worker statements must be ended with a semicolon (;).

In Python, delimitation is achieved with a block structure that is shown by tabbing. In Java, delimitation is achieved by curly braces $\{\cdots\}$.

In Python, a boss statement must own a block containing at least one worker statement. In Java, a boss statement must have a block attached to it that is contained in curly braces. An empty block can be indicated by an empty pair of matching curly braces. Technically, you can get away with omitting the empty block, but it is much better to make your intent explicit.

Knowing these basic facts will make it fairly easy for you to understand simple Java programs.

Now, make the following class in Java and save it in the file `Hello.java`

```
public class Hello
{
    public void go()
    {
        System.out.println("Hello, World!");
    }
}
```

Notice that there is no `self` argument as there is in Python. Next, hit F5 or select Compile from the menus. Save the file. Click on the Compiler Output tab and you will see that compilation has occurred. Now click on the interactions pane. You will see something like this.

```
Welcome to DrJava. Working directory is /home/morrison/book/Java
>
```

The working directory will be the directory occupied by your code `Hello.java`. Henceforth, when showing interactive sessions, we will not show the `Welcome` line. To run the program, click on the interactions tab. The

```
>
```

is a prompt, much like a UNIX prompt. It is waiting for you to enter a Java command. At the prompt enter

```
> greet = new Hello();
```

You have just created an instance of your class named `greet`. The class supplied a blueprint for the creation of an *object* of type `Hello`. Your class has one *method* called `go`. We shall use the term "method" to describe any function that lives inside of a class in Java or Python. Since all code in Java lives inside of classes, functions will be known uniformly as methods. Any object of type `Hello` can be sent the message "`go()`". We now do so.

```
Welcome to DrJava.  Working directory is /home/morrison/book/texed/Java
> greet = new Hello();
> greet.go()
Hello, World!
>
```

The mysterious line `greet = new Hello();` tells Java, "make a new object of type `Hello`." The variable `greet` points at a `Hello` object. Observe that `new` is a language keyword that is designated for creating objects.

You can think of the Java Virtual Machine as being an object factory. The class you make is a blueprint for the creation of objects. You may use the `new` keyword to manufacture as many objects as you wish. When you use this keyword, you tell Java what kind of object you want created, and it is brought into existence. Here we show a second `Hello` object getting created by using `new`.

```
> greet = new Hello();
> greet.go()
Hello, World!
> snuffle = new Hello();
> snuffle.go()
Hello, World!
> greet
Hello@3cb075
> snuffle
Hello@e99ce5
>
```

Now there are two `Hello` objects in existence. Each has the capability to "`go()`." This is the only capability we have given our `Hello` objects so far. Every Java object has the built–in capability of representing itself as a string. The string representation of a `Hello` object looks like

```
Hello@aBunchOfHexDigits
```

You can see we have created two distinct `Hello` objects; the string representation of `greet` is `Hello@3cb075` and the string representation of `snuffle` is `Hello@e99ce5`. Each of the variables `greet` and `snuffle` is pointing at its own `Hello` object.

The method `go()` represents a *behavior* of a `Hello` object. These objects have one behavior, they can `go()`. You can also see here that objects have identity. They "are". the two instances, `snuffle` and `greet` we created of the `Hello` class are different from one another. So far, we know that *objects have identity and behavior.*

This is all evocative of some things we have seen in Python. For example, if we create a string in Python, we can invoke the string method `upper()` to convert all alpha characters in the string to upper case. Here is an example of this happening.

```
>>> x = "abc123;"
>>> x.upper()
'ABC123;'
>>>
```

The Python string object and the Java Hello object behaved identically. When we sent the string `x` the message `upper()`, it returned a copy of itself with all alpha characters converted to upper-case. In the Python interactive mode, this copy is put to the Python interactive session, which acts as `stdout`.

The Java Hello object `greet` put the text `"Hello, World!"` to `stdout`. Click on the console pane to see that. Things put to `stdout` in DrJava are also put to the interactions pane. All things put to `stdout` are printed in green by default.

Now let us go through the program line-by-line and explain all that we see. The first line

```
public class Hello
```

is a boss statement. Read it as "To make a Hello," Since "To make a Hello," is not a complete sentence, we know the class head is a boss statement. Therefore it gets NO semicolon.

The word `public` is an *access specifier*. In this context, it means that the class is visible outside of the file. Python has no such modesty; it lacks any system of access specifiers. Later, you may have several classes in a file. Only one may be public. You may place other classes in the same file as your public class. This is done if the other classes exist solely to serve the public class. Java requires the file bear the name of the public class in the file. The compiler will be angry if you do not do this, and you will get an error message.

The words `public` and `class` are language keywords, so do not use them as identifiers. The `class` keyword says, clearly enough, "We are making a class here." Now we go to the next line

```
{
```

This line has just an open curly brace on it. Java, in contrast to Python, has no format requirement. This freedom is dangerous. We will adopt certain formatting conventions. Use them, or develop your own and *be very consistent*. Consistent formatting makes mistakes easy to see and correct. It is unwise consistency that is the "hobgoblin of small minds;" wise consistency is a great virtue in computing.

The single curly brace acts like tabbing in in Python: it is a delimiter. It delimits the beginning of the `Hello` class's code. The next line

```
    public void go()
```

is a function header. In Python you would write

```
def go():
```

There are some differences. The Python method header is a boss statement so it has a colon (:) at the end. Remember, boss statements in Java have no mark at the end. There is an access specifier `public` listed first. This says, "other classes can see this function." The interactions pane behaves like a different class, so `go()` can be seen by the interactions pane when we create a `Hello` object there. The other new element is the keyword `void`. A function in Java must specify the type of object it returns. If a function returns nothing, its return type *must* be marked `void`. Next you see the line

```
    {
```

This open curly brace says, "This is the beginning of the code for the function `go`." It serves as a delimiter marking the beginning of the function's body. On the next line we see the ungainly command

```
    System.out.println("Hello, World!");
```

The `System.out.println` command puts the string `"Hello, World!"` to standard output and then it adds a newline. The semicolon is required for this statement because it is a worker statement. Try removing the semicolon and recompiling. You will see angry yellow.

```
1 error found:
File: /home/morrison/book/texed/Java/Hello.java  [line: 5]
Error: /home/morrison/book/texed/Java/Hello.java:5: ';' expected
```

A semicolon must be present at the end of all worker statements in Java. It is a mistake to put a semicolon at the end of a boss statement. In Java, the compiler can sometimes fail to notice this and your program will have a strange logic error.

The next line

```
    }
```

signifies the end of the `go` method. Drag the mouse over the opening curly brace for the `go` method and you will get "toothpaste color" that highlights the closing curly brace and the stuff enclosed by the two curly braces. This brace-matching feature is very useful. Do not be shy!

Finally,

```
}
```

ends the class.

In summary, all Java code is packaged into classes. What we have seen here is that we can put functions (which we call *methods*) in classes. The methods placed in classes give objects created from classes behaviors. We shall turn next to looking at Java's type system so we can write a greater variety of methods.

Programming Exercises Add new methods to our `Hello` class with these features.

1. Have a method use `System.out.print()` twice. How is it different from `System.out.println()`?
2. Have a method do this.

   ```
   System.out.printf("<tr><td>%s</td><td>%s</td></tr>",
       16, 256);
   ```

   Experiment with this `printf` construct. Note its similarities to Python's formatting `%` construct.
3. Create the `Hello` class using Python, giving it a `go` method.

1.3 Java's Integer Types

Creating a new class allows you to create new types. Every time you create a class, you are actually extending the Java language. Like all things that are built out of other things, there must be some "atoms" at the bottom. Said atoms are called *primitive types* in Java.

We will begin by discussing Java's four integer types. These are all primitive types. Let us begin by studying these in the interactions pane.

Type	Size	Explanation
`long`	8 bytes	This is the double–wide 8 byte integer type. It stores a value between -9223372036854775808 and 9223372036854775807. These are 64-bit two's complement integers
`int`	4 bytes	This is the standard two's complement 4 byte integer type, and the most commonly used integer type. It stores a value between -2147483648 and 2147483647.
`short`	2 bytes	This is the standard 2 byte integer type. It stores a value between -32768 and 32767 in two's complement notation.
`byte`	1 byte	This is a one–byte integer that stores an integer between -128 and 127 in two's complement notation.

You should note that Python 3 has one integer type and that Python 2 has two: `int` and `long`.

1.3.1 Some Interaction Pane Nuances

When you work in the interactions pane, you are working in a run time environment, as you would be at the Python interactive prompt. Your statements are compiled "live," so if you enter a statement that is nonsensical to Java, you will get a run–time error (red), and you will never see angry yellow. On the dark side, many errors will result in dreaded exploding hearts (remember: dark red ink).

Hit F5 to clear the interactions pane. If there is a messed-up program in your code window, right click on it in the skinny window on the left and select Close. Then hit F5. The interactions pane will be cleared. Type in

```
> int x = 5
```

You are seeing the assignment operator `=` at work here. This works just as it does in Python; you should read it as, "x gets 5" and not "x equals 5." As is true in Python, it is a worker statement. Consequently in a compiled program, it must be ended with a semicolon. The expression on the right–hand side is evaluated and stored into the variable on the left–hand side. Now hit the enter key. You will see this. The reply will be toothpaste-colored.

```
> int x = 5
5
```

If you type a semicolon at the end like this and hit enter, the output will be suppressed. It is no surprise that the assignment `x = 5` returns a 5.

```
> int x = 5;
>
```

To create a variable in Java, you need to specify its type and an identifier (variable name). This is because Java is a it statically compiled language; the types of all variables must be known at compile time. In general a variable is created in a declaration of the form

```
type variableName;
```

You can initialize the variable when you create it like so.

```
type variableName = value;
```

When you create variables inside of methods you should always initialize them, or the compiler will growl at you. You may do any of these things in the interactions pane. Remember, the semicolon will suppress any output. You can see the value held by any variable by typing it at the interactive prompt.

```
> int x = 5;
> x
5
```

Now let us deliberately do something illegal. We will set a variable of type `byte` equal to 675. Watch Java rebel.

```
> byte b = 675
Error: Bad types in assignment
>
```

The error message appears in red type. Again, this occurs because the interactions pane is a run time environment. This would attract the compiler's attention in a compiled program.

1.3.2 Using Java integer types in Java Code

So far we have seen Java's four integer types: `int`, `byte`, `short`, and `long`. To see them in code, begin by creating this file.

```
public class Example
{
    public void go()
    {
    }
}
```

Once you enter the code in the code window, compile and save it. It now does nothing. Now we will create some variables in the method `go` and do some experiments. Modify your code to look like this and compile.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);

    }
}
```

Compile the program. Start an interactions pane session and enter the code
`e = new Example()`

```
> Example e = new Example()
Example@3cb075
```

Since we did not put a semicolon at the end, we see the mysterious artifact `Example@(some gibberish)`. If you noticed the gibberish looks like hex code, you are right. All Java objects can print themselves, what they print by default is not very useful. Later we will learn how to change that. Now let us send the message "`go()`" to our `Example` object `e`.

Recall from the `Hello` class that `System.out.println` puts things to standard output with a newline at the end.

```
> e = new Example()
Example@3cb075
> e.go()
x = 5
```

Inside the `System.out.println()` command, we see the strange sequence `"x = " + 5`. Java has a built–in string type `String`, which is akin to Python's `str`. In Python, you would have written

```
print "x = " + str(x)
```

Java has a feature called "lazy evaluation" for strings. Once Java knows that an expression is to be a string, any other objects concatenated to the expression are automatically converted into strings. That is why you see

```
x = 5
```

printed to `stdout`. Note that Python is very strict in this matter and requires you to explicitly convert objects to string before they can be concatenated to a string.

Now let us add some more code to our `Example` class so we can see how these integer types work together.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
```

```
        byte b = x;
        System.out.println("b = " + b);
    }
}
```

Now we compile our masterpiece and we get angry yellow and these scoldings from Java.

```
1 error found:
File: Example.java  [line: 7]
Error: Example.java:7: possible loss of precision
found   : int
required: byte
```

Indeed, line 7 contains the offending code

```
byte b = x;
```

To fully understand what is happening, let's do a quick comparison with Python and explain a few differences with Java.

Types: Java vs. Python Python is a strongly, dynamically typed language. This means that objects are aware of their type and that decisions about type are made at run time. Variables in Python are merely names; they have no type.

In contrast, Java is a strongly, statically typed language. In the symbol table, Java keeps the each variable's name, the object the variable points at and the variable's type. Types are assigned to variables at *compile time*. In Python a variable may point at an object of any type. In Java, variables have type and may only point at objects of their own type.

Now let's return to the example. The value being pointed at by `x` is 5. This is a legitimate value for a variable of type `byte`. However, `x` is an integer variable and knows it is an integer. The variable `b` is a `byte` and it is aware of its `byte`ness. When you perform the assignment

```
b = x;
```

Java sees that `x` is an integer. An integer is a bigger variable type than a byte. The variable `b` says, "How dare you try to stuff that 4–byte integer into my one–byte capacity!" Java responds chivalrously to this plea and the compiler calls the proceedings to a halt.

In this case, you can cast a variable just as you did in Python. Modify the program as follows to cast the integer `x` to a `byte`.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

Your program will now run happily.

```
> e = new Example();
> e.go()
x = 5
b = 5
>
```

Now let's play with fire. Change the value you assign `x` to 675.

```
public class Example
{
    public void go()
    {
        int x = 675;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

This compiles very happily. It runs, too!

```
> e = new Example()
Example@afae4a
> e.go()
x = 675
b = -93
>
```

Whoa! When casting, you can see that the doctrine of *caveat emptor* applies. If we depended upon the value of `b` for anything critical, we can see we might

be headed for a nasty logic error in our code. When you cast, you are telling Java, "I know what I am doing." With that right, comes the responsibility for dealing with the consequences.

Notice that you are casting from a larger type to a smaller type. This is a type of *downcasting*, and it can indeed cause errors that will leave you downcast. Since we discussed downcasting, let's look at the idea of upcasting that should easily spring to mind. For this purpose, we have created a new program that upcasts a byte to an integer

```
public class UpCast
{
    public void go()
    {
        byte b = 122;
        System.out.println("b = " + b);
        int x =  b;
        System.out.println("x = " + x);

    }
}
```

This compiles and runs without comment.

```
> u = new UpCast();
> u.go()
b = 122
x = 122
>
```

The four integer types are just four integers with different sizes. Be careful if casting down, as you can encounter problems. Upcasting occurs without comment. Think of this situation like a home relocation. Moving into a smaller house can be difficult. Moving into a larger one (theoretically) presents no problem with accommodating your stuff.

Important! If you use the arithmetic operators `+`, `-`, `*` or `/` on the short integral types byte and short, they are automatically upcast to integers as are their results.

Finally let us discuss the problem of type overflow and "doughnutting." Since the `byte` type is the smallest integer type, we will demonstrate these phenomena on this type. Observe that the binary operators `+`, `-`, `* /`, and `%` work in java just as they do in Python 2.x on integer types. Also we have the compound assignment operators such as += which work exactly as they do in Python.

Open the interactions pane and run these commands. By saying `int b = 2147483647`, we guarantee that Java will regard `b` as a regular integer.

```
> byte b = 2147483647
> b += 1            This is b = b + 1.
-2147483648
>                   Uh oh.
```

The last command `b += 1` triggered an unexpected result. This phenomenon called *type overflow*. As you saw in the table at the beginning of the section, a byte can only hold values between -2147483648 and 2147483647. Adding 1 to 2147483647 yields -128; this phenomenon is called *doughnutting*. It is an artifact of the workings of two's complement notation. You can see that this occurs in C/C++ as well.

This is caused by the fact that integers in Java are stored in two's complement notation. See the section in the Cyberdent in *Computing in Python* to learn why this happens.

1.4 The Rest of Java's Primitive Types

The table below shows the rest of Java's primitive types. We see there are eight primitive types, four of which are integer types.

Type	Size	Explanation
`boolean`	1 byte	This is just like Python's `bool` type. It holds a `true` or `false`. Notice that the boolean constants `true` and `false` are not capitalized as they are in Python.
`float`	4 bytes	This is an IEEE 754 floating point number It stores a value between -3.4028235E38 and 3.4028235E38.
`double`	8 bytes	This is an IEEE 754 double precision number. It stores a value between -1.7976931348623157E308 and 1.7976931348623157E308. This is the same as Python's `float` type. It is the type we will use for representing floating–point decimal numbers.
`char`	2 byte	This is a two–byte Unicode character. In contrast to Python, Java has a separate character type.

1.4.1 The `boolean` Type

Let us now explore booleans. Java has three boolean operations which we will show in a table

Operator	Role	Explanation
`&&`	and	This is the boolean operator $\wedge$. It is a binary infix operator and the usage is `P && Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `true`, the expression `Q` is ignored.
`||`	or	This is the boolean operator $\vee$. It is a binary infix operator and the usage is `P || Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `false`, the expression `Q` is ignored.
!	not	This negates a boolean expression. It is a unary prefix operator. Be careful to use parentheses to enforce your intent!

Hand–in–hand with booleans go the *relational operators*. These work just as they do in Python on primitive types. The operator `==` checks for equality, `!=` checks for inequality and the operators `<`, `>`, `<=` and `>=` act as expected on the various primitive types. Numbers (integer types and floating point types) have their usual orderings. Characters are ordered by their ASCII values. It is an error to use inequality comparison operators on boolean expressions.

Now let us do a little interactive session to see all this at work. You are encouraged to experiment on your own as well and to try to break things so you better understand them.

```
> 5 < 7
true
> 5 + 6 == 11             == tests for equality
true
> !( 4 < 5)               ! is "not"
false
> (2 < 3) && (1 + 2 == 5) and at work
false
> (2 < 3) || (1 + 2 == 5) "or" at work
true
> 100 % 7 == 4            % is just like Python!
false
>
```

1.4.2 Floating–Point Types

When dealing with floating–point numbers we will only use the `double` type. Do not test floating–point numbers for equality or inequality. Since they are stored inexactly in memory, comparing them exactly is a dangerous hit–or–miss proposition. Instead, you can check and see if two floating–point numbers are within some tolerance of one another. Here is a little lesson for the impudent to ponder. Be chastened!

```
> x = 1.0/2.0
0.5
> x
0.5
> x = 997.0/1994.0
0.5
> x == y
false
>
```

All integer types will upcast to the `double` type. You can also downcast doubles to integer types; you should experiment and see what kinds of truncation occur. You should experiment with this in the interactions pane. Remember, downcasting can be hazardous and ... leave you downcast.

1.4.3 The `char` type

In Python, characters are recorded as one–character strings. Java works differently and is more like C/C++ in this regard. It has a separate type for characters, `char`.

Recall that characters are really just bytes. Java uses the *unicode* scheme for encoding characters. All unicode characters are two bytes. The ASCII characters are prepended with a zero byte to make them into unicode characters. You can learn more about unicode at `\protect\protect\protect\edefOT1{OT1}\ let\enc@update\relax\protect\edefcmr{cmr}\protect\edefm{m}\protect\ edefn{n}\protect\xdef\OT1/cmr/m/it/10{\OT1/cmr/m/n/10}\OT1/cmr/m/it/ 10\size@update\enc@update\ignorespaces\relax\protect\relax\protect\ edefcmr{cmtt}\protect\xdef\OT1/cmr/m/it/10{\OT1/cmr/m/n/10}\OT1/cmr/ m/it/10\size@update\enc@updatehttp://www.unicode.org`.

Integers can be cast to characters, and the unicode value of that character will appear.

Here is a sample interactive session. Notice that the integer 945 in unicode translates into the Greek letter α.

```
> (char) 65
```

```
'A'
> (char) 97
'a'
> (char)945
'α'
> (char)946
'β'
>
```

Similarly, you can cast an integer to a character to determine its ASCII (or unicode) value.

The relational operators may be used on characters. Just remember that characters are ordered by their Unicode values. The numerical value for the 8 bit characters are the same in Unicode. Unicode characters are two bytes; all of the 8 bit characters begin with the byte `00000000`.

1.5 More Java Class Examples

Now let us develop more examples of Java classes. Since we have the primitive types in hand, we have some grist for producing useful and realistic examples. Let us recall the basics. All Java code must be enclosed in a class. So far, we have seen that classes contain methods, which behave somewhat like Python functions.

Open DrJava and place the following code in the code window; this produces an empty class named `MyMethods`.

```
public class MyMethods
{
}
```

When you are done, hit the F5 button and save the resulting file as `MyMethods.java` in the directory you are using for Java programs. DrJava should automatically place this name in the Save dialog box. In the bottom pane, the Compiler Output tab will be activated and you will see it has the text "Compilation Completed" in it. If you get any angry yellow, correct any errors. Remember, we never want to stray far from a compiling, running program.

So far, our program does nothing. Now let us give it a method.

```
public class MyMethods
{
    public double square(double x)
    {
```

```
        return x*x;
    }
}
```

Compile this program. Once it compiles, enter the following in the interactions pane.

```
> MyMethods m = new MyMethods();
```

Recall that `new` tells Java, "make a new `MyMethods` object." Furthermore, we have assigned this to the variable `m`. Now type `m.getClass()` and see `m`'s class.

```
> m.getClass()
class MyMethods
```

Every Java object is born with a `getClass()` method. It behaves much like Python's `type()` function. For any object, it tells you the class that created the object. In this case, `m` is an instance of the `MyMethods` class, so `m.getClass()` returns `class MyMethods`.

We endowed our class with a `square` method; here we call it.

```
> m.square(5)
25.0
```

The name of the method leaves us no surprise as to its result. Now let us look inside the method and learn its anatomy.

```
    public double square(double x)
    {
        return x*x;
    }
```

In Python, you would make this function inside of a class by doing the following.

```
class MyMethods:
    def square(self, x):
        return x*x
```

in both the top line is called the *function header*. Notice that in Python, you must use the `self` variable in the argument list for any methods you create. Python functions begin with the `def` statement; this tells Python we are defining a function. Java methods begin with an *access specifier* and then a *return type*. The access specifier controls visibility of the method. The access specifier `public`

says that the `square` method is visible outside of the class `MyMethods`. The return type says that the `square` method will return a datum of type `double`.

In both Python and Java, the next thing you see is the function's name, which we have made `square`. The rules for naming methods in Java are the same as those for naming variables. To review, an identifier name may start with an alpha character or an underscore. The remaining characters may be numerals, alpha characters or underscores.

Inside the parentheses, we see different things in Java and Python. In Python, we see a lone `x`. In Java, we see `double x`. Since Java is statically typed, it requires all arguments to specify the type of the argument as well as the argument's name. This restriction is enforced *at compile time*. In contrast, Python makes these and all decisions at run time.

In general every Java method's header has the following form.

```
returnType functionName(type1 arg1, type2 arg2,  ...  , typen argn)
```

The list

```
[type1, type2, ... typen]
```

of a Java method is called the method's *signature*, or "sig" for short. Notice that the argument names are not a part of the signature of a method. Remember, such names are really just dummy placeholders. Methods in Java may have zero or more arguments, just as functions and methods do in Python.

Try entering `m.square('a')` in the interactions pane.

```
> m.square('a')
Error: No 'square' method in 'MyMethods' with arguments: (char)
>
```

If we were compiling a program we would get angry yellow. Since the interactions pane is a run-time environment, Java objects by saying that a character is an illegal argument for your method `square`. Java methods have type restrictions in their arguments. Users who attempt to pass data of illegal type to these methods are rewarded with compiler errors. This sort of protection is a two–edged sword. Add this method to your `MyMethods` class.

```
public double dublin(double x)
{
    return x*2;
}
```

Now let us do a little experiment.

```
> m = new MyMethods();
> m.dublin(5)
10.0
> m.dublin("string")
Error: No 'dublin' method in 'MyMethods'
with arguments: (java.lang.String)
>
```

What have we seen? The `dublin` method belonging to the `MyMethods` class will accept integer types, which upcast to doubles, or doubles, but it rejects a string. (More about Java's string type later)

We will now write the analogous function in Python; notice what happens. Place this Python code in a file named `method.py`.

```
def dublin(x):
    return x*2
x = 5
print "dublin(" + str(x) + ") = " + str(dublin(x))
x = "string"
print "dublin(" + str(x) + ") = " + str(dublin(x))
```

Now let us run it.

```
$ python method.py
dublin(5) = 10
dublin(string) = stringstring
$
```

Python makes decisions about objects at run time. The call `dublin(5)` is fine because it makes sense to take the number 5 and multiply it by the number 2. The call `dublin("string")` is fine for two reasons. First, multiplication of a string by an integer yields repetition, so the return statement in the function `dublin` makes sense to Python at run time. Secondly, variables in Python have no type, so there is no type restriction in `dublin`'s argument list. You will notice that static typing makes the business of methods more restrictive. However, compiler errors are better than run time errors, which can conceal ugly errors in your program's logic and which can cause surprisingly unappealing behavior from your function.

Just as in Python, you may have functions that produce no output and whose action is all side–effect. To do this, just use the `void` return type, as we did in the `Hello` class.

Chapter 2

Java Objects

2.0 Java Object Types

We have seen that Java has eight primitive types: the four integer types, the floating–point types `double` and `float`, the `boolean` type and the `char` type.

Python has a string type; you might ask why we have not given much emphasis to string in Java yet. This is because the string type in Java is *not* a primitive type. It is an example of a Java *object* or *class* type. This distinction is extremely important, because there are significant differences in the behaviors of the two types. We will make a close study of the Java string class and compare its behavior to the string type in Python.

You will see that Java strings have many capabilities. You can slice them as you can Python strings, they know their lengths, and you have access to all characters. You will learn how to use the Java API guide to learn more about any class's capabilities, including those of `String`.

2.1 Java Strings as a Model for Java Objects

We will first turn our attention to the Java String type. Java handles strings in a manner similar to that of Python. Strings in Java are immutable. Java has an enormous *standard library* containing thousands of classes. The string type is a part of this vast library, an it is implemented in a class called `String`.

Because strings are so endemic in computing, the creators of Java gave Java's string type some features atypical of Java classes, which we shall point out as we progress.

Let us begin by working interactively. Here we see how to read individual

characters in a string by their indices.

```
> String x = "abcdefghijklmnopqrstuvwxyz"
"abcdefghijklmnopqrstuvwxyz"
> x.charAt(0)
'a'
> x[0]
Error: 'java.lang.String' is not an array
> x.charAt(25)
'z'
> x.length()
26
>
```

Now let us deconstruct all of this. Strings in Java enjoy an exalted position. The line

```
> x = "abcdefghijklmnopqrstuvwxyz"
```

makes a *tacit* call to `new` and it creates a new `String` object in memory. Only a few other Java class enjoys the privilege of making tacit calls to `new`; these are the *wrapper classes*.

Each primitive type has a wrapper class; for example, `int` has the wrapper class `Integer`. You can create an `Integer` either by saying

```
Integer n = 5;
```

or by saying

```
Integer n = new Integer(5);
```

This tacit call to `new` is enabled by a feature called *autoboxing*. We will meet the wrapper classes in full later.

Coming back to our main thread, you can create a string using `new` as well.

```
String x = new String("abcdefghijklmnopqrstuvwxyz");
```

Here we made an *explicit* call to `new`. This is not done very often in practice, as it is excessively verbose, and it can create duplicate copies of immutable objects.

Access to characters is granted via the `charAt` string method. The expression

```
> x.charAt(25)
```

can be read as "x's character at index 25." Just as in Python, the dot (.) indicates the genitive case. The nastygram

```
Error:  'java.lang.String' is not an array
```

arises because the square–bracket operator, which exists in Python, is only used to extract array entries in Java. Arrays are a Java data structure, which we will learn about Java soon. Finally, we see that a string knows its length; to get it we invoke the `length()` method.

`String` has another atypical feature not found in other Java classes. The operators `+` and `+=` are implemented for Strings. The `+` operator concatenates two strings, just as it does in Python. The `+=` operator works for strings just as it does in Python.

```
> x = "abc"
"abc"
> x += "def"
"abcdef"
>
```

Note, however, that the string `"abc"` is not changed. It is orphaned and the String variable `x` now points at `"abcdef"`. The mechanism of orphaning objects in Java works much as it does in Python. Both Python and Java are garbage-collected languages.

2.1.1 But is there More?

You might be asking now, "Can I learn more about the Java String class?" Fortunately, the answer is "yes;" it is to found in the Java API (Applications Programming Interface) guide. This is a freely available online comprehensive encyclopaedia of all of the classes in the Java Standard Library.

In the beginning you will see much that you will not understand. This is OK: we will learn how to dig for the information we need. As your knowledge of Java progresses, more of the things posted on these pages will be understandable to you. We recommend that you download this documentation and place it on your computer so you can use it any time. It is available at `http://http://download.oracle.com/javase/6/docs/api/`, which wll give you instructions for installing it.

Bring up the API guide and you will see that the page is divided into three frames. On the left, is a skinny frame that is further divided into two smaller frames.

The frame on the top–left is called the *package frame*. The Java Standard Library is organized into units called *packages*. Packages form a hierarchical

structure that behaves much like your file system. The principal package in Java is called `java.lang`; the `String` class resides in this package. If you click on `java.lang` in the package window, you can seee the other classes in this package in the frame just below the package frame. This is called the *class frame*.

If you click on a class in the class frame, its documentation will appear on the right in the *main frame*.

If you click on a package in the package frame, only classes belonging to that package will be displayed in the class frame. When you are working, this can help narrow your search. You will use this feature as you become more familiar with the standard library's hierarchy.

Let us find the documentation for the `String` class. In your browser, do a Find on Page. In the search window, type a space, then String (try omitting the space and you will see the method to this madness). You will find the String class in the class window. You can also find it by navigating downward in the class frame. Find it either way and click on its link.

You will see lots of incomprehensible incantations. Do not be worried. Right near the top you will see

> java.lang
> class String

This tells you that the class String is part of the package `java.lang`, which is the "core" of the Java language. If you click on the package `java.lang` in the package window, you can see all of the other classes in `java.lang`.

Now let us ferret out the usable goodies. Scroll down to the area that has the heading "Method Summary." Here you will see a complete listing of all `String` methods. It is a long list, but we will focus on some methods that will be familiar because of Python.

At the top of the list, you can see the now–familiar `charAt` method. When reading the method summary, note that the first column tells the return type of the method. The return type of `charAt` is `char`. Next you see

```
charAt(int index)
```

The word `charAt` is a link. We shall click on it in a moment. The `int index` gives the argument list of the method `charAt`. What we see here is that `charAt` accepts an integer argument. The guide then says "Returns the char value at the specified index." This gives us some pretty good information about `charAt`. We know it is used for fetching the characters from a string. It needs to know which character you wish to fetch; this is specified by the (integer) index of the character.

Now click on the link; you will go to the *method detail* for `charAt`. Right after the heading it says

```
public char charAt(int index)
```

This is the method header that appears in the actual `String` class. It then goes on to say the following.

> Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

The following paragraph

> If the `char` value specified by the index is a surrogate, the surrogate value is returned.

looks pretty mysterious, so we will ignore it for now. The **Parameters:** heading describes the argument list and the **Returns:** heading describes the return value. There are no surprises here.

What is interesting is the `Throws:` heading. This describes run time errors that can be caused by misuse of this method. These errors are not found by the compiler. If you trigger one, your program dies gracelessly and you get a dreaded "exploding heart" of red text. You have observed similar tantrums thrown by Python when you give it an index that is out of bounds in a string, list or tuple.

We shall use this web page in the next sections so keep it open. First it will be necessary to understand a fundamental difference between Java object types and Java primitive types.

Programming Exercises Write a class called `Exercises11` and place the following methods in it.

1. Write the method

   ```
   public boolean isASubstringOf(String quarry, String field)
   {
   }
   ```

   It should return true when `quarry` is a contiguous substring of `field`. (Think Python `in` construct.)

2. Suppose you have declared the string `cat` as follows.

   ```
   String cat = "abcdefghijklmonopqrstuvwxyz";
   ```

Find at least two ways in the API guide to obtain the string `"xyz"`. You may use no string literals (stuff inside of "..."), just methods applied to the object `cat`. There are at least three ways. Can you find them all?

3. The Python repetition operator `*`, which takes as operands a string and an integer, and which repeats the string the integer number of times does not work in Java. Write a method

   ```
   String repeat(String s, int n)
   ```

   that replicates the action of the Python repetition operator. And yes, there is recursion in Java.

2.2 Primitive vs. Object: A Case of `equals` Rights

We will study the equality of string objects. A big surprise lies ahead so pay close attention. Create this interactive session in Python. All is reassuringly familiar.

```
>>> good = "yankees"
>>> evil = "redsox"
>>> copy = "yankees"
>>> good == copy
True
>>> good == evil
False
```

No surprises greet us here. Now let us try the same stuff in Java.

```
> good = "yankees"
"yankees"
> evil = "redsox"
"redsox"
> copy = "yankees"
"yankees"
> good == evil
false
> good == good
true
> good == copy
false
>
```

Beelzebub! Some evil conspiracy appears to be afoot! Despite the fact that both `good` and `copy` point to a common value of `"yankees"`, the equality test

returns a `false`. Now we need to take a look under the hood and see what is happening.

First of all, let's repeat this experiment using integers.

```
> Good = 5;
> Evil = 4;
> Copy = 5;
> Good == Evil
false
> Good == Good
true
> Good == Copy
true
```

This seems to be at odds with our result with strings. This phenomenon occurs because primitive and class types work differently. A primitive type *points directly at its datum*. When you use `==` on two variables, you are asking if they point to the same value.

Strings do not point directly at their datum; this is true of all object types in Java. What a java object holds is a *pointer*, i.e. a memory address where the string is stored. In Python, objects *never* point directly at thier datum. Python types such as `bool`, `float` and `int` are actually immutible objects. This phenomenon is a major difference between Python and Java. Python has no primitive types.

We saw `good == good` evaluate to true because `good` points to the same actual object in memory as itself. We saw `good == copy` evaluate to false, because `good` and `copy` point to separate copies of the string `"yankees"` stored in memory. Therefore the test for equality evaluates to false.

Recall we said that objects have behavior and identity. The `==` operator is a test for identity. It checks if two objects are in fact one and the same. This behavior is identical to that of the Python `is` keyword, which checks for equality of identity.

What do we do about the equality of strings? Fortunately, the `equals` method comes to the rescue.

```
> good.equals(good)
true
> good.equals(evil)
false
> good.equals(copy)
true
>
```

Ah, paradise restored... Just remember to use the `equals` method to check for equality of strings. This method has a close friend `equalsIgnoreCase` that will do a case-insensitive check for equality of two strings. These comments also apply to the inequality operator `!=`. This operator checks for inequality of identity. To check and see if two strings have unequal values use ! in conjunction with `equals`. Here is an example

```
> !(good.equals(copy))
false
> !(good.equals(evil))
true
>
```

Finally, notice that Python compares strings lexicographically according to Unicode value by using inequality comparison operators. These do not work in Java. It makes no sense to compare memory addresses. However, the string class has the method

```
int compareTo(String anotherString)
```

We show it at work here.

```
> little = "aardvark"
"aardvark"
> big = "zebra"
"zebra"
> little <= big
Error: Bad type in relational expression
> little.compareTo(big)
-25
> little.compareTo(big) < 0
true
> little.compareTo(big) == 0
false
> little.compareTo(big) > 0
false
>
```

You may be surprised it returns an integer. However, alphabetical string examples can be done as in the example presented here. This method's sibling method, `compareToIgnoreCase` that does case–insensitive comparisons and works pretty much the same way.

2.2.1 Aliasing

Consider the following interactive session.

```
> smith = "granny"
"granny"
> jones = smith;
> smith == jones
true
>
```

Here we performed an assignment, `jones = smith`. What happens in an assignment is that the right–hand side is evaluated and then stored in the left. Remember, the string `smith` points at a memory address describing the location where the string `"granny"` is actually stored in memory. So, this memory address is given to `jones`; both `jones` and `smith` hold the same memory address and therefore both point at the one copy of `"granny"` that we created in memory.

This situation is called *aliasing*. Since strings are immutable, aliasing can cause no harm. We saw in the Python book that aliasing can create surprises. First we will need to explore a property of objects we have omitted heretofore in our discussion: *state*.

The state of a string is given by the characters it contains. How these are stored is not now known to us, and we really do not need to know or care. We shall tour the rest of the string class in the Java API guide, then turn to the matter of state.

2.3 More Java `String` Methods

Python's slicing facilities are implemented in Java using `substring`. Here is an example of `substring` at work.

```
> x = "abcdefghijklmnopqrstuvwxyz"
> x.substring(5)
"fghijklmnopqrstuvwxyz"
> x.substring(3,5)
"de"
> x.substring(0,5)
"abcde"
> x
"abcdefghijklmnopqrstuvwxyz"
```

Notice that the original string is not modified by any of these calls; copies are the advertised items are returned by these calls. The `endsWith` method seems pretty self–explanatory.

```
> x.endsWith("xyz")
```

```
true
> x.endsWith("XYZ")
false
```

The `indexOf` method allows you to search a string for a character or a substring. In all cases, it returns a -1 if the string or character you are seeking is not present.

```
> x.indexOf('a')
0
> x.indexOf('z')
25
> x.indexOf('A')
-1
> x.indexOf("bc")
1
```

You can pass an optional second argument to the `indexOf` method to tell is to start its search at a specified index. For example, since the only instance of the character `'a'` in the string `x` is at index 0, we have

```
> x.indexOf('a', 1)
-1
```

You are encouraged to further explore the `String` API. It contains many useful methods that make strings a useful an powerful programming tool. The programming exercises at the end of this section will give you an opportunity to do this.

2.4 Java Classes Know Things: State

So far, we have seen that objects have identity and that they have behavior, which is reflected by a class's methods.

We then saw that a string "knows" the character sequence it contains. We do not know how that sequence is stored, and we do not need to know that. The character sequence held by a string is reflective of its *state*. The state of an object is what an object "knows." Observe that the outcome of a Java method on an object can, and often does, depend upon its state.

To give you a look behind the scenes, we shall now produce a simple class which will produce objects having state, identity and behavior. To do this, it will necessary to introduce some new ideas in Java, the *constructor* and *method overloading*.

Place the following code in the Java code window, compile and save it in a file named `Point.java`. We are going to create a simple class for representing points in the plane with integer coördinates.

```
public class Point
{
}
```

What does such a point need to know? It needs to know its `x` and `y` coördinates. Here is how to make it aware of these.

```
public class Point
{
    private int x;
    private int y;
}
```

You will see a new keyword: `private`. This says that the variables `x` and `y` are not visible outside of the class. These variables are called *instance variables* or *state variables*. We shall see that they specify the state of a `Point`.

Why this excessive modesty? Have you ever bought some electronic trinket, turned it upside–down and seen "No user serviceable parts inside" emblazoned on the bottom? The product–liability lawyers of the trinket's manufacturer figure that an ignorant user might bring harm to himself whilst fiddling with the entrails of his device. Said fiddling could result in a monster lawsuit that leaves the princplles of the manufacturer living in penury.

Likewise, we want to protect the integrity of our class; we will not allow the user of our class to monkey with the internal elements of our program. We will permit the client programmer access to these elements by creating methods that give access. This is a hard-and-fast rule of Java programming: *Always declare your state variables private.*

Now compile your class. Let's make an instance of this class and deliberately get in trouble.

```
> p = new Point()
Point@1db9852
> p.x
IllegalAccessException:
(then a bunch of nasty messages in red)
>
```

We have debarred ourselves from having any access to the state variables of an instance of the `Point` class. This makes our class pretty useless. How do we get out of this pickle?

2.4.1 Quick! Call the OBGYN!

Clearly a `Point` needs help initializing its coördinates. For this purpose we use a special method called a *constructor*. A constructor has no return type. When the constructor is finished, all state variables should be initialized. Constructors are OBGYNs: they oversee the birth of objects. The constructor for a class must have the same name as the class. In fact, only constructors in a class may have the same name as the class. We now write a constructor for our `Point` class.

```
public class Point
{
    private int x;
    private int y;
    public Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
}
```

Now compile your class. To make the a point at (3,4), call the constructor by using `new`. The `new` keyword calls the class's constructor and oversees the birth of an object.

```
> p = new Point(3,4)
Point@3cb075
> q = new Point()
NoSuchMethodException: constructor Point()
>
```

The `Point p` is storing the point (3,4). Remember, the variable `p` itself only stored a memory address. The point (3,4) is stored at that address.

One other thing we see is that once we create a constructor the *default constructor*, which has an empty signature, no longer exists.

Note the similarity of this process to the Python Point class we created earler. The Python `__init__` method behaves much like a Java constructor; it is called every time a new Python object of type `Point` is created.

```
import math
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

```
p = Point()
print ("p = ({0}, {1})".format(p.x, p.y))
q = Point(3,4)
print ("q = ({0}, {1})".format(q.x, q.y))
```

Now go back to the `String` class in the API guide. Scroll down to the constructor summary; this has a blue header on it and it is just above the method summary. You will see that the string class has many constructors. How is this possible? We faked in in Python by using default arguments. Can we do this for our point class in Jav?

Happily, the answer is "yes".

2.4.2 Method and Constructor Overloading

Recall that the signature of a method is an ordered list of the types of its arguments. Java supports *method overloading*: you may have several methods bearing the same name, provided they have different signatures. This is why you see several versions of `indexOf` in the `String` class. Java resolves the ambiguity caused by overloading at compile time by looking at the types of arguments given in the signature. It looks for the method with the right signature and it then calls it.

Notice that the static typing of Java allows it to support method overloading. Python has a feature that looks like method overloading. It allows arguments to have default values, so it looks like it has method overloading. Here is another example of Python default arguments at work.

```
def f(x = 0, y = 0, z = 0):
return x + y + z
print "f() = ", f()
print "f(3) = ", f(3)
print "f(3, 4) = ", f(3, 4)
print "f(3, 4, 5) = ", f(3, 4, 5)
```

```
$ python overload.py
f() =  0
f(3) =  3
f(3, 4) =  7
f(3, 4, 5) =  12
$
```

You can use principle on constructors, too. Let us now go back to our `Point` class. We will make the default constructor (sensibly enough) initialize our point to the origin.

```
public class Point
{
    private int x;
    private int y;
    public Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    public Point()
    {
        x = 0;
        y = 0;
    }
}
```

Compile this class. Then type in this interactive session.

```
> p = new Point(3,4)
Point@175ade6
> q = new Point()
Point@a7c45e
>
```

Voila! The default constructor is now working.

2.4.3 Get a load of `this`

The eleventh commandment reads, "Thou shalt not maintain duplicate code." This sounds Draconian, but it is for reasons of convenience and sanity. If you want to modify your program, you want to do the modifications in ONE place. Having duplicate code forces you to ferret out every copy of a duplicated piece of code you wish to modify. You should strive to avoid this.

One way to avoid it is to write separate methods to perform tasks you do frequently. Here, however, we are looking at our constructor. You see duplicate code in the constructors. To eliminate it, you may use the `this` keyword to call one constructor from another. We shall apply `this` here.

```
public class Point
{
    private int x;
    private int y;
    public Point(int _x, int _y)
    {
```

```
        x = _x;
        y = _y;
    }
    public Point()
    {
        this(0,0);
    }
}
```

2.4.4 Now Let Us Make this Class DO Something

So far, our `Point` class is devoid of features. We can create points, but we cannot see what their coördinates are. Now we shall provide *accessor methods* that give access to the cöordinates. While we are in here we will also write a special method called `toString`, which will allow our points to print nicely to the screen.

First we create the accessor methods. Here is how they should work.

```
> p = new Point(3,4);
> p.getX()
3
> p.getY()
4
> q = new Point();
> q.getX()
0
> q.getY()
0
>
```

Making them is easy. Just add these methods to your point class.

```
    public int getX()
    {
        return x;
    }
    public int getY()
    {
        return y;
    }
```

These accessor or "getter" methods allow the user of your class to see the coördinates of your point but the user cannot use the getter methods to change

the state of the point. So far, our point class is immutable. There is no way to change its state variables, only a way to read their values.

To get your points to print nicely, create a `toString` method. Its header *must* be

```
public String toString()
```

In this method we will return a string representation for a point. Place this method inside of your `Point` class.

```
public String toString()
{
    return "("+ x + ", " + y + ")";
}
```

Compile and run. The `toString()` method of an object's class is called *automatically* whenever you print an object. Every Java object is born with a `toString` method. We saw that this built–in method for our point class was basically useless. By implementing the `toString` method in our class, we are customizing it for our purposes. Here we see our nice representation of a `Point`.

```
> p = new Point(3,4)
(3, 4)
> System.out.println(p)
(3, 4)
>
```

You will see that many classes in the standard libarary customize this method.

Now let us write a method that allows us to compute the distance between two points. To do this we will need to calculate a square-root. Fortunately, Java has a scientific calculator. Go to the API guide and look up the class `Math`. To use a `Math` function, just prepend its name with `Math.`; for example `Math.sqrt(5)` computes $\sqrt{5}$. Many of the names of these functions are the same as they are in Python's `math` library and in and C/C++'s `cmath` and `math.h` libraries.

Now add this method to your class. You will see that it is just carrying out the distance formula between your point `(x,y)` and the point `q`.

```
public double distanceTo(Point q)
{
    return Math.sqrt( (x - q.x)*(x - q.x) + (y - q.y)*(y - q.y));
}
```

Compile and run.

```
> origin = new Point()
(0, 0)
> p = new Point(5,12);
> origin.distanceTo(p)
13.0
> p.distanceTo(origin)
13.0
>
```

Programming Exercises Here is a chance to try out some new terriitory in Python. Python has special methods called *hooks* that do speical jobs. We have met the `init` hook; all hooks are surrounded by double-underscores.

1. The Python hook `__str__` tells Python to represent a Python object as a string. Its method header is

   ```
   def __str__(self):
   ```

   Make a method for the Python Point class that represents a Point as a string.
2. The Python hook `__eq__` can check for equality of objects. You can cause to points to be compared using `==` with this hook. Its header is

   ```
   def __eq__(self):
   ```

   Implement this for our Python `Point` class.

2.4.5 Who am I?

In Java there are two programming roles: that of the class developer and that of the client programmer. You assume both roles, as all code in Java lives inside of classes. You are the class developer of the class you are writing, and the client programmer of the classes you are using. Any nontrivial Java program involves at least two classes, the class itself and often the class `String` or `System.out`. In practice, as you produce Java classes, you will often use several different classes that you have produced or from the Java Standard Library.

So while we are creating the point class, we should think of ourselves as *being* `Point`s. A `Point`s knows its coördinates. Since you are a point when programming in the class `Point`, you have access to your private variables. You also have access to the private variables of any instance of class `Point`. This is why in the `distanceTo` method, we could use `q.x` and `q.y`.

In the last interactive session we made two points with the calls

```
origin = new Point();
p = new Point(5,12)
```

This resulted in the creation of two points. The call

```
p.distanceTo(origin)
```

returned 13.0. What is says is "Call `p`'s `distanceTo` method using the argument `origin`." In this case, you should think of "you" as `p`. The point `origin` is playing the role of the point `q` in the method header. Likewise, the call

```
origin.distanceTo(p)
```

is calling `p`'s distance to `origin`. In the first case, "I" is `origin`, in the second, "I" is `p`.

2.4.6 Mutator Methods

So far, all of our class methods have only looked at, but have not changed, the state of a point object. Now we will make our points mutable. To this end, create two "setter" methods and place them in your `Point` class.

```
    public void setX(int a)
    {
        x = a;
    }
    public void setY(int b)
    {
        y = b;
    }
```

Now compile and type in the following.

```
> p = new Point()
(0, 0)
> p.setX(5)
> p
(5, 0)
> p.setY(12)
> p
(5, 12)
>
```

Our point class is now mutable: We are now giving client programmers permission to reset each of the coördinates. These new methods are called "mutator" methods, because they change the state of a `Point` object. Instances of our

`Point` class are mutable, much as are Python lists. Mutabliity can be convenient, but it can be dangerous, too. Watch us get an ugly little surprise from aliasing.

To this end, let us continue the interactive session we started above.

```
> q = p;
> q
(5, 12)
> q.setX(0)
> p
(0, 12)
> q
(0, 12)
>
```

Both `p` and `q` point at the same object in memory, which is initially storing the point `(5,12)`. Now we say, "`q`, set the x–coördinate of the point you are pointing at to 0. Well, `p` happens to be pointing at precisely the same object. In this case `p` and `q` are aliases of one another. If you call a mutator method on either object, it changes the value pointed at by the other object!

If we wanted `p` and `q` to be independent copies of one another, a different procedure is required. Let us now create a method called `clone`, which will return an independent copy of a point.

```
public Point clone()
{
    return new Point(x,y);
}
```

Hit F5, compile, and you will have a fresh interactive session. Now we will test–drive our new `clone` method. We will make a point `p`, an alias for the point `alias`, and a copy of the point `copy`.

```
> p = new Point(3,4)
(3, 4)
> alias = p
(3, 4)
> copy = p.clone();
> p
(3, 4)
```

Continuing, let us check all possible equalities.

```
> p == alias
```

```
true
> copy == alias
false
> p == copy
false
```

We can see that `p` and `q` are in fact aliases the same object, but that `alias` is not synonymous with either `p` or `q`.

```
> p
(3, 4)
> alias
(3, 4)
> copy
(3, 4)
```

All three point at a point stored in memory that is `(3,4)`. Now let us call the mutator `setX` on `p`; we shall then inspect all three.

```
> p.setX(0)
> p
(0, 4)
> alias
(0, 4)
> copy
(3, 4)
```

The object pointeed to by both `p` and `q` was changed to `(0,4)`. The copy, however, was untouched.

Look at the body of the `clone` method. It says

```
return new Point(x,y);
```

This tells Java to make an entirely new point with coördinates `x` and `y`. The call to `new` causes the constructor to spring into action and stamp out a fresh, new `Point`.

2.5 The Scope of Java Variables

In this section, we shall describe the lifetime and visiblity of Java varibles. The rules differ somewhat from Python, and you will need to be aware of these differences to avoid unpleasant surprises.

There are two kinds of variables in Java, state variables and *local* variables. Local variables are variables created inside of any method in Java. State variables are visible anywhere in a class. Where they are declared in a class is immaterial. We will adhere to the convention that state variables are declared at the beginning of the class. This makes them easy to find and manage. You could move them to the end of the class with no effect.

The rest of our discussion pertains to local variables. All local variables in Java have a *block*; this is delimited by the closest pair of matching curly braces containing the variable's declaration. The first rule is that *no local variable is visible outside of its block*. The second rule is that *a local variable is not visible until it is created*. You will notice that these rules are stricter than those of Python. As in Python, variables in Java are not visible prior to their creation; this rule is exactly the same.

Here is an important difference. Variables created inside of Python functions are visible from their creation to the end of the function, even if they are declared inside of a block in that function. Here is a quick example in a file named `laxPyScope.py`.

```
def artificialExample(x):
    k = 0
    while k < len(x):
        lastSeen = x[k]
         k += 1
    return lastSeen
x = "parfait"
print "artificialExample(" + x + ") = ", artificialExample(x)
```

It is easy to see that the function `artificialExample` simply returns the last letter in a nonempty string. We run it here.

```
$ python laxPyScope.py
artificialExample(parfait) =  t
$
```

Observe that the variable `lastSeen` was created inside a block belonging to a `while` loop. In Java's scoping rules, this variable would no longer be visible (it would be destroyed) as soon as the loop's block ends.

There are some immediate implications of this rule. Any variable declared inside of a method in a class can only be seen inside of that method. That works out the same as in Python. Let us code up exactly the same thing in Java in a class `StrictJavaScope`. In this little demonstration, you will see Java's `while` loop at work.

```
public class StrictJavaScope
```

```
{
    public char artificialExample(String x)
    {
        int k = 0;
        while( k < x.length())
        {
             char lastSeen = x.charAt(k);
             k += 1;
        }
        return lastSeen;
    }
}
```

Now hit F5 and brace yourself for angry yellow.

```
1 error found:
File: /home/morrison/StrictJavaScope.java  [line: 11]
Error: /home/morrison//StrictJavaScope.java:11: cannot find symbol
symbol  : variable lastSeen
location: class StrictJavaScope
```

You will also see that the return statement is highlighted with angry yellow. Your symbol `lastSeen` died when the `while` loop ended. Even worse, it got declared each time the loop was entered and died on each completion of the loop.

How do we fix this? We should declare the `lastSeen` variable before the loop. Then its block is the entire function body, and it will still exist when we need it. Here is the class with repairs effected.

```
public class StrictJavaScope
{
    public char artificialExample(String x)
    {
        int k = 0;
        char lastSeen = ' ';
        while( k < x.length())
        {
             lastSeen = x.charAt(k);
             k += 1;
        }
        return lastSeen;
    }
}
```

Peace now reigns in the valley.

```
> s = new StrictJavaScope();
> s.artificialExample("parfait")
't'
>
```

`while` We are at it The use of the `while` loop is entirely natural to us and it looks a lot like Python. There are some differences and similarities. The differences are largely cosmetic and syntactical. The semantics are the same, save of this issue of scope we just discussed.

- **similarity** The `while` statement is a boss statement. No mark occurs in Java at the end of a boss statement.
- **difference** Notice that there is NO colon or semicolon at the end of the `while` statement. Go ahead, place a semicolon at the end of the `while` statement in the example class. It compiles. Run it. Now figure out what you did, Henry VIII.
- **difference** Notice that predicate for the `while` statement is enclosed in parentheses. This is required in Java; in Python it is optional.
- **similarity** The `while` statement owns a block of code. This block can be empty; just put an empty pair of curly braces after the loop header.

The scoping for methods and state variables is similar. State variables have *class scope* and they are visible from anywhere inside of the class. They may be modified by any of the methods of the class. Any method modifying a state variable is a mutator method for the class. Be careful when using mutator methods, as we have discussed some of their perils when we talked about aliasing. A good general rule is that if a class creates small objects, give it no mutator methods. For our `Point` class, we could just create new `Point`s, rather than resettiing coördinates. Then you do not have to think about alaiasing. In fact, it allows you to share objects among variables freely and it can save space. It also eliminates the need for copying objects.

Later, we will deal with larger objects, like graphics windows and displays. We do not want to be unnecessarily calling constuctors for these large objects and we will see that these objects in the standard libarary have a lot of mutator methods.

All methods are visible inside of the class. To get to methods outside of the class, you create an instance of the class using `new` and call the method via the instance. Even if your state variables are (foolishly) `public`, you must refer to them via an instance of the class. Let us discuss a brief example to make this clear.

Suppose you have a class `Foo` with a method called `doStuff()` and public a public state variable `x`. Then to get at `doStuff` or `x` we must first create a new `Foo` by calling a constructor. In this example we will use the default.

```
Foo f = new Foo();
```

Then you can call `doStuff` by making the call

```
f.doStuff();
```

Here you are calling `doStuff` via the instance `f` of the class `Foo`. To make `f`'s `x` be 5, we enter the code

```
f.x = 5;
```

Notice that the "naked" method name and the naked variable name are not visible outside of the class. In practice, since all of our state variables wil be marked `private`, no evidence of state variables is generally visible outside of any class.

2.6 The Object-Oriented Weltanschauung

Much emphasis has been placed here on classes and objects. In this section we will have a discussion of programming using objects. We will begin by discussing the procedual programming methods we developed in Chapters 0-7 of the Python book.

2.6.1 Procedural Programming

When we first started to program in Python, we wrote very simple programs that consisted only of a main routine. These programs carried out small tasks and were short so there was little risk of confusion or of getting lost in the code.

As we got more sophisiticated, we began using Python functions as a means to break down, or modularize, our program into managable pieces. We would then code each part and integerate the functions into a program that became a coherent whole. Good design in Python is "top down." You should nail down what you are trying to accomplish with your program. Then you should break the program down into components. Each component could then be broken into smaller components. When the components are of a manageable size, you then code them up as functions.

To make this concrete, let us examine the case of writing a program that accepts a string and which looks through an English wordlist, and which shows all anagrams of the string you gave as input which appear in the wordlist.

To do this, you could write one monolithic procedure. However, the procedure would get pretty long and it would be trying to accomplish many things at once. Instead we might look at this and see we want to do the following

Obtain the word from the user.
Open a wordlist file.
Generate all the anagrams on the user–supplied word.
Sort the list in alphabecial order
Go through the sorted list, checking each anagram to see if it is in the wordlist
Return the list of words we obtained to the user.

Not all the tasks here are of the same difficulty. The first one, obtain the word from the user, is quite easy to do. We, however have to make a design decison and decide how to get the word from the user. This is a matter of deciding the program's *user interface*.

The job of sorting the (huge) list looks daunting, until we realize the Python list objects have a method `sort()`, which enables you to easily put the list of anagrams in order. Java, like Python offers a huge libary of classes and types. The purpose of this library is to help keep you from wasting time reinventing wheels. Use wheels whenever possible, do not reinvent them.

Python is an object-oriented language like Java with a library of classes. Many of the Python classes are extremely useful; the same is true in Java. Always look for a solution to your problem in the standard libarary before trying to solve it yourself! If you create classes intelligently, you will see that you will be able to reuse a lot of code you create.

Returning to our problem, you would revisit each subproblem you have found. If the subproblem is simple enough to write a function for it, code it. Otherwise, break it down further.

This is an example of top–down design for a *procedural* program. We keep simplifying procedures until they become tractible enough to code in a single function. We program with *verbs*.

The creation of functions gives us a layer of abstraction. Once we test our functions, we use them and we do not concern ourselves with their internal details (unless a function goes buggy on us), we use them for their *behavior*. Once a function is created and its behavior is known, we no longer concern ourselves with its local variables and the details of its implementation.

This is an example of *encapsulation*; we are thinking of a function in terms of its behavior and not in terms of its inner workings.

2.6.2 Object–Oriented Programming

In object–oriented programming, we program with *nouns*. A class is a sophisticated creature. It creates *objects*, which are computational creatures that have state, identity and behavior. We shall see here that encapsulation plays a large role in object-oriented programming. Good encapsulation dictates that we hide

things from client programmers they do not need to see. This is one reason we make our state variables private. We may even choose to make certain methods private, if the do think they are of real use other than that of a service role the other methods of the class.

When you used the `String` class, you did not need to know how the characters of a `String` are stored. You do not need to know how the `substring()` method works: you merely know the behavior it embodies and you use it. What you can see in the API guide is the *interface* of a class; this is a class's public portion. You are a client programmer for the entire standard libary, so you will only use a class's interface.

What you do not see is the class's *implementation*. You do not know how the `String` class works internally. You could make a good guess. It looks as if a list of characters that make up the string is stored somewhere. That proabably reflects the state of a `String` object.

A string, however, is a fairly simple object. The contents of a window in a graphical user interface (GUI) in Java is stored in an object that is an instance of the class `JFrame`. How do you store such a thing? Is it different on different platforms? All of a sudden we feel the icy breath of the possibly unknown.... However, there is nothing to fear! In Java the `JFrame` class has behaviors that allow you to work with a frame in a GUI and you do not have to know *how* the internal details of the `JFrame` work. This is the beauty of encapsulation. Those details are thannkfully hidden from us!

For an everyday example let us think about driving a car. You stick in the key, turn it, and the ignition fires the engine. You then put the car in gear and drive. Your car has an interface. There is the shifter, steering wheel, gas pedal, the music and climate controls, the brakes and the parking brake. There are other interface items like door locks, door openers and the dome light switch.

These constitute the "public" face of your car. You work with this familiar interface when driving. It is similar across cars. Even if your car runs on diesel, the interface you use to drive is not very different from that of a gasoline–fueled car.

You know your car has "private parts" to which you do not have direct access when driving. Your gas pedal acts as a mutator method does; it causes more gas to flow to the fuel injection system and causes the RPM of the engine to increase. The RPM of the engine is a state variable for your car. Your tachometer (if your car has one) is a getter method for the RPM of your engine. You affect the private parts (the implementation) of your car only indirectly. Your actions occur through the interface.

Let's not encapuslate things for a moment. Imagine if you had to think about *everything* your car does to run. When you stick your key in the ignition, if you drive a newer car, an electronic system analyzes your key fob to see if your key is genuine. That then triggers a switch that allows the ignition to start

the car. Then power flows to the starter motor..... As you are tooling down the highway, it is a safe bet you are not thinking about the intracacies of your car's fuel injection system and the reactions occurring in its catalytic converter. You get the idea. Encapsulation in classes simplifies things to the essence and allows us to think about the problem (driving here) at hand. You use the car's interface to control it on the road. This frees your mind to think about your actual driving.

So, a Java program is one or more classes working together. We create instance of these classes and call methods via these instances to get things done. In the balance of this book, you will gain skill using the standard libary classes. You will learn how to create new classes and to create extensions of existing ones. This will give you a rich palette from which to create programs.

Chapter 3

Translating Python to Java

3.0 Introduction

We are going to frame the concepts we learned in Python in Java. During this chapter, we will do a comparison of the design and mechanics of the two languages.

3.1 Java Data Structures

Recall that a data structure is a container by which we store a collection of related objects under a single name. In Python, we met the data structures `list`, `tuple` and `dict`. Python lists are mutable heterogeneous sequences. Python tuples are like lists, but they are immutable. Python dictionaries allow us to store key-value pairs. Python lists grow according to our needs and they shrink when we delete items from them.

Java has two data types comparable to Python lists. We begin by learning about the *array*; it is a homogeneous mutable sequence type of fixed size. When you create an array it must be of a fixed size. If you run out of room and wish to add more entries, you must create a new, bigger array, copy your array into its new home and then abandon the old array. You should only use arrays if you want a list of a fixed size. All entries in the array must be of the same type. Arrays can be of primitive or object type. The array itself is an object. We see that an array lives up to its description as a homogeneous mutable sequence type.

Open an interactive session in DrJava and hit F5 to reset the interactions pane. We will use our first `import` statement here. The `import` statement works much as it does in Python. Importing the class `java.util.Arrays` will

give us a convenient way to print the contents of an array; the built-in string representation of an array is unsatisfactory.

Let us begin by declaring a variable of integer array type.

```
> import java.util.Arrays;
> int[] x;
```

Now let's try to assign something to an entry.

```
> x[0] = 1
NullPointerException:
  at java.lang.reflect.Array.get(Native Method)
```

We are greeted by a surly error message. Here is one sure reason why.

```
> Arrays.toString(x)
"null"
```

Right now, the array is pointing at Java's "graveyard state" `null`. If you attempt to use a method on a object pointing at `null`, you will get a run time error called a `NullPointerException`. We need to give the array some actual memory to point to; this is where we indicate the array's size.

We call the special array constructor to attach an actual array to the array pointer `x`. After we attach the array, notice how we obtain the array's length.

```
> x = new int[10];
> Arrays.toString(x)
"[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
> x.length
10
```

Observe that Java politely placed a zero in each entry in this integer array. This will happen for any primitive numerical type. If you make an array of booleans, it will be populated with the value `false`. In a character array, watch what happens.

```
> char[] y = new char[10];
> Arrays.toString(y)
"[, , , , , , , , , ]"
> (int) y[0]
0
```

The array is filled with the *null character* which has ASCII value 0. It is not a printable character. An array of object type is filled with `null`s. You will need to loop through the array and use `new` to attach an object to each entry.

Arrays have indices, just as lists do in Python. Remember, you should think of the indices as living *between* the array entries. Arrays know their length, too; just use `.length`. Notice that this is NOT a method and it is an error to use parentheses.

3.1.1 Goodies inside of `java.util.Arrays`

The convenience class `java.util.Arrays` is a "service class" that has useful methods for working with arrays. We will demonstrate some of its methods here. If you are working on arrays, look to it first as as a means of doing routine chores with arrays. Its methods are fast, efficient and tested.

Go to the API page; you will see some useful items there. Here is a summary of the most important ones for us. We will use `Type` to stand for a primitive or object type. Hence `Type[]` means an array of type `Type`. You call all of these methods by using `Arrays.`*method(arg(s))*.

Header	Action
`Type[] copyOf(Type[] original, int newLength)`	This copies your array and returns the copy. If the newLength is shorter than your array, your array is truncated. Otherwise, it is lengthened and the extra entries are padded with the default value of `Type`.
`Type[] copyOfRange(Type[] original, int from, int to)`	This returns a slice of your original array, between indices `from` and `to`. Using illegal entries generates a run time error.
boolean equals(Type[] array1, Type[] array2)	This returns `true` if the two arrays have the same length and contain the same values. It works just like Python's `==` on lists.
void fill(Type[] array, Type value)	This will replace all of the entries in the array `array` with the value `value`.
void fill(Type[] array, int from, int to, Type value)	This will replace the entries between indices `from` and `to` with the value `value`.
`String toString(Type[] array`	This pretty-prints your array as a string. You have seen this used.

3.1.2 Fixed Size? I'm finding this *very* confining!

We now introduce a new class and a new piece of Java syntax. An `ArrayList` is a variable-size array. There are two ways to work with `ArrayList`s and we will show them both.

Let us create an `ArrayList` and put some items in it. To work with an `ArrayList` you will need to import the class. The `import` statement in the inter-

active session below shows how to make the class visible. `java.util.ArrayList` is the fully–qualified name of this class. The import statement puts us on a "first-name" basis with the class.

How do I know what to import? Look the class `ArrayList` up in the API guide. Near the top of the page, you will see this.

> java.util
> ArrayList<E>

This tells you that the `ArrayList` class lives in the package `java.util`. Therefore you should place this at the top of your code. Do not put the <E> in the `import` statement.

```
import java.util.ArrayList;
```

We will use the `ArrayList`'s `add` method to place new items on the list we create.

```
> import java.util.ArrayList;
> ArrayList pool = new ArrayList();
> pool
[]
> pool.add("noodle")
true
> pool.add("chlorine")
true
> pool.add("algicide")
true
> pool
[noodle, chlorine, algicide]
> pool.get(0)
"noodle"
```

All looks pretty good here. But then there is an irritating snag.

```
> pool.get(0).charAt(0)
Error: No 'charAt' method in '}java.lang.Object' with arguments: (int)
>
```

3.1.3 A Brief but Necessary Diversion: What is this `Object` object?

To explain what just happened to us properly, we will take a look into the near future that lurks in Chapter 4. Every object of object type in Java, logically

enough, is an `Object`. Go into the API guide and look up the class `Object`.

Every Java class has a place in the Java class hierarchy, including the ones you create. What is different from human family trees is that a Java class has *one* parent class. A Java class can have any number of children. This hierarchy is independent of the hierarchical structure imposed on the Java Standard Library by packages.

The Java class hierarchy is an Australian (upside–down) tree, just like your file system. In LINUX, your file system has a root directory called `/`. In the Java class hierarchy, the class `Object` is the root class.

Heretofore, we have created seemingly stand–alone classes. Our classes, in fact have not really been "stand–alone." Automatically, Java enrolls them into the class hierarchy and makes them children of the `Object` class. This is why every object has a `toString()` method, even if you never created it.

The only stand–alone types are the primitive types. They are entirely outside of the Java class hierarchy. At the end of this chapter, you will see that these, too, have `Object` analogs.

What is entailed in this parent–child relationship? The child *inherits* the public portion of the parent class. In a human inheritance, the heirs can decide what to do with the property they receive. They can use the property for its original purpose or redirect it to a new purpose. In Java, the same rule applies. When we made a `toString()` method for our `Point` class, we decided to redirect our inheritance. Every Java object is born with a `toString()` method. Unfortunately the base `toString()` method gives us a default string representation of our object that looks like this.

```
ClassName@ABunchOfHexDigits
```

We decided this is not terribly useful so we *overrode* the *base* `toString()` method and replaced it with our own. To override a method in a parent class, just re–implement the method, with exactly the same signature, in the child class. We also overrode the `clone()` method in the parent class. If you intend to copy objects, do not trust the `clone()` you inherit from `Object`.

This table describes the methods in the `Object` class and the relevance of each of them to us now.

Object Method	Description
`clone()`	This method creates and returns a copy of an object. You should override this if you intend to use independent copies of instance of your class.
`finalize()`	This method is automatically called when the garbage collector arrives to reclaim the object's memory. We will rarely if ever use it.
`getClass()`	This method tells you the class that an object was created from.
`notify()`	This method is used in threaded programs. We will deal with this much later

The three `wait` methods and the `notifyAll` methods all apply in threaded programming. Threads allow our programs to spawn subprocesses that run independently of our main program. Since these are a part of `Object`, this tells you that threading is built right into the core of the Java language. We will develop threading much later.

3.1.4 And Now Back to the Matter at Hand

Everything is returned from an `ArrayList` is an `Object`. Strings have a `charAt()` method, but an `Object` does not. As a result you must perform a cast to use things you get from an `ArrayList`. Here is the (ugly) syntax. Ugh. It's as ugly as Scheme or Lisp.

```
> ((String)pool).get(0).charAt(0)
'n'
>
```

This is the way things were until Java 5. Now we have *generics* that allow us to specify the type of object to go in an `ArrayList`. Generics make a lot of the ugliness go away. The small price you pay is you must specify a type of object you are placing in the `ArrayList`. The type you specify is placed in the *type parameter* that goes inside the angle brackets `< .... >`. You may use any object type as a type parameter; you may not do this for primitive types.

```
ArrayList<String> farm = new ArrayList<String>();
> farm.add("cow")
true
> farm.get(0).charAt(0)
'c'
>
```

Warning: Deception Reigns King Here! All here has a pleasing cosmetic appearance. However, it's time to take a peek behind the scenes and see the

real way that generics work.

What happens behind the scenes is that the *compiler* enforces the type restriction. It also automatically inserts the needed casts for the get() method. Java then erases all evidence of generics prior to run time.

The generic mechanism should not work at run time. However, the wizards who created DrJava made generics work at run-time.

At run time you actually could add any type of of object to an ArrayList of strings in the interactions pane. So here is what happens behind the scenes.

1. You make an `ArrayList` of some type, say `String` by using the `ArrayList<String>` syntax.
2. You put things on the list with `add` and friends and gain access to them with the `get()` method.
3. The compiler will add the necessary casts to `String` type when you refer to the entries of the `ArrayList` using `get()`, removing this annoyance from your code.
4. The compiler then performs *type erasure*; it eliminates all mention of the type parameter from the code, so to the run time environment, `ArrayList`s look like old–style `ArrayList`s at run time.

This is a smart decision for two reason. One reason is that it prevents legacy code from breaking. That code will get compiler growlings and warnings about "raw types" but it will continue to work.

Secondly, if you declare `ArrayList`s of various type, each type of `ArrayList` does not generate new byte code. If you are familiar with C++, you may have heard that C++'s version of generics, *templates*, causes "code bloat;" each new type declared using a C++ template creates new object code in your executable file. Because of type erasure, Java does not do this.

Let us now make a sample class that takes full advantage of generics. First, let us make a version without generics and see something go wrong.

```
import java.util.ArrayList;
public class StringList
{
    ArrayList theList;
    public StringList()
    {
        theList = new ArrayList();
    }
    public boolean add(String newItem)
    {
        return theList.add(newItem);
```

```
    }
    public String get(int k)
    {
        return theList.get(k);
    }
}
```

Compile this program and you will get a nastygram like this.

```
1 error and 1 warning found:
-------------
*** Error ***
-------------
File: /home/morrison/book/texed/Java/StringList.java  [line: 15]
Error: /home/morrison/book/texed/Java/StringList.java:15: incompatible types
found   : java.lang.Object
required: java.lang.String
-------------
** Warning **
-------------
File: /home/morrison/book/texed/Java/StringList.java  [line: 11]
Warning: /home/morrison/book/texed/Java/StringList.java:11:
 warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.ArrayList
```

The error is that we are advertising that `get` returns a `String`; the `ArrayList`'s `get()` only returns an `Object`. Now let us add the type parameter `<String>` to the code. Your code compiles. Let us now inspect our class interactively. We can now cast aside our worries about casts.

```
> StringList s = new StringList();
> s.add("Babe Ruth")
true
> s.add("Mickey Mantle")
true
> s.add("Lou Gehrig")
true
> s.get(0)
"Babe Ruth"
> s.get(0).charAt(0)
'B'
>
```

You can see that `s.get(0)` in fact returns a `String`, not just an `Object`, since it accepts the `charAt()` message.

Programming Exercises These exercises will help familiarize you with the `ArrayList` API page. This class offers an abundance of useful services. Create a class called `Al` and place the indicated methods inside of it. Inspect them in the interactions pane to test them.

1. Make a new `ArrayList` of strings named `roster`.
2. Add several lower-case words to the `ArrayList`; view its contents as you add them.
3. Enter this command `import java.util.Collections`.
4. Type this command `Collections.sort(roster);` Tell what happens.
5. Type this command `Collections.shuffle(roster);` Tell what happens.
6. Add some upper-case words. How do they get sorted in the list using `Collections.sort()`?

3.2 Conditional Execution in Java

Java, like Python, supports conditional execution. Python has `if`, `elif` and `else` statements. These are all boss statements. All of this is the works the same way in Java, but the appearance is a little different. Here is a comparison method called `ticketTaker` in Python and Java. First we show the Python version.

```
def ticketTaker(age):
    if age < 13:
        print "You may only see G movies."
    elif age < 17:
        print "You may only see PG or G movies."
    elif age < 18:
        print "You may only see R, PG, or G-rated movies."
    else:
        print "You may see any movie."
```

The Java version is quite similar. The keywords change a bit. Notice that the predicates are enclosed in parentheses. This is required. Observe in this example that you can put a one–line statement after an `if`, `else if` or `else` without using curly braces. If you want one more than one line or an empty block attached to any of these, you must use curly braces.

```
public void ticketTaker(int age):
    if (age < 13)
    {
        System.out.println("You may only see G movies.");
```

```
    }
    else if (age < 17)
    {
        System.out.println("You may only see PG or G- movies.");
    }
    else if (age < 18)
    {
        System.out.println("You may only see R, PG or G movies.");
    }
    else
    {
        System.out.println("You may see any movie.);
    }
```

Both languages support a ternary statement. We shall illustrate it in an absolute value function for both languages. First here is the Python version.

```
def abs(x):
    return x if x >= 0 else -x
```

Now we show Java's ternary operator at work.

```
public int abs(int x)
{
    return x >= 0 ? x : -x;
}
```

Use parenthesis to keep the order of operations from producing undesired results where necessary.

Java supports an additional mechanism, the `switch` statement for forking. We show an example of this statement and then explain its action.

```
public class Stand
{
    public String fruit(char c)
    {
        String out = "";
        switch(c)
        {
            case 'a': case 'A':
                out = "apple";
                break;
            case 'b': case 'B':
                out = "blueberry";
                break;
```

```
            case 'c': case 'C':
                out = "cherry";
                break;
            default:
                out = "No fruit with this letter";
        }
        return out;
    }
}
```

Let us now instantiate the `Stand` class and test its `fruit` method.

```
Welcome to DrJava.  Working directory is /home/morrison/book/texed/Java
> s = new Stand()
Stand@6504bc
> s.fruit('A')
"apple"
> s.fruit('b')
"blueberry"
> s.fruit('z')
"No fruit with this letter"
>
```

The `switch`–`case` statement only allows you to switch on a variable of *integral type*, i.e. an integer or character type. Java 7 also allows you to switch on a `String`.

This construct cannot be used on variables of floating-point type. Do not use it on a `boolean` variable; for these, we use the `if` machinery. At the end of each row of one or more `case`s, you place zero or more lines of code followed by a `break` statement. Remove various `break` statements and note the behavior of the function. You will see that they play an important role. If you do not like `switch`–`case`, you can live without it with little or no deleterious effect.

3.3 Extended–Precision Integer Arithmetic in Java

We shall introduce a new class, `BigInteger`, which does extended–precision integer arithmetic. Go into the Java API guide and bring up the page for `BigInteger`. Just under the main heading

> java.math
> Class BigInteger

you well see this class's family tree. Its parent is `java.lang.Number` and its grandparent is `java.lang.Object`. The fully–qualified name of the class is

`java.math.BigInteger`. To use the class, you will need to put the import statement

```
import java.math.BigInteger;
```

at the top of your program. You can always look at the bottom of the family tree to see what import statement is needed.

Remember that you never need to import any class that is in `java.lang`, such as `java.lang.String`. These are automatically imported for you. Python seamlessly integrates super-long integers into the language. This is not so in Java. Java class developers cannot override the basic operators like +, -, * and /.

Begin by looking a the Constructor summary. The most useful constructor to us seems to be

```
BigInteger(String val)
```

Now we shall experiment with this in an interactive session.

```
> import java.math.BigInteger;
> p = new BigInteger("1");
> p
1
>
```

We now have the number 1 stored as a `BigInteger`. Continuing our session, we attempt to compute 1 + 1.

```
> p + p
Error: Bad type in addition
>
```

In a program this would be a compiler error. Now go into the method summary and look for `add`.

```
> p.add(p)
2
> p
1
>
```

The `add` method computes `1 + 1` in `BigInteger` world and comes up with `2`. Notice that the value of `p` did not change. This is no surprise, because `BigInteger`s are immutable.

To find out if a class makes immutable objects, look in the preface on its page in the API guide. First you see the header on this page, then the family tree. Then there is a horizontal rule, and you see the text

```
public class BigInteger
extends Number
implements Comparable<BigInteger>
```

The phrase "`extends Number`" just means that the `Number` class is the parent of `BigInteger`. We will learn what "implements" means when we deal with interfaces; we do not need it now.

Next you see the preamble, which briefly describes the class. Here it says "Immutable arbitrary–precision integers." So, as with strings, you must orphan what a variable points at to get the variable to point at anything new. Now let us see exponentiation, multiplication, subtraction and division at work.

```
> import java.math.BigInteger;
> a = new BigInteger("1341121");
> b = a.pow(5);
> a
1341121
> b
433850212910726822977864452960100
> c = b.multiply(new BigInteger("100"))
433850212910726822977864452960100
> d = a.subtract(new BigInteger("1121"));
> d
1340000
> d.divide(new BigInteger("1000"))
1340
>
```

It would be convenient to have a way to convert a regular integer to a big integer. There is a method

```
static BigInteger valueOf(long val)
```

that looks promising, but what is this `static` thing? For now it means you can call this method without making an instance of the class. We will get more into `static` methods later. To call this (static) method, the usage is

```
BigInteger.valueof(whateverIntegerYouWantConverted)
```

This is true of any static method. Use static items now without fear. You may always call them using the class name as shown here. We already slipped this in

whilst you were not looking; every method in the classes `Math` and `Arrays` are static. The `BigInteger.valueOf()` method is called a *static factory method*; it is a "factory" that converts regular integers into their bigger brethren.

We now show an example or two. Be reminded of the need to use the `equals` method when working with variables pointing at objects, so you do not get a surprise.

```
> import java.math.BigInteger;
> p  = BigInteger.valueOf(3)
3
> q = new BigInteger("3")
3
> p == q
false
> p.equals(q)
true
>
```

3.4 Recursion in Java

Java supports recursion, and subject to the new syntax you learned in Java, it works nearly the same way. All of the pitfalls and benefits you learned about in Python apply in Java. Let us write a factorial function using the `BigInteger` class Recall the structure of the factorial function in Python.

```
def factorial(n):
    return 1 if n <= 0 else n*factorial(n - 1)
```

Everything was so simple and snappy.

Now we have to convert this to Java using the operations provided by `BigInteger`. We do have some tools at hand. `BigInteger.valueOf()` converts regular integers into their leviathan brethren. We also have to deal with the `.multiply` syntax to multiply. Finally, we must remember, *we are returning a* `BigInteger`. Bearing all those consideration in mind, you should get something like this. If the ternary operators is not quite to your taste, use an `if` statement instead. We have broken the big line here solely for typographical convenience.

```
import java.math.BigInteger;

public class Recursion
{
    public BigInteger factorial(int n)
```

```
    {
        return n > 0 ?
            factorial(n - 1).multiply(BigInteger.valueOf(n)):
            BigInteger.valueOf(1);
    }
}
```

Now let us test our function.

```
> r = new Recursion();
> r.factorial(6)
720
> r.factorial(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000
> r.factorial(1000)
40238726007709 ... (scads of digits)  ...00000
>
```

Recursion can be used as a repetition mechanism. We add a second method `repeat` to our class to character or string is passed it any specified integer number of times to imitate Python's `string * int` repeat mechanism. This will serve as a nice example of method overloading. First let us work with the `String` case. Let us call the String `s` and the integer `n`. If `n <= 0`, we should return an empty string. Otherwise, let us glue a copy of `s` to the string `repeat(s, n - 1)`

```
public String repeat(String s, int n)
    {
        String out = "";
        if(n > 0)
        {
            out += s + repeat(s, n - 1);
        }
        return out;
    }
```

Now we get the character case with very little work.

```
    public String repeat(char ch, int n)
    {
        return repeat("" + ch, n)
    }
```

Now our `repeat` method will repeat a character or a string. We do not need to worry about the character or string we need to repeat. Method overloading makes sure the right method is called.

3.5 Looping in Java

We have already seen the `while` loop in Java. It works in a manner entirely similar to Python's `while` loop. For your convenience, here is a quick comparison

```
while predicate:
    bodyOfLoop

while(predicate)
{
    bodyOfloop
}
```

It looks pretty much the same. All of the same warnings (beware of hanging and spewing) apply for both languages.

Java also offers a second version of the `while` loop, the `do-while` loop. Such a loop looks like this.

```
do
{
    bodyOfloop
}
while(predicate);
```

The body of the loop executes unconditionally the first time, then the predicate is checked. What is important to realize is that the predicate is checked *after* each execution of the body of the loop. When the predicate evaluates to `false`, the loop's execution ends. Almost always, you should prefer the `while` loop over the `do-while` loop. When using this loop, take note of the semicolon; you will get angry yellow if you omit it.

Java has two versions of the `for` loop. One behaves somewhat like a variant of the `while` loop and comes to us from C/C++. The other is a definite loop for iterating through a collection.

First let us look at the C/C++ `for` loop; its syntax is

```
for(initializer; test; between)
{
    loopBody
}
```

This loop works as follows. The `initializer` runs once at when the loop is first encountered. The initializer may contain variable declarations or initializations. Any variable declared here has scope only in the loop.

The `test` is a predicate. Before each repetition of the loop, the `test` is run. If the `test` fails (evaluates to `false`), the loop is done and control passes beyond the end of the loop. If the `test` passes, the code represented by `loopBody` is executed. The `between` code now executes. The `test` predicate is evaluated, if it is true, the `loopBody` executes. This process continues until the `test` fails, at which time the loop ends and control passes to the line of code immediately beyond the loop. This loop is basically a modified `while` loop.

Java also has a `for` loop for collections that works similarly to Python's `for` loop. Observe that the loop variable `k` is an iterator, just as it is in Python's `for` loop. It has a look-but-don't-touch relationship with the entries of the array. It grants access but does not allow mutation. This works for both class and primitive types.

```
import java.util.ArrayList;
> ArrayList<String> cats = new ArrayList<String> ();
> cats.add("siamese")
true
> cats.add("javanese")
true
> cats.add("manx")
true
> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
> for(String k : cats){k = "";}//Look, but don't touch!
> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
```

3.6 Static and `final`

You have noticed that the `static` keyword appears sometimes in the API guide. In Java, `static` means "shared." Static portions of your class are shared by all instances of the class. They must, therefore, be independent of any instance of the class, or *instance-invariant*.

When you first instantiate a class in a program, the *Java class loader* first sets up housekeeping. It loads the byte code for the class into RAM.

Before the constructor is called, any static items go in a special part of memory that is visible to all instances of the class. Think of this portion of memory as being a bulletin board visible to all instances of the class. You may make static items public or private, as you see fit.

When state variable or method is static, it can be called by the class's name. For instance, `BigInteger.valueOf()` is a static method that converts any `long` into a `BigInteger`. Recall we called this method a static factory method; it is static and behaves as a "factory" that accepts `long`s and converts the to `BigIntegers`.

Two other familiar examples are the `Math` and `Arrays` classes. In the `Math` class, recall you find a square-root by using `Math.sqrt()`, in `Arrays`, `Arrays.toString(somearray)` creates a string representation of the array passed it. All of `Math`'s and `Arrays` methods are static. Neither has a public constructor. Both are called *convenience* or `service` classes that exist as containers for related methods.

You can also have state variables that are declared static. In the `Math` library, there are `Math.PI` and `Math.E`. These instance variables are static. They are also `final`; they are immutable variables. Variables anywhere in Java can be marked `final`; this means you cannot change the datum the variables point to. However, you can call mutator methods on that datum and change the state of the object a `final` variable points to. Be aware that, in this context, finality is a property of variables and not objects. What you cannot do is to make such a variable point at a different object. Since primitive types have no mutator methods, they cannot be changed at all if they are marked `final`. Primitive and object variables are completely constant if they are marked `final` ; the values they point at and the state of the objects they point at cannot be changed.

You can also have a `static` block at in your class. Code inside this block is run when the class is first loaded. Use it to do things such as initializing static data members. In fact, it is a desirable postcondition of your `static` block running that all static state variables be initialized. Now let us put `final` and `static` to work.

The `Minter` class shown here gives each new instance an ID number, starting with 1. The static variable `nextID` acts as a "well" from which ID numbers are drawn. The `IDNumber` instance variable is marked `final`, so the `ID` number cannot be changed throughout any given `Minter`'s lifetime.

```
public class Minter
{
    private static int nextID;
    final private int ID;
    static
    {
        nextID = 1;
```

```
    }
    public Minter()
    {
        ID = nextID;
        nextID++;
    }
    public String toString()
    {
        return "Minter, ID = " + ID;
    }
}
```

3.6.1 Etiquette Between Static and Non-Static Members

Since the Java class loader creates the static data for a class before any instance of the class is created, there is a separation between static and non–static portions of a class.

Non–static methods and state variables may access static portions of a class. This works because the static portion of the class is created before any instance of the class is created, so everything needed is in place. Outside of your class, other classes may see and use the static portions of your class that are marked `public`. These client programmers do not need to instantiate your class. They can gain access to any static class member, be it a method or a state variable by using the

```
ClassName.staticMember
```

construct.

The reverse is `not` permitted. If a method of your class is static, it cannot call non-static methods of your class. It is not allowed to use non–static state variables.

The key to understanding why is to know that *static data is shared by all instances of the class*. Hence, to be well-defined, *static data must be instance–invariant*. Since your class methods can, and more often than not, do depend on the state variables in your class, they in general are not instance–invariant. Static methods and variables belong to the class as a whole, not any one instance. This restriction will be enforced by the compiler. Even if a method does not depend upon a class's state, unless you declare it `static`, it is not static and static methods may not call it.

To use any class method in the non-static portion of your class, you must first instantiate the class and call the methods via that instance. We will see an example this at work in the following subsection

3.6.2 How do I Make my Class Executable?

To make your class executable, add the following special method.

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

To run your class, compile it. Select the interactions pane in the bottom window and hit F2. You can also type

```
> java YourClassName
```

at the prompt. Do not put any extension on `YourClassName`.

For a simple example, place this method in the `Minter` class we just studied.

```
public static void main(String[] args)
 {
     Minter m = new Minter();
     System.out.println(m);
 }
```

Run the class and you will see it is now executable. Hitting the F2 button in your class's code window automatically causes the `java` command to be placed at the prompt.

```
> java Minter
Minter, IDNumber = 1
```

Observe that we made a tacit call to a method of the class `Minter`. To use the class, we had to create an instance `m` of `Minter` first. When we called `System.out.println`, we made a tacit call to `m.toString()`. You cannot make naked (no-instance) methods calls to non-static methods in `main`. You can, however, see the private parts of instances of the class.

Really, it is best to think of `main` as being outside the class and just use instance of your or other classes and use their (public) interface.

Finally, notice that `main` has an argument list with one argument, `String[] args`. This argument is an array of `String`s. This is how command–line arguments are implemented in Java. We now show a class that demonstrates this feature.

```
public class CommandLineDemo
```

```
{
    public static void main(String[] args)
    {
        int num = args.length;
        System.out.println("You entered " + num + " arguments.");
        int count = 0;

        for (String k: args)
        {
            System.out.println("args[" + count + "] = " + k);
            count ++;
        }
    }
}
```

Now we shall run our program with some command–line arguments. You need to type in the `java` command yourself rather than just hitting F2.

```
> java CommandLineDemo one two three
You entered 3 arguments.
args[0] = one
args[1] = two
args[2] = three
>
```

Even if you do not intend for your class to be executable, the `main` method is an excellent place for putting test code for your class. Making your class executable can save typing into the interactions pane. It is also necessary if you ever want to distribute your application in an *Java archive*, which is an executable file.

3.7 The Wrapper Classes

Every primitive type in Java has a corresponding *wrapper class*. Such a class "wraps" the primitive object. These classes also supply various useful methods associated with each primitive type. Here is a table showing the wrapper classes.

Wrapper Classes	
Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

All of these classes have certain common features. You should explore the API guide for each wrapper. They have many helpful features that will save you from reinventing a host of wheels.

- **Immutability** Instances of these classes are immutable. You may not change the datum. You may only orphan it. This should remind you of Python.
- **Finality** These classes are final. You may not create child classes from them.
- **A `toString()` Method**, which returns a string representation of the datum.
- **A static `toString(primitiveType p)` Method** This method will convert the object passed it into a string. For example, `Integer.toString(45)` returns the string `"45"`.
- **A static parsePrimitive(String s) Method** This method converts a string into a value of the primitive type. For example,

  ```
  Integer.parseInt("455")}
  ```

  converts the string `"455"` into the integer `455`. For numerical types, a `NumberFormatException` is thrown if an malformed numerical string is passed them. The `Character` wrapper does not have this feature. You should take note of how this method works in a `Boolean`.
- **Membership in `java.lang`** All of these classes belong to the package `java.lang`; you do not need to include anything to use these classes.

3.7.1 Autoboxing and Autounboxing

These Java 5 features make using the wrapper classes simple. Autoboxing automatically promotes a primitive to a wrapper class type where appropriate. Here is an example. The command

```
Integer i = 5;
```

really amounts to

```
Integer i = new Integer(5);
```

This call to `new` "boxes" the primitive 5 into an object of type `Integer`. The command `Integer i = 5;` automatically boxes the primitive 5 inside an object of type `Integer`. This results in a pleasing syntactical economy.

Autounboxing allows the following sensible–looking code.

```
Integer i = 5;
int x = i;
```

Here, the datum is automatically "unboxed" from the wrapper `Integer` type and it is handed to the primitive type variable `x`.

The following convenience is also provided.

```
> Integer i = 5;
> int x = 5;
> i == x
true
>
```

Autoboxing and autounboxing eliminate a lot of verbosity from Java; we no longer need to make most `valueOf` and `equals` calls.

3.8 A Caveat

Do not box primitive types gratuitously. If you can keep variables primitive without a sacrifice of clarity or functionality, do so. Here is an example of a big mistake caused by seemingly innocuous choice.

```
for(Integer i = 0; i < 1000000; i++)
{
//code
}
```

Here the integer i will be unboxed to be incremented and reboxed one million times. This will be a significant performance hit. This is much better.

```
for(int i = 0; i < 1000000; i++)
{
//code
}
```

Note that if you wish to make an `ArrayList` of primitives, you should use the wrapper type as the type parameter. For example

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

creates an array list with integer entries.

Use the wrapper classes to do these exercises.

Programming Exercises

1. Write an expression to see if a character is an upper-case letter.
2. Write a method that converts an integer into a comma format string as follows.
   - `456` $\mapsto$ `456`
   - `32768` $\mapsto$ `32,768`
   - `1048576` $\mapsto$ 1,048,576
3. Write a simple program that queries the user with a `JOptionPane` to enter an integer and which converts the integer string entered by the user to an `int`.

3.9 Case Study: An Extended-Precision Fraction Class

We have achieved several goals in this chapter, the most important of which are understanding what makes up a Java class and understanding the core Java language so as to be Turing-complete.

To tie everything together, we will do a case study of creating a class called `BigFraction`, which will work like the `BigInteger` class and provide many of the same operations, except it will do exact fractional arithmetic.

We select this case study because it brings to the fore a variety of important design questions. When we are done, we will have a nice facility for computing with fractions. You will get to see the development of a moderately sophisiticated class from scratch.

As we proceed we will place appropriate test code in an `main` method.

Let us begin by thinking about fractions. We know a fraction has two important items reflecting its state: its numerator and its denominator. Fractions have some slippery properties. For example, we know that

$$\frac{1}{4} = \frac{256}{1024}.$$

The representation of a fraction in terms of numerator and denominator is not unique.

An interesting collection of numbers is the *harmonic numbers*; they are defined by

$$H_n = \sum_{k=1}^{n} \frac{1}{k}, \qquad n \in \mathbb{N}.$$

Let us show the first few harmonic numbers. It is easy to see that $H_1 = 1$. We have

$$H_2 = 1 + \frac{1}{2} = \frac{3}{2}.$$

Next,

$$H_3 = H_2 + \frac{1}{3} = \frac{3}{2} + \frac{1}{3} = \frac{11}{6}.$$

Now let's skip down to H_{10}.

$$H_{10} = \frac{7381}{2520}.$$

One thing is clear: as we keep adding fractions, their numerators and denominators have a propensity to keep getting larger. We know that the primitive `int` and `long` types are not going to cut the mustard here because they will overflow and produce false results.

We will therefore use `BigInteger` for the numerator and denominator of our `BigFractionss`. We should be able to compute H_{100} or even H_{1000}.

3.9.1 Making a Proper Constructor and `toString()` Method

When starting out to build a class, we begin by creating a suitable constructor. Along the way, you will need a `toString()` method so you can see what you are doing.

We begin with this crude attempt. We are mimicing our work on the `Point` class we developed earlier.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
    }
}
```

It is easy to see that there will be problems. Suppose a client programmer writes this code.

```
BigInteger a = new BigInteger("256");
BigInteger b = new BigInteger("1024");
BigFraction f = new BigFraction(a, b);
```

It seems ridiculous that this fraction should be stored as 256/1024 when it is in fact 1/4. Hence, it seems we should keep our fractions reduced.

To reduce a fraction, you compute the greatest common divisor of the numerator and denominator and divide it out of both. Notice that the `BigInteger` class computes gcds for you, so we can alter our constructor as folllows.

```
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
    }
```

Let us now see what this looks like.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("256");
> BigInteger b = new BigInteger("1024");
> BigFraction f = new BigFraction(a,b)
> f
BigFraction@6ad20835
>
```

Oops. The built-in `toString()` method is not doing such a great job. Let's override it so it make fractions that look like this: 45/17. Here is our revised class.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
```

```
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
}
```

Now we try our unreduced fraction and find things in a happy state.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("256");
> BigInteger b = new BigInteger("1024");
> BigFraction f = new BigFraction(a,b);
> f
1/4
>
```

There is yet one more thing to do to button this up. This little session should prove convincing.

```
> BigInteger b = new BigInteger("-1024");
> BigFraction f = new BigFraction(a,b);
> f
1/-4
>
```

If we put the negative on the top, the `toString()` method will work nicely. We also get the benefit that we can check fraction equality by just checking for equality of numerator and denomonator.

All we need do is to add something like this to the constructor.

```
if(denom < 0)
{
    denom = -denom;
}
```

However, we are indulging here in illegal operations on `BigInteger`s. Looking on the API page, we can see that there is a `negate()` method that returns a copy of the `BigInteger` with its sign changed. Also, there is a `compareTo` method. The expression

```
foo.compareTo(goo)
```

returns a negative integer if `foo < goo`, a positive integer if `foo > goo` and 0 if `foo == goo`. We integrate these features into our class and we now have

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
}
```

3.10 Overloading the Constructor

Wouldn't it be nice to be able to make a `BigFraction` with ordinary integers. In fact, it would be a smart play to use the `long` type, since a `long` type argument will happily accept an `int`, `short`, or `byte`. We will use `this` to call the main constructor, so we do not have to repeat all of the heavy lifting it does.

To this end, we avail ourselves of the `valueOf` method for `BigInteger`.

```
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
```

While we are here, let's make the (obvious) default.

```
    public BigFraction()
    {
        this(0,1);
    }
```

Finally we shall send an ugly message to the woebegone client programmer who tries to create a `BigFraction` with a zero demoninator. Insert this line in the main constructor, just after `num` and `denom` are initialized.

```
    if(denom.equals(BigInteger.ZERO)
        throw new IllegalArgumentException();
```

This will bring immediate program death to the miscreant client programmer who calls it.

Here is our class with everything added to it.

```
import java.math.BigInteger;
public class BigFraction
{

    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    public BigFraction()
    {
        this(0,1);
```

```
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
}
```

Finally, let's take this all for a test-drive. First we look at our main "workhorse" constuctor.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("1048576");
> BigInteger b = new BigInteger("7776");
> BigFraction f = new BigFraction(a,b)
> f
32768/243
>
```

Our second constructor makes this process less verbose.

```
> BigFraction g = new BigFraction(1048576, 7776)
> g
32768/243
>
```

Here we see our default constructor.

```
> BigFraction z = new BigFraction()
> z
0/1
>
```

Finally we tempt and see death.

```
> BigFraction rotten = new BigFraction(5,0)
java.lang.IllegalArgumentException
    at BigFraction.<init>(BigFraction.java:12)
    at BigFraction.<init>(BigFraction.java:25)
>
```

This exception object will immediately halt any program that is running and that calls the constructor illegally.

3.11 Creating an equals Method

This process is always the same. First do the species test. Then cast the `Object` in the argument list to a `BigFraction`. Once this is done, creating `equals` is easy, since all we need to is to compare equality of numerator and denominator.

```
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
```

Note that since we are comparing `BigIntegers` in the `return` statement, we must use the `equals` method for `BigInteger`.

Now lets take this for a walk. We begin by making some instances.

```
> BigFraction f = new BigFraction(1,3);
> BigFraction g = new BigFraction(1,2);
> BigFraction h = new BigFraction(2,4);
> f
1/3
> g
1/2
> h
1/2
>
```

Notice that none are equal under `==`.

```
> f == g
false
> f == h
false
> g == h
false
>
```

Next, we trot out our shiny new `equals` method.

```
> f.equals(g)
false
> f.equals(h)
```

```
false
> g.equals(h)
true
>
```

Finally, we violate the species test and watch a `false` come right back at us as it should.

```
> f.equals("platypus")
false
>
```

3.12 Hello Mrs. Wormwood! Adding Arithmetic

To as great an extent as posible, we shall imitate the interface that is presented to us by the `BigInteger` class. We need to define four methods: `add`, `subtract`, `multiply`, and `divide`. Each of these methods will take a `BigFraction` as an argument, and will return a `BigFraction`. We begin with addition.

We learned from Mrs. Wormwood that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

The header for our `add` method will be

```
public BigFraction add(BigFraction that)
```

Remember, since we are programing in `BigFraction`, we have a `num` and a `denom` and we are

$$\frac{\texttt{num}}{\texttt{denom}}.$$

We are going to add ourself to `that`. Since `that` is a `BigFraction` has a `num` and a `denom`, too. These are known as `that.num` and `that.denom`.

So, we will wind up doing this little arithmetic arabesque to provide us with a framework for writing the actual code.

$$\frac{\texttt{num}}{\texttt{denom}} + \frac{\texttt{that.num}}{\texttt{that.denom}} = \frac{\texttt{num} * \texttt{that.denom} + \texttt{denom} * \texttt{that.num}}{\texttt{denom} * \texttt{that.denom}}$$

Let's take this a piece at a time, beginning with the first term in the numerator of the sum. We are not allowed to write

```
num*that.denom
```

We have to translate it into the language of `BigInteger`, which says we do the following; we elect to store the result in the `BigInteger term1`.

```
BigInteger term1 = num.multiply(that.denom);
```

Now do the same thing with the second term.

```
BigInteger term2 = denom.multiply(that.num);
```

As a result, the numerator will be

```
term1.add(term2)
```

Now we deal with the denominator

```
BigInteger bottom = denom.multiply(that.denom);
```

Our entire fraction in these terms is

$$\frac{\texttt{term1} + \texttt{term2}}{\texttt{bottom}}.$$

So our `return` statement reads

```
return new BigFraction(term1.add(term2), bottom);
```

Assembing it all we have the completed `add` method.

```
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.add(term2), bottom);
}
```

Let's now do a little test.

```
> BigFraction f = new BigFraction(1,2)
> BigFraction g = new BigFraction(1,3)
> f.add(g)
5/6
>
```

Subtraction is easy, we just change an `add` into a `subtract`.

```
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.subtract(term2), bottom);
}
```

Multiplication is done "straight across."

```
public BigFraction multiply(BigFraction that)
{
    BigInteger top = num.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(top, bottom);
}
```

To divide, invert and multply.

```
public BigFraction divide(BigFraction that)
{
    BigInteger top = num.multiply(that.denom);
    BigInteger bottom = denom.multiply(that.num);
    return new BigFraction(top, bottom);
}
```

Finally, since `BigInteger` has `pow`, we will add that too. We only define this for integer powers, since noninteger powers of rational numbers are seldom rational. If the power is negative, we invert the fraction, strip the sign off of the power, then compute the power.

```
public BigFraction pow(int n)
{
    if(n > 0)
        return new BigFraction(num.pow(n), denom.pow(n));
    if(n == 0)
        return new BigFraction(1,1);
    else
    {
        n = -n;   //strip sign
        return new BigFraction(denom.pow(n), num.pow(n));
    }
}
```

3.13 The Role of static and final

In keeping with the behavior of BigInteger we will make our BigFractions immutable. You will notice that none of our methods we have created so far allow changes in state.

To make this intent clear, we should mark the num and denom state variables final. Since BigIntegers are immutable, this renders the state variables constant. Our class creates immutable objects.

The BigInteger class has static constants ONE and ZERO. We add constants like this to our class as follows. First we create the static objects ONE and ZERO. We shall make them public.

```
public static final BigFraction ZERO;
public static final BigFraction ONE;
```

Place these before the declarations for the state variables in the class. Note: these are *not* state variables, since they reflect a property of the class as a whole, not the state of any particular object. To initialize them, create a static block. You do so as follows.

```
static
{
    ZERO = new BigFraction();
    ONE = new BigFraction(1,1);
}
```

Clients of your class can now use BigFraction.ZERO to get 0 as a BigFraction and BigFraction.ONE to get 1 as a BigFraction.

If you compile now, you will get errors because the constructor performs some reassignments. We can reëngineeer it as follows to get rid of the reassignments.

```
public BigFraction(BigInteger _num, BigInteger _denom)
{
    if(_denom.equals(BigInteger.ZERO))
        throw new IllegalArgumentException();
    BigInteger d = _num.gcd(_denom);
    if(_denom.compareTo(BigInteger.ZERO) < 0)
    {
        _num = _num.negate();
        _denom = _denom.negate();
    }
    num = _num.divide(d);
    denom = _denom.divide(d);
}
```

You will notice that `BigInteger` has a static `valueOf` method that converts `longs` to `BigIntegers`. We now make two static factory methods named `valueOf`. One will take a `long` and promote it to a `BigFraction`. The other will do this service for `BigInteger`.

```
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

Here is the current appearance of the entire class.

```
import java.math.BigInteger;
public class BigFraction
{
    public static final BigFraction ZERO;
    public static final BigFraction ONE;
    static
    {
        ZERO = new BigFraction();
        ONE = new BigFraction(1,1);
    }
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public BigFraction(long _num, long _denom)
    {
```

```
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    public BigFraction()
    {
        this(0,1);
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
    public boolean equals(Object o)
    {
        if(! (o instanceof BigFraction))
            return false;
        BigFraction that = (BigFraction) o;
        return num.equals(that.num) && denom.equals(that.denom);
    }
    public BigFraction add(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.add(term2), bottom);
    }
    public BigFraction subtract(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.subtract(term2), bottom);
    }
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
    public BigFraction divide(BigFraction that)
    {
        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
    public BigFraction pow(int n)
    {
        if(n > 0)
```

```
            return new BigFraction(num.pow(n), denom.pow(n));
        if(n == 0)
            return new BigFraction(1,1);
        else
        {
            n = -n;   //strip sign
            return new BigFraction(denom.pow(n), num.pow(n));
        }
    }
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
}
```

Programming Exercises Add these methods to our existing `BigFraction` class. These will make our `BigFraction`s more resemble `BigIntegers`.

1. Write a method `public BigInteger bigIntValue()` that divides the denominator into the numerator and which truncates towards zero.
2. Write the method `public BigFraction abs()` which returns the absolute value of this `BigFraction`.
3. Write the method `public BigFraction max(BigFraction)` which returns the larger of this `BigFraction` and `that`.
4. Write the method `public BigFraction min(BigFraction)` which returns the smaller of this `BigFraction` and `that`.
5. Write a method `public BigFraction negate()` which returns a copy of this `BigFraction` with its sign changed.
6. Write the method `public int signum()` which returns +1 if this `BigFraction` is positive, -1 if it is negative and 0 if it is zero.
7. Write the method `public int compareTo(BigFraction that)` which returns +1 if this `BigFraction` is larger than `that`, -1 if `that` is larger than this `BigFraction` and 0 if this `BigFraction` equals `that`.
8. Add a static method public BigFraction harmonic(int n) which computes the nth harmonic number. Throw an `IllegalArgumentException` if the client passes an `n` that is negative.
9. (Quite Challenging) Write the method `public double doubleValue()` which returns a floating point value for this `BigFraction`. It should re-

turn `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` where appropriate. Test this very carefully; it is not easy to get it right.

Chapter 4

The Big Fraction Case Study

4.0 Case Study: An Extended-Precision Fraction Class

We have achieved several goals in the last chapter, the most important of which are understanding what makes up a Java class and understanding the core Java language so as to be Turing-complete.

To tie everything together, we will do a case study of creating a class called `BigFraction`, which will work like the `BigInteger` class and provide many of the same operations, except it will do exact fractional arithmetic. This class will have a professional appearance, and an interface similar to that of `BigInteger`.

During this chapter, you will learn about `javadoc`; this allows you to create an API page for your class that will have the same appearance as the page you see on the web. When you are done with this chapter, you will have a class suitable for others to use as clients who wish to perform extended-precision rational arithmetic. The `javadoc` feature can be done at the UNIX command line. It can also be created for you by DrJava.

4.0.1 A Brief Weltanschauung

Before we begin let us remind ourselves of some basic mathematical facts and provide a rationale for what we are about to do. We are all familiar with the *natural* (counting) numbers

$$\mathbb{N} = \{1, 2, 3, 4, ...\}.$$

We can also start counting at zero because we are C family language geeks with

$$\mathbb{N}_0 = \{0, 1, 2, 3, ...\}.$$

The set of all integers (with signs) is often denoted by $\mathbb{Z}$. Why the letter Z? This comes from the German word *zahlen*, meaning "to count."

The `BigInteger` class creates a computational environment for computing in $\mathbb{Z}$ without danger of overflow, unless you really go bananas.

The *rational numbers* consist of all numbers that can be represented as a ratio of integers; the symbol used for them is $\mathbb{Q}$. The Q is for "quotient." So,

$$\mathbb{Q} = \{m/n : m \in \mathbb{Z}, n \in \mathbb{N}\}.$$

The `BigFraction` class will create an environment for computing in $\mathbb{Q}$ similar to that which `BigInteger` provides for $\mathbb{Z}$.

Not all real numbers (which we represent with `double`) are rational. It is a well-known fact that $\sqrt{2}$ and the beloved constant π are not rational. In fact, most of the time you take a square root, you will despoil the rationality of any rational number you operate on.

This explains why `BigInteger`'s `pow` method accepts only non-negative integers. It becomes irate and produces an abrasive run-time error if you attempt to compute a negative power for an `BigInteger`. You will see that when we create a `pow` method for our `BigFractions` it will only accept integers (any integer in fact), but not any other kind of rational number.

We select this case study because it brings to the fore a variety of important design questions. When we are done, we will have a nice facility for computing with fractions. You will get to see the development of a moderately sophisticated class from scratch.

4.1 Start your Engines!

Let us begin by thinking about fractions. We know a fraction has two important items reflecting its state: its numerator and its denominator. Fractions have some slippery properties. For example, we know that

$$\frac{1}{4} = \frac{256}{1024}.$$

The representation of a fraction in terms of numerator and denominator is not unique.

An interesting collection of numbers is the *harmonic numbers*; they are defined by

$$H_n = \sum_{k=1}^{n} \frac{1}{k}, \qquad n \in \mathbb{N}.$$

Let us show the first few harmonic numbers. It is easy to see that $H_1 = 1$. We have

$$H_2 = 1 + \frac{1}{2} = \frac{3}{2}.$$

Next,

$$H_3 = H_2 + \frac{1}{3} = \frac{3}{2} + \frac{1}{3} = \frac{11}{6}.$$

Now let's skip down to H_{10}.

$$H_{10} = \frac{7381}{2520}.$$

One thing is clear: as we keep adding fractions, their numerators and denominators have a propensity to keep getting larger. We know that the primitive `int` and `long` types are not going to cut the mustard here because they will overflow and produce false results.

We will therefore use `BigInteger` for the numerator and denominator of our `BigFraction`s. We should be able to compute H_{100} or even H_{1000}.

4.2 Making a Proper Constructor and `toString()` Method

When starting out to build a class, we begin by creating a suitable constructor. Along the way, you will need a `toString()` method so you can see what you are doing.

We begin with this crude attempt. We are mimicking our work on the `Point` class we developed earlier.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
    }
}
```

It is easy to see that there will be problems. Suppose a client programmer writes this code.

```
BigInteger a = new BigInteger("256");
BigInteger b = new BigInteger("1024");
BigFraction f = new BigFraction(a, b);
```

It seems ridiculous that this fraction should be stored as 256/1024 when it is in fact 1/4. Hence, it seems we should keep our fractions reduced.

To reduce a fraction, you compute the greatest common divisor of the numerator and denominator and divide it out of both. Notice that the `BigInteger` class computes GCDs for you, so we can alter our constructor as follows.

```
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
    }
```

Let us now see what this looks like.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("256");
> BigInteger b = new BigInteger("1024");
> BigFraction f = new BigFraction(a,b)
> f
BigFraction@6ad20835
>
```

Oops. The built-in `toString()` method is not doing such a great job. Let's override it so it make fractions that look like this: 45/17. Here is our revised class.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
```

```
        denom = denom.divide(d);
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
}
```

Now we try our unreduced fraction and find things in a happy state.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("256");
> BigInteger b = new BigInteger("1024");
> BigFraction f = new BigFraction(a,b);
> f
1/4
>
```

There is yet one more thing to do to button this up. This little session should prove convincing.

```
> BigInteger b = new BigInteger("-1024");
> BigFraction f = new BigFraction(a,b);
> f
1/-4
>
```

If we put the negative on the top, the `toString()` method will work nicely. We also get the benefit that we can check fraction equality by just checking for equality of numerator and denominator.

All we need do is to add something like this to the constructor.

```
if(denom < 0)
{
    denom = -denom;
}
```

However, we are indulging here in illegal operations on `BigInteger`s. Looking on the API page, we can see that there is a `negate()` method that returns a copy of the `BigInteger` with its sign changed. Also, there is a `compareTo` method. The expression

```
foo.compareTo(goo)
```

returns a negative integer if `foo < goo`, a positive integer if `foo > goo` and 0 if `foo == goo`. We integrate these features into our class and we now have

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
}
```

4.3 Overloading the Constructor

Wouldn't it be nice to be able to make a `BigFraction` with ordinary integers? In fact, it would be a smart play to use the `long` type, since a `long` type argument will happily accept an `int`, `short`, or `byte`. We will use `this` to call the main constructor, so we do not have to repeat all of the heavy lifting it does.

To this end, we avail ourselves of the `valueOf` method for `BigInteger`.

```
public BigFraction(long _num, long _denom)
{
    this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
}
```

While we are here, let's make the (obvious) default.

```
public BigFraction()
{
    this(0,1);
}
```

Finally we shall send an ugly message to the woebegone client programmer who tries to create a `BigFraction` with a zero denominator. Insert this line in the main constructor, just after `num` and `denom` are initialized.

```
if(denom.equals(BigInteger.ZERO)
    throw new IllegalArgumentException();
```

This will bring immediate program death to the miscreant client programmer who calls it.

Here is our class with everything added to it.

```
import java.math.BigInteger;
public class BigFraction
{

    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    public BigFraction()
    {
        this(0,1);
    }
    public String toString()
    {
        return "" + num + "/" + denom;
    }
```

```
}
```

Finally, let's take this all for a test-drive. First we look at our main "workhorse" constructor.

```
> import java.math.BigInteger;
> BigInteger a = new BigInteger("1048576");
> BigInteger b = new BigInteger("7776");
> BigFraction f = new BigFraction(a,b)
> f
32768/243
>
```

Our second constructor makes this process less verbose.

```
> BigFraction g = new BigFraction(1048576, 7776)
> g
32768/243
>
```

Here we see our default constructor.

```
> BigFraction z = new BigFraction()
> z
0/1
>
```

Finally we tempt and see death.

```
> BigFraction rotten = new BigFraction(5,0)
java.lang.IllegalArgumentException
    at BigFraction.<init>(BigFraction.java:12)
    at BigFraction.<init>(BigFraction.java:25)
>
```

This exception object will immediately halt any program that is running and that calls the constructor illegally.

4.4 Creating an `equals` Method

This process is always the same. First do the species test. Then cast the `Object` in the argument list to a `BigFraction`. Once this is done, creating `equals` is easy, since all we need to is to compare equality of numerator and denominator.

```
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
```

Note that since we are comparing `BigIntegers` in the `return` statement, we must use the `equals` method for `BigInteger`.

Now lets take this for a walk. We begin by making some instances.

```
> BigFraction f = new BigFraction(1,3);
> BigFraction g = new BigFraction(1,2);
> BigFraction h = new BigFraction(2,4);
> f
1/3
> g
1/2
> h
1/2
>
```

Notice that none are equal under `==`.

```
> f == g
false
> f == h
false
> g == h
false
>
```

Next, we trot out our shiny new `equals` method.

```
> f.equals(g)
false
> f.equals(h)
false
> g.equals(h)
true
>
```

Finally, we violate the species test and watch a `false` come right back at us as it should.

```
> f.equals("platypus")
false
>
```

4.5 Hello Mrs. Wormwood! Adding Arithmetic

To as great an extent as possible, we shall imitate the interface that is presented to us by the `BigInteger` class. We need to define four methods: `add`, `subtract`, `multiply`, and `divide`. Each of these methods will take a `BigFraction` as an argument, and will return a `BigFraction`. We begin with addition.

We learned from Mrs. Wormwood that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

The header for our `add` method will be

```
public BigFraction add(BigFraction that)
```

Remember, since we are programming in `BigFraction`, we have a `num` and a `denom` and we are

$$\frac{\texttt{num}}{\texttt{denom}}.$$

We are going to add yourself to `that`. Since `that` is a `BigFraction` has a `num` and a `denom`, too. These are known as `that.num` and `that.denom`.

So, we will wind up doing this little arithmetic arabesque to provide us with a framework for writing the actual code.

$$\frac{\texttt{num}}{\texttt{denom}} + \frac{\texttt{that.num}}{\texttt{that.denom}} = \frac{\texttt{num} * \texttt{that.denom} + \texttt{denom} * \texttt{that.num}}{\texttt{denom} * \texttt{that.denom}}$$

Let's take this a piece at a time, beginning with the first term in the numerator of the sum. We are not allowed to write

```
num*that.denom
```

We have to translate it into the language of `BigInteger`, which says we do the following; we elect to store the result in the `BigInteger term1`.

```
BigInteger term1 = num.multiply(that.denom);
```

Now do the same thing with the second term.

```
BigInteger term2 = denom.multiply(that.num);
```

As a result, the numerator will be

```
term1.add(term2)
```

Now we deal with the denominator

```
BigInteger bottom = denom.multiply(that.denom);
```

Our entire fraction in these terms is

$$\frac{\texttt{term1} + \texttt{term2}}{\texttt{bottom}}.$$

So our `return` statement reads

```
return new BigFraction(term1.add(term2), bottom);
```

Assembling it all we have the completed `add` method.

```
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.add(term2), bottom);
}
```

Let's now do a little test.

```
> BigFraction f = new BigFraction(1,2)
> BigFraction g = new BigFraction(1,3)
> f.add(g)
5/6
>
```

Subtraction is easy, we just change an `add` into a `subtract`.

```
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.subtract(term2), bottom);
}
```

Multiplication is done "straight across."

```
public BigFraction multiply(BigFraction that)
{
    BigInteger top = num.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(top, bottom);
}
```

To divide, invert and multiply.

```
public BigFraction divide(BigFraction that)
{
    BigInteger top = num.multiply(that.denom);
    BigInteger bottom = denom.multiply(that.num);
    return new BigFraction(top, bottom);
}
```

Finally, since `BigInteger` has `pow`, we will add that too. We only define this for integer powers, since non-integer powers of rational numbers are seldom rational. If the power is negative, we invert the fraction, strip the sign off of the power, then compute the power.

```
public BigFraction pow(int n)
{
    if(n > 0)
        return new BigFraction(num.pow(n), denom.pow(n));
    if(n == 0)
        return new BigFraction(1,1);
    else
    {
        n = -n;   //strip sign
        return new BigFraction(denom.pow(n), num.pow(n));
    }
}
```

4.6 The Role of `static` and `final`

In keeping with the behavior of `BigInteger` we will make our `BigFraction`s immutable. You will notice that none of our methods we have created so far allow changes in state.

To make this intent clear, we should mark the `num` and `denom` state variables `final`. Since `BigInteger`s are immutable, this renders the state variables constant. Our class creates immutable objects.

The BigInteger class has static constants ONE and ZERO. We add constants like this to our class as follows. First we create the static objects ONE and ZERO. We shall make them public.

```
public static final BigFraction ZERO;
public static final BigFraction ONE;
```

Place these before the declarations for the state variables in the class. Note: these are *not* state variables, since they reflect a property of the class as a whole, not the state of any particular object. To initialize them, create a static block. You do so as follows.

```
static
{
    ZERO = new BigFraction();
    ONE = new BigFraction(1,1);
}
```

Clients of your class can now use BigFraction.ZERO to get 0 as a BigFraction and BigFraction.ONE to get 1 as a BigFraction.

If you compile now, you will get errors because the constructor performs some reassignments. We can reëngineer it as follows to get rid of the reassignments.

```
public BigFraction(BigInteger _num, BigInteger _denom)
{
    if(_denom.equals(BigInteger.ZERO))
        throw new IllegalArgumentException();
    BigInteger d = _num.gcd(_denom);
    if(_denom.compareTo(BigInteger.ZERO) < 0)
    {
        _num = _num.negate();
        _denom = _denom.negate();
    }
    num = _num.divide(d);
    denom = _denom.divide(d);
}
```

Now you should be glad you used this in the sibling constructors. No modification of these is necessary.

You will notice that BigInteger has a static valueOf method that converts longs to BigIntegers. We now make two static factory methods named valueOf. One will take a long and promote it to a BigFraction. The other will do this service for BigInteger.

```
public static BigFraction valueOf(long n)
```

```
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

Here is the current appearance of the entire class.

```
import java.math.BigInteger;
public class BigFraction
{
    public static final BigFraction ZERO;
    public static final BigFraction ONE;
    static
    {
        ZERO = new BigFraction();
        ONE = new BigFraction(1,1);
    }
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        num = _num;
        denom = _denom;
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    public BigFraction()
```

```
{
    this(0,1);
}
public String toString()
{
    return "" + num + "/" + denom;
}
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.add(term2), bottom);
}
public BigFraction subtract(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.subtract(term2), bottom);
}
public BigFraction multiply(BigFraction that)
{
    BigInteger top = num.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(top, bottom);
}
public BigFraction divide(BigFraction that)
{
    BigInteger top = num.multiply(that.denom);
    BigInteger bottom = denom.multiply(that.num);
    return new BigFraction(top, bottom);
}
public BigFraction pow(int n)
{
    if(n > 0)
        return new BigFraction(num.pow(n), denom.pow(n));
    if(n == 0)
        return new BigFraction(1,1);
```

```
        else
        {
            n = -n;   //strip sign
            return new BigFraction(denom.pow(n), num.pow(n));
        }
    }
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
}
```

Programming Exercises Add these methods to our existing `BigFraction` class. These will make our `BigFraction`s more resemble `BigIntegers`.

1. Write a method `public BigInteger bigIntValue()` that divides the denominator into the numerator and which truncates towards zero.
2. Write the method `public BigFraction abs()` which returns the absolute value of this `BigFraction`.
3. Write the method `public BigFraction max(BigFraction)` which returns the larger of this `BigFraction` and `that`.
4. Write the method `public BigFraction min(BigFraction)` which returns the smaller of this `BigFraction` and `that`.
5. Write a method `public BigFraction negate()` which returns a copy of this `BigFraction` with its sign changed.
6. Write the method `public int signum()` which returns +1 if this `BigFraction` is positive, -1 if it is negative and 0 if it is zero.
7. Write the method `public int compareTo(BigFraction that)` which returns +1 if this `BigFraction` is larger than `that`, -1 if `that` is larger than this `BigFraction` and 0 if this `BigFraction` equals `that`.
8. Add a static method public BigFraction harmonic(int n) which computes the nth harmonic number. Throw an `IllegalArgumentException` if the client passes an `n` that is negative.
9. (Quite Challenging) Write the method `public double doubleValue()` which returns a floating point value for this `BigFraction`. It should return `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` where appropriate. Test this very carefully; it is not easy to get it right.

4.7 Using Javadoc

The kind of class we have created represents a real extension of the Java language that could be useful to others. Now we need to give our class an API page so it has a professional appearance and so it can easily be used by others.

Javadoc comments are delimited by the starting token `/**` and the ending token `*/`. C and C++ style comments delimited by `//` and `/* ..... */` do not appear on Javadoc pages.

You may use HTML markup in your javadoc where needed.

Use Javadoc to document your *interface*, the public portion of your class. Do not javadoc `private` methods or state variables.

We will produce a full javadoc page for our `BigFraction` class. Let us begin with the constructors.

```
 /**
 * This constructor stores a <tt>BigFraction</tt> in
 * reduced form, with any negative factor appearing in
 * the numerator.
 * @param _num the numerator of the <tt>BigFraction</tt>
 * @param _denom the denominator of the <tt>BigFraction</tt>
 * @throws <tt>IllegalArgumentException</tt> if the creation
 * of a zero-denominator <tt>BigFraction</tt> is attempted.
 */
public BigFraction(BigInteger _num, BigInteger _denom)
{
    if(_denom.equals(BigInteger.ZERO))
        throw new IllegalArgumentException();

    BigInteger d = _num.gcd(_denom);
    if(_denom.compareTo(BigInteger.ZERO) < 0)
    {
        _num = _num.negate();
        _denom = _denom.negate();
    }
    num = _num.divide(d);
    denom = _denom.divide(d);
}
/**
 * This creates the <tt>BigFraction</tt> <tt>_num/_denom</tt>
 * This fraction will be fully reduced and any negative factor
 * appears in the numerator.
 * @param _num the numerator
 * @param _denom the denominator
```

```
     * @throws <tt>IllegalArgumentException</tt> if the creation
     * of a zero-denominator <tt>BigFraction</tt> is attempted.
     */
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    /**
     * This default constructor produces BigFraction 0/1.
     */
    public BigFraction()
    {
        this(0,1);
    }
```

We see the special markup `@param`; this is the description given for each parameter. The markup `@throws` warns the client that an exception can be thrown by a method. You should always tell exactly what triggers the throwing of an exception, as the penalty for an exception can be program death.

4.7.1 Triggering Javadoc

First we give instructions for DrJava. Bring up the Preferences by hitting control-; or by selecting the Preferences item from the bottom of the Edit menu. Under Web browser put the path to your web browser. An example of a valid path is

```
/usr/lib/firefox/firefox.sh
```

If you use Windoze, your path should begin with `\tt C:\`. If you use a Mac, it will be in your Applications folder. You can browse for it by hitting the `...` button just to the right of the Web Browser text field.

The javadoc will be saved in a directory called `doc` that is created in same directory as your class's code. Allow the javadoc to be saved in that folder, or files will "spray" all over your directory and make a big mess.

You can also javadoc at the command line with

```
$ javadoc -d someDirectory BigFraction.java
```

The javadoc output will be placed in the directory `someDirectory` that you specify. Make sure you use the `-d` option to avoid spraying.

Do this DrJava will open your shiny new Javadoc page in the browser you anointed for that purpose. Scroll down to the constructors area and you will see your documentation. Click on each constructor and see its method detail.

In either case, the program *must* compile before any javadoc will be generated.

I don't see my javadoc! Make sure you are using the javadoc comment tokens like so.

```
/**
*    stuff
*/
```

and not regular multiline comment token that look like this.

```
/*
*    stuff
*/
```

4.7.2 Documenting `toString()` and `equals()`

You will see a new markup device `@return` and `overrides` which tells you what these methods override. You will notice if you look in the javadoc you generated, that an `overrides` tag is already in the method detail.

```
    /**
    * @return a string representing this BigFraction of  the form
    * numerator/denominator.
    */
   @Override
   public String toString()
   {
       return "" + num + "/" + denom;
   }
```

Note the use of the `@Override` construct just after our javadoc markup. This is called an *annotation*, and the compiler checks that you have used the right signature to actual override the method. If you don't it will be flagged as a compiler error. Always use this annotation if you are implementing the methods `public boolean equals(Object o)` or `public String toString()`.

Now we deal similarly with the `equals` method.

```
    /**
    * @param o an Object we are comparing this BigFraction to
    * @return true iff this BigFraction and that are equal numerically.
    * A value of <tt>false</tt> will be returned if the Object o is not
```

```
     * a BigFraction.
     */
    @Override
    public boolean equals(Object o)
    {
        if(! (o instanceof BigFraction))
            return false;
        BigFraction that = (BigFraction) o;
        return num.equals(that.num) && denom.equals(that.denom);
    }
```

4.7.3 Putting in a Preamble and Documenting the Static Constants

We show where to preamble goes, after the imports and before the head for the class. Place a succinct description of your class here to let your clients know what it does.

```
import java.math.BigInteger
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <tt>add</tt> method,
 * - with the <tt>subtract</tt>
 * method, * with the <tt>multiply</tt> method,
 * and / with the <tt>divide</tt> method.
 * It computes integer powers
 * of fractions using the <tt>pow</tt> method.
 */
public class BigFraction
{
    //code
}
```

Documenting the static constants is very straightforward.

```
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
```

```
 */
public static final BigFraction ONE;
```

4.7.4 Documenting Arithmetic

Next we javadoc all of the arithmetic operations we have provided the client. Notice how we add an exception if the client attempts to divide by zero.

```
/**
 * This add BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <tt>this</tt> + <tt>that</tt>
 */
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.add(term2), bottom);
}
/**
 * This subtracts BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <tt>this</tt> - <tt>that</tt>
 */
public BigFraction subtract(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.subtract(term2), bottom);
}
/**
 * This multiplies BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <tt>this</tt> * <tt>that</tt>
 */
public BigFraction multiply(BigFraction that)
{
    BigInteger top = num.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(top, bottom);
}
/**
 * This divides BigFractions.
```

```
 * @param that a BigFraction we are adding to this BigFraction
 * @return <tt>this</tt>/<tt>that</tt>
 * @throws <tt>IllegalArgumentException</tt> if division by
 * 0 is attempted.
 */
public BigFraction divide(BigFraction that)
{
    if(that.equals(BigFraction.ZERO))
        throw new IllegalArgumentException();
    BigInteger top = num.multiply(that.denom);
    BigInteger bottom = denom.multiply(that.num);
    return new BigFraction(top, bottom);
}
 /**
 * This computes an integer power of BigFraction.
 * @param n an integer power
 * @return <tt>this</tt><sup><tt>n</tt></sup>
 */
public BigFraction pow(int n)
{
    if(n > 0)
        return new BigFraction(num.pow(n), denom.pow(n));
    if(n == 0)
        return new BigFraction(1,1);
    else
    {
        n = -n;   //strip sign
        return new BigFraction(denom.pow(n), num.pow(n));
    }
}
```

Finally, we will take care of our two `valueOf` methods.

```
 /**
 * @param n a long we wish to promote to a BigFraction.
 * @return A BigFraction object wrapping n
 */
public static BigFraction valueOf(long n)
{
    return new BigFraction(n, 1);
}
/**
* @param num a BigInteger we wish to promote to a BigFraction.
* @return A BigFraction object wrapping num
*/
public static BigFraction valueOf(BigInteger num)
```

```
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

4.7.5 The Complete Code

Here it is!

```
import java.math.BigInteger;
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <tt>add</tt> method,
 * - with the <tt>subtract</tt>
 * method, * with the <tt>multiply</tt> method,
 * and / with the <tt>divide</tt> method.
 * It computes integer powers
 * of fractions using the <tt>pow</tt> method.
 */
public class BigFraction
{
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
     */
    public static final BigFraction ONE;

    static
    {
        ZERO = new BigFraction();
        ONE = new BigFraction(1,1);
    }
    private final BigInteger num;
    private final BigInteger denom;
    /**
     * This constructor stores a <tt>BigFraction</tt> in
     * reduced form, with any negative factor appearing in
     * the numerator.
     * @param _num the numerator of the <tt>BigFraction</tt>
     * @param _denom the denominator of the <tt>BigFraction</tt>
     * @throws <tt>IllegalArgumentException</tt> if the creation
```

```
     * of a zero-denominator <tt>BigFraction</tt> is attempted.
     */
    public BigFraction(BigInteger _num, BigInteger _denom)
    {
        if(_denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = _num.gcd(_denom);
        if(_denom.compareTo(BigInteger.ZERO) < 0)
        {
            _num = _num.negate();
            _denom = _denom.negate();
        }
        num = _num.divide(d);
        denom = _denom.divide(d);
    }
    /**
     * This creates the <tt>BigFraction</tt> <tt>_num/_denom</tt>
     * This fraction will be fully reduced and any negative factor
     * appears in the numerator.
     * @param _num the numerator
     * @param _denom the denominator
     * @throws <tt>IllegalArgumentException</tt> if the creation of a
     * zero-denominator <tt>BigFraction</tt> is attempted.
     */
    public BigFraction(long _num, long _denom)
    {
        this(BigInteger.valueOf(_num), BigInteger.valueOf(_denom));
    }
    /**
     * This default constructor produces BigFraction 0/1.
     */
    public BigFraction()
    {
        this(0,1);
    }
    /**
     * @return a string representing this BigFraction of  the form
     * numerator/denominator.
     */
    @Override
    public String toString()
    {
        return "" + num + "/" + denom;
    }
    /**
```

```
     * @param o an Object we are comparing this BigFraction to
     * @return true iff this BigFraction and that are equal numerically.
     * A value of <tt>false</tt> will be returned if the Object o is not
     * a BigFraction.
     */
    @Override
    public boolean equals(Object o)
    {
        if(! (o instanceof BigFraction))
            return false;
        BigFraction that = (BigFraction) o;
        return num.equals(that.num) && denom.equals(that.denom);
    }
    /**
     * This add BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <tt>this</tt> + <tt>that</tt>
     */
    public BigFraction add(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.add(term2), bottom);
    }
    /**
     * This subtracts BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <tt>this</tt> - <tt>that</tt>
     */
    public BigFraction subtract(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.subtract(term2), bottom);
    }
    /**
     * This multiplies BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <tt>this</tt> * <tt>that</tt>
     */
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
```

```
        return new BigFraction(top, bottom);
    }
    /**
     * This divides BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <tt>this</tt>/<tt>that</tt>
     * @throws <tt>IllegalArgumentException</tt> if division by
     * 0 is attempted.
     */
    public BigFraction divide(BigFraction that)
    {
        if(that.equals(BigFraction.ZERO))
            throw new IllegalArgumentException();
        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
      /**
     * This computes an integer power of BigFraction.
     * @param n an integer power
     * @return <tt>this</tt><sup><tt>n</tt></sup>
     */
    public BigFraction pow(int n)
    {
        if(n > 0)
            return new BigFraction(num.pow(n), denom.pow(n));
        if(n == 0)
            return new BigFraction(1,1);
        else
        {
            n = -n;   //strip sign
            return new BigFraction(denom.pow(n), num.pow(n));
        }
    }
    /**
     * @param n a long we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping n
     */
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
     /**
     * @param num a BigInteger we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping num
     */
```

```
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
}
```

Programming Exercises

1. Add javadoc for all of the methods you wrote in the previous set of programming exercises.
2. Write a second class called `TestBigFraction`. Place a `main` in this class and have it test `BigFraction` and its methods. Place the classes in the same directory.

Chapter 5

Interfaces, Inheritance and Java GUIs

5.0 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 two–class programs. One class has been the class you are writing, the other has been the interactions pane.

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

To get started, 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.

5.1 A Short GUI Program

We shall also begin to explore the Java GUI classes. 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.

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

- `java.awt` This is the "old brain" of Java GUIs.
- `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` This package holds classes that are useful in responding to such things as keystrokes, mouse clicks, and the selection of menu items. Things in this package 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;
> 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. 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);
}
```

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.

```
> 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 `getContenPane()`, which returns 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 deconstruct things, we will see it is very understandable. Look in the API guide. The call

```
f.getContentPane()
```

returns 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 from the interactions pane is to set the frame to be invisible, then to be visible.

```
> 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 paine. Now let us look at the idea of inheritance in general.

5.2 Inheritance

Inheritance provides a mechanism by which we can customize the capabilities of 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. We will discuss the problems with multiple inheritance schemes shortly.

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 and discuss better ways of doing things later.

Now we will create a `Rectangle` class.

```
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);
    }
```

```
}
```

Then 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`.

```
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 `Squre` is a `Rectangl`! 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.

You might ask now, "Why not make everything an `Object` and save a lot of work?" 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 tradeoff 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
>
```

5.2.1 Polymorphism and Delegation

How does this polymorphism thing work? We had the `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.

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.

- **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.

5.2.2 Understanding More of the API Guide

Go to the API guide and bring up `JFrame`. Here is the family tree 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 libaries and the inheritance structure are two different structures.

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 are just another name for state or instance variables. You will notice that many of these are in caps. It is common to put a variable name in caps when the variable is marked `final`.

Fields can be inherited from ancestor classes and these are also listed on the API page. One 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.

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`. You will learn how to use `super` to initialize state variables in a parent class. You will see the `protected` keyword fairly often in the API guide.

5.2.3 Deprecated Can't be Good

Do not use deprecated elements. This is the Java community's way of telling you something is on the way out. Often, when something is deprecated, the API guide will indicate the correct way to accomplish your goal.

5.2.4 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.

5.2.5 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 inheri-

tance." The ability to inherit is very valuable, so we should only inherit when it yields significant benefits with little downside. We will see how to circumnavigate this and obtain the benefits of polymorphism with interfaces. First, we shall dispense with an important technical detail.

5.3 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`.

`final` **Exam!**	
`primitive`	When a variable of primitive type is marked final, it is immutable
`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 a final object. What is immutable here is the pointing relationship between the identifier marked `final` and its 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.

5.4 Back to the '70's with Cheesy Paneling, or I Can't Stand it, Call the Cops!

In this section, we will return to the graphical world We will produce a more elaborate example of a simple GUI app. This will allow us to introduce some new Swing classes and apply them. Also, watch for a nice, natural use of a two–dimensional array to arise. The result will be part of an interface to a graphical calculator.

Java supplies a class called a `JPanel` that is an ideal tool for corralling a group of related graphical widgets. You can add `JPanel`s to the content pane. You can also add a `Container`. Remember, this is the type of the content pane of a `JFrame`. The problem is, that graphical widgets are like badly–behaved children. They don't play nicely without supervision.

To see this, let us repair to the interactions pane for an enlightening session in which we try to place two buttons in a window. We begin by building the frame and adding the left button.

```
> import javax.swing.JFrame
> import javax.swing.JButton
> f = new JFrame("Two buttons, I hope");
> f.setSize(500,500);
> left = new JButton("left");
> 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 ironic.

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 first chick in the nest kills its sibling so it gets all the food. This is a problem. How do we get these (childish) widgets to play nicely?

A cop is needed. The cop comes in the form of a *layout manager*, who tells the widgets how to play nicely. It will lay down the law. Widgets by themselves are terrible children who try to hog everything for themselves. They are undisciplined and will appear anywhere but where you want them. They need a layout manager to impose order and to make things work. Now hit F5 to clear the interactions pane and we shall start from scratch.

We will use a `GridLayout`; the full name of this class is

```
java.awt.GridLayout
```

Here is our interactive session. We will "peek" as we build the GUI. You can open the `GridLayout` API page. It forces widgets to occupy a grid. We wil make four buttons and we will peek after each one is added to see what is happening. Begin by creating the window. Keep this session open, as we will add to it.

```
> import java.awt.GridLayout;
> import javax.swing.JFrame;
> import javax.swing.JButton;
> f = new JFrame("Four Buttons Playing Nicely");
> f.setSize(500,500);
> f.setVisible(true);
```

Now add this line. It will make the widgets live in a 2×2 grid.

```
> f.getContentPane().setLayout(new GridLayout(2,2));
```

We add theses lines. They add the `JButton` labeled 0 to the window. Notice that this lone button occupies the upper half of the content pane.

```
> b00 = new JButton("0");
```

```
> f.getContentPane().add(b00);
> f.setVisible(false);
> f.setVisible(true);
```

Now add the second button. Set visible to false then true to refresh everything.

```
> b01 = new JButton("1");
> f.getContentPane().add(b01);
> f.setVisible(false);
> f.setVisible(true);
```

The result is two buttons, one occupying the top half of the content pane, the other occupying the bottom half. Next, we add the third button to the content pane.

```
> b10 = new JButton("2");
> f.getContentPane().add(b10);
> f.setVisible(false);
> f.setVisible(true);
```

The result here is that the buttons marked 0 and 1 occupy the top row and that the button marked 2 occupies the left half of the bottom row. Now let us administer the coup d'grace.

```
> b11 = new JButton("3");
> f.getContentPane().add(b11);
> f.setVisible(false);
> f.setVisible(true);
```

Voila! Four buttons are playing nicely in a 2×2 grid. The unruly children have been disciplined into their appropriate roles.

There are several types of layout managers we will be interested in. You should instantiate them all and, add buttons and watch them work.

Layout Managers	
`null`	This is an absence of a layout manager. You 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 incredible 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.

5.4.1 Recursion is our Friend

You can create `JPanel`s, 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.

For example, if we were writing a calculator, we might have a panel hold the a 4X3 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.

5.5 A Framework for our GUI Programs

We are going to create a simple framework for creating 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 subprocess 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. Having several threads running at once is called *multithreading*. Every thread has its own call stack which is independent of the other threads.

Whether you know it or not, all Java programs of any size at all are multithreaded. The garbage collector runs it its own thread, monitoring your program for orphaned objects and deallocating their memory. 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 computercome in the form of *events* Events include such things as keystrokes, mouse clicks, and button pushes.

Java manages events with a data structure called a *queue*. Queues are 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 logicaly 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.

To accomplish this we must do something called "implementing the Runnable interface." Happily this is quite uncomplicated. For now, that means two things, insert `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);
    }
}
```

The other thing required is for you to have a `run` method tht 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

```
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 must orchestrate the enire 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. Let us now return to building a simple application that has holds the number and operator keys for a calculator.

5.6 Creating a Complex View

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
{
}
```

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();
        numberPanel.setLayout(new GridLayout(4,3));
        opPanel = new JPanel();
        opPanel.setLayout(new GridLayout(5,1));
        getContentPane().add(BorderLayout.CENTER, numberPanel);
        getContentPane().add(BorderLayout.EAST, opPanel);
    }
}
```

So far, nothing is visible. Let us now put in a `main` method and a `run` method to build the GUI.

```
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();
        opPanel = new JPanel();
    }
    public void run()
    {
        opPanel.setLayout(new GridLayout(5,1));
        numberPanel.setLayout(new GridLayout(4,3));
        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);
    }

}
```

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

```
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();
```

```
        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);
    }

}
```

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. It will now be necessary to get this live by initializing 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.

5.7 Interfaces

In the last section, we created a GUI with buttons, and in the exercises, you made an application with menus. However, these pretty things *do nothing*. How do we make feature like buttons live? How do we get our programs to respond to mouse clicks or keyboard hits? To accomplish these goals, we will first need to understand interfaces. The discussion ahead will at first seem totally unrelated to the issue of GUIS, but in the next section, all will snap into place.

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. The waste is that the code in the `Shape` class is useless. You cannot compute geometric quantities of a shape without first knowing what kind of shape it actually is.

Let us now create an interface, which we shall call `IShape` for handling shapes. We shall decided that the essence of being a shape here is knowledge of your diameter, perimeter and area. Save it 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. You are not allowed to have any code inside of an interface. You may only place method headers in it.

An interface is a contract in Java. You can sign the contract as follows. You know, for instance, that a `Rectangle` should be a `IShape`. To make this so, 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. This contract is enforced by the compiler.

You have already used his construct: the `Runnable` interface has only one method: `public void run()`.

Interfaces are not classes. 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
```

Successfully, the variable of interface type pointed at all of the different shapes. 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
```

You cannot create instances of interfaces.

5.7.1 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`.

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.

You will see that the `ActionListener` will be the tool we use to make a button live. First we have some simple exercises to get you used to interfaces.

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.

5.8 Making a JButton live with ActionListener

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("AAAAAAAAAAAAAAAAAAAAHHHH!!");
    }
}
```

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.

5.9 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 refreshe when they are maximized, minimized or resized.

So, to get started we create a class such as `DrawPanel.java` shown here.

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

public class DrawPanel extends JPanel
{
    @Override
    public void paintComponent(Graphics g)
    {
        //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 resizing the window. The entire window will be filled with black. Why? When the window is resized, 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());
```

Resize the window and watch the `stdout` window. As the window resizes, 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 red rectangle that resizes 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);
}
```

5.10 Abstract Classes

You see now that we have classes, which are blueprints for manufacturing objects. They are blueprints for creating objects that have state, identity and behavior. A logical question is this: *Is is possible to "partially implement" a class?* Can you create a class stub that has some methods implemented, some data members and some unimplemented methods?

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. It is called an *abstract class*. 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.

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 it class hierarchy. All classes live in a family tree. All of our farm classes descend from the root class tt FarmAsset.

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

This looks like a regular class, save for the addition of the word `abstract`. Since this class is marked `abstract`, it cannot be instantiated. You must produce a new descendant class to make an actual instance. 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");
    }
    public String toString()
    {
        return "Pig named " + getName();
    }
}
```

```
i> p = new Pig("Wilbur");
> p.getNoise()
```

```
"Reeeeet! Snort! Snuffle!"
> p.getName()
"Wilbur"
> p.getMeatName()
"pork"
> p
Pig named Wilbur
```

Can Methods be Left Unimplemented? Yes. If you do not implement a method, you can declare it `abstract` and just specify a method header. Any concrete child class will be *required* to implement that method. In this way an abstract class can behave like a class and like an interface. If a class has any abstract methods, it must be declared abstract.

Here is a 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.

Chapter 6

The Tricolor Case Study

6.0 Introduction

This chapter consists of a case study. We are going to get a fully functioning application to draw shapes in its content pane called `Tricolor`. It will be menu-driven and fully graphical.

6.1 Building the View for Tricolor

The `Tricolor` application will feature three menus: File, Color and Position. The File menu will have one item, quit, which will quit the application. The color menu will have three colors: red, green and blue. The Position menu will have three positions, left middle and right.

The content pane will contain three panels, a left, middle and right panel. When a panel is selected, the panel will be painted the color of the item in the Color menu. All panels will be white when the app starts.

We begin by creating a graphical shell as follows.

```
import javax.swing.JFrame;

public class Tricolor extends JFrame implements Runnable
{
    public Tricolor()
    {
        super("Tricolor");
    }
    public void run()
```

```
    {
        setSize(500,500);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

6.2 Our Panels Need to Know their Colors

Now it is time to make some design decisions. We need three panels that know their color. To this end, we create a new class called `ColorPanel`, which extends `JPanel` and which knows its color. It will paint itself its color.

From this point, we know that we will need to change the color of these panels. Since graphical elements tend to be big, we favor mutability here and allow changes of state. We will allow this panel to change its color.

```
import javax.swing.JPanel;
public class ColorPanel extends JPanel
{
    private Color color;
}
```

Let us now make this panel white in the constructor. Also, let us implement a method `setColor` that sets the color of this panel. The color of this panel will be controlled by the items in the Color menu. As a result, we know now that it must be mutable. We insert the method `setColor` so the panel's color can change in response to the selection of a menu item.

```
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Graphics;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
    {
        color = Color.WHITE;
    }
    public void setColor(Color c)
```

```
    {
        color = c;
    }
}
```

Finally, we will tell the panel to paint itself its color. While we are here, let's create a `toString()` method in case we want to check anything along the way. Note the use of the `@Override` annotation to get the compiler to check that we are overriding correctly.

```
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Graphics;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
    {
        color = Color.WHITE;
    }
    public void setColor(Color c)
    {
        color = c;
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(color);
        g.fillRect(0,0,getWidth(), getHeight());
    }
    @Override
    public String toString()
    {
        return "I am a color panel of color" + color;
    }
}
```

6.3 Inserting ColorPanels into the Tricolor App

The `Tricolor` app will now have four state variables. One will be for each of the three panels. The fourth will be for a variable that points at the current panel. By default, and before the Position menu is created, we will make the current panel point at the left panel.

```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

public class Tricolor extends JFrame implements Runnable
{
    private ColorPanel leftPanel;
    private ColorPanel middlePanel;
    private ColorPanel rightPanel;
    private ColorPanel currentPanel;

    public Tricolor()
    {
        super("Tricolor");
        leftPanel = new ColorPanel();
        middlePanel = new ColorPanel();
        rightPanel = new ColorPanel();
        currentPanel = leftPanel;
    }
    public void run()
    {
        setSize(500,500);
        Container c = getContentPane();
        c.setLayout(new GridLayout(1,3));
        c.add(leftPanel);
        c.add(middlePanel);
        c.add(rightPanel);
        setVisible(true);
    }
```

If you have doubt as to whether the `ColorPanel`s got inserted, just go into the `ColorPanel` constructor and temporarily set it to be green. Then it will be obvious. The `Tricolor` app now knows four panels: the three in the content pane, plus the one to be colored.

6.4 Le Carte

It's time to begin making the menus. We will produce the menu bar and the menus. This is akin to carpentry. We must make and nail in the menu bar, a container in which the menus live. Then we install the menus. So let us begin.

We will create a method called `makeMenus` to keep `run` from becoming a run-on method. So, add the call `makeMenus()` to your `run` method. Your `run` method should look like this. As long as you place this before `setVisible`, it does not matter greatly where you put it.

```
public void run()
{
    setSize(500,500);
    makeMenus();
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,3));
    c.add(leftPanel);
    c.add(middlePanel);
    c.add(rightPanel);
    setVisible(true);
}
```

Now we will make the menu bar and nail it in. Place this code in `makeMenus`.

```
JMenuBar mbar = new JMenuBar();
setJMenuBar(mbar);
```

The `JMenuBar` is a container that holds widgets. In particular it holds `JMenu`s. We make these and add them using ... duh ... `add`.

```
private void makeMenus()
{
    JMenuBar mbar = new JMenuBar();
    setJMenuBar(mbar);
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    JMenu positionMenu = new JMenu("Position");
    mbar.add(positionMenu);
}
```

Run this. You will see menu headers living in the menu bar. Before we begin generating menu items, we need to *think*. You will now see an app with three menus and its three panels put into place.

Now we begin with the Color menu. We will create menu items that are smart enough to *know their colors*. Be warned that this class will undergo an evolution as we proceed.

When designing a class ask: What does the class need to know? This tells us what its state must be. Then ask: what does the class need to do? This tells us what methods we need to write.

Note we created a `ColorPanel` that knows its color. We will now create a `ColorMenuItem` that knows its color. Since this color will be permanently assigned to this menu item, we will mark it `final`.

Note that because there is no canonical association between colors and their names, our constructor must allow us to pass both the color and its name. Since we can pass the color name off to the parent constructor, we never bother to store it as a state variable. We can do this if the need arises. But it will never do so in this example.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    public ColorMenuItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
    }
}
```

6.5 It's time to build the controller!

We begin with the low-hanging fruit. Our sole item in the File menu is a quit item. So let us make that. Just add these two lines into the `makeMenus()` method just after you create and add the File menu.

```
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(quitItem);
```

To make this live we will need an action listener that shuts the app down when its `actionPerformed` code gets called. We will call this class `QuitListener`. In the file `QuitListener.java`, place the following code.

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

public class QuitListener implements ActionListener
{
```

```
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

This small class is all we need. Now we attach an instance of it to `quitItem` as follows.

```
quitItem.addActionListener(new QuitListener());
```

Insert this line of code in `makeMenus()` just after you add the `quitItem` to the File menu. Now run the code. Pull down the `File` menu and choose `Quit`. The application will quit! We have the first element of the controller.

6.6 The Color Menu and the Controller

Next we will take care of the Color menu. You might think we are going to do this.

1. Create the three menu items.
2. Create a listener class
3. If the red menu item is chosen, turn the panel red.
4. If the green menu item is chosen, turn the panel green.
5. If the blue menu item is chosen, turn the panel blue.

But that is *not* going to happen. Every time we want to add a new color, we are forced to do a lot of work. Another way to proceed might be to create separate listeners for each item. Then, when that item is chosen, that item's color is used. All of the listeners would look alike. This is a violation of the 11th commandment: *Thou shalt not maintain duplicate code!* What, then, is the best course of action?

Remember: The power to delegate is the power to accomplish! We will make the menu items responsible for knowing their colors and for having a listener that causes their colors to be used.

Previously we created `ColorMenuItem.java`.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
```

```
{
    private final Color color;
    public ColorMenuItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
    }
}
```

We can populate the Color menu with items that are instances of this class. Add these lines to `makeMenus` after you add the Color menu.

```
    colormenu.add(new ColorMenuItem(Color.RED, "red"));
    colormenu.add(new ColorMenuItem(Color.GREEN, "green"));
    colormenu.add(new ColorMenuItem(Color.BLUE, "blue"));
```

Once you do this, run your program and see that the menu items are now present. Note that their controllers are not yet written, so they do not yet work.

Next, we must begin to write a controller for a color menu item. Begin by creating this new class, `ColorMenuItemListener.java`. Our color menu items must know their colors, so we included that as a state variable. We also include a constructor to initialize the state variable. So our class compiles we put in the `actionPerformed` method required by the `ActionListener` interface. We put in some code that prints to `stdout` so we can test if the listener is properly attached when the time comes.

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

public class ColorMenuItemListener implements ActionListener
{
    private final Color color;
    public ColorMenuItemListener(Color _color)
    {
        color = _color;
    }
    public void actionPerformed(ActionEvent e)
    {
        //tell the current panel to change color
        System.out.println("fooment");
    }
}
```

Now that we have a listener, we can go back to our `ColorMenuItem` class and attach a listener as follows in the last line of the constructor.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    public ColorMenuItem(Color _color, String _colorName)
    {
        super(_colorName);
        color = _color;
        addActionListener(new ColorMenuItemListener(color));
    }
}
```

Now compile all and run. When you select a `ColorMenuItem`, you will see `"foomen`t put to `stdout`. But life is not that simple.

Next, we need to think about the code needed to make the listener work. We must think about the lines of communication necessary to accomplish this. The listener must know the current panel and tell it to change its color. There is a problem: `ColorMenuItemListener` has no way to communicate back to the application object, `Tricolor`. The `ColorMenuItem` is the intermediary: it needs to know of `Tricolor` so it can pass it on to its listener.

How do we set that up? We add a state variable that is a `Tricolor`. We will make this `final` since there will be only one `Tricolor` as the program runs. So Let us modify the menu item class first. We insert a new state variable and initialize it in the constructor.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    private final Tricolor tc;
    public ColorMenuItem(Color _color, String _colorName, Tricolor _tc)
    {
        super(_colorName);
        color = _color;
        tc = _tc;
        addActionListener(new ColorMenuItemListener(tc));
    }
}
```

Compile and see brokenness. Look in `makeMenus()` to find the offending code. We have changed the constructor for the `ColorMenuItem` and must therefore fix the constructor calls in the calling routine `makeMenus()` in `Tricolor`. Here is how they currently look

```
colormenu.add(new ColorMenuItem(Color.RED, "red"));
colormenu.add(new ColorMenuItem(Color.GREEN, "green"));
colormenu.add(new ColorMenuItem(Color.BLUE, "blue"));
```

Modify them to this. You are now passing an instance of the `Tricolor` class to each of them.

```
colormenu.add(new ColorMenuItem(Color.RED, "red", this));
colormenu.add(new ColorMenuItem(Color.GREEN, "green", this));
colormenu.add(new ColorMenuItem(Color.BLUE, "blue", this));
```

We are not done yet. Now we must modify the constructor of the `ColorMenuItemListener` to accept a reference to a `Tricolor` as follows.

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

public class ColorMenuItemListener implements ActionListener
{
    private final Tricolor tc;
    private final Color color;
    public ColorMenuItemListener(Tricolor _tc, Color _color)
    {
        tc  = _tc;
        color = _color;
    }
    public void actionPerformed(ActionEvent e)
    {
        //tell the current panel to change color
        //System.out.println("fooment");
    }
}
```

and then change the constructor calls to this class back in `ColorMenuItem`. We just modify the line where we add the action listener.

```
addActionListener(new ColorMenuItemListener(tc, color));
```

We need in the `actionPeformed` method to be able to get the current panel and change its color. We need the following accessor in `Tricolor`.

```
    public ColorPanel getCurrentPanel()
    {
        return currentPanel;
    }
```

To tell the current panel to change to our color, we use

```
tc.getCurrrentPanel().setColor(color);
```

Place this code in the listener to see the following

```
    tc.getCurrentPanel().setColor(color);
    tc.repaint();
```

We use the `repaint()` to refresh the window and have our change take effect. Use a call to `repaint()` whenever you want the graphics to update.

We now show the finished appearance of the two classes. First `ColorMenuItem.java` is shown here.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    private final Tricolor tc;
    public ColorMenuItem(Color _color, String _colorName, Tricolor _tc)
    {
        super(_colorName);
        color = _color;
        tc = _tc;
        addActionListener(new ColorMenuItemListener(tc, color));
    }
}
```

Now here is the listener class.

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

public class ColorMenuItemListener implements ActionListener
{
    private final Tricolor tc;
```

```
    private final Color color;
    public ColorMenuItemListener(Tricolor _tc, Color _color)
    {
        tc  = _tc;
        color = _color;
    }
    public void actionPerformed(ActionEvent e)
    {
        //Tell the current panel to change
        //color and refresh the whole thing.
        tc.getCurrentPanel().setColor(color);
        tc.repaint();
    }
}
```

Now run this. You will see that the Color menu now functions.

Programming Exercises

1. Add a new menu item for the color white. How much code do you have to add?
2. Add a new menu item called `tarheel` which uses color `0xabcdef`.

6.7 The Position Menu and Its Controller

We now go to work on the Position menu. Again we will be subclassing `JMenuItem`. We will make the menu item know two things: which panel belongs to it and we will make it know the application so it can communicate with it.

Let us sketch in `PositionMenuItem.java`.

```
import javax.swing.JMenuItem;

public class PositionMenuItem extends JMenuItem
{
    private final Tricolor tc;
    private final ColorPanel attachedPanel;
}
```

We now create a constructor. We will pass a string along to label the menu item called `pos`.

```
import javax.swing.JMenuItem;

public class PositionMenuItem extends JMenuItem
{
    public final Tricolor tc;
    public final ColorPanel attachedPanel;
    public PositionMenuItem(Tricolor _tc, ColorPanel _attachedPanel,
       String pos)
    {
        super(pos);
        tc = _tc;
        attachedPanel = _attachedPanel;
    }
}
```

Now we create the listener class. This will need to know the application and the attached panel. We comment in a procedure for the action listener to carry out.

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

public class PositionMenuItemListener implements ActionListener
{
    private final Tricolor tc;
    private final ColorPanel attachedPanel;
    public PositionMenuItemListener(Tricolor _tc, ColorPanel _attachedPanel)
    {
        tc = _tc;
        attachedPanel = _attachedPanel;
    }
    public void actionPerformed(ActionEvent e)
    {
        //set the current panel to the selected value
    }
}
```

We need the ability to set the current panel. To to this add this accessor method to `Tricolor`

```
public void setCurrentPanel(ColorPanel c)
{
    currentPanel = c;
}
```

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

public class PositionMenuItemListener implements ActionListener
{
    private final Tricolor tc;
    private final ColorPanel attachedPanel;
    public PositionMenuItemListener(Tricolor _tc, ColorPanel _attachedPanel)
    {
        tc = _tc;
        attachedPanel = _attachedPanel;
    }
    public void actionPerformed(ActionEvent e)
    {
        //set the current panel to the selected value
        tc.setCurrentPanel(attachedPanel);
    }
}
```

We are now ready to make the menu items. Notice that we do not repaint because a change of panel does not cause any graphics to need updating. The panel to be colored changed, and it will change the next time we select a `ColorMenuItem`.

Now let us add our items to the position menu.

```
positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
```

Now everything is going to work. Below, we see all of the classes in their entirety.

6.8 All Code Shown

This shows all of the classes.

6.8.1 Tricolor.java

This is the main application class.

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

```
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

public class Tricolor extends JFrame implements Runnable
{
    ColorPanel leftPanel;
    ColorPanel rightPanel;
    ColorPanel middlePanel;
    ColorPanel currentPanel;

    public Tricolor()
    {
        super("Tricolor");
        leftPanel = new ColorPanel();
        middlePanel = new ColorPanel();
        rightPanel = new ColorPanel();
        currentPanel = leftPanel;
    }
    public ColorPanel getCurrentPanel()
    {
        return currentPanel;
    }
    public void setCurrentPanel(ColorPanel c)
    {
        currentPanel = c;
    }
    public void run()
    {
        setSize(500,500);
        makeMenus();
        //install Color Panels
        Container c = getContentPane();
        c.setLayout(new GridLayout(1,3));
        c.add(leftPanel);
        c.add(middlePanel);
        c.add(rightPanel);
        setVisible(true);
    }
    private void makeMenus()
    {
        //make and install menu bar
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        //File menu
```

```
        JMenu fileMenu = new JMenu("File");
        mbar.add(fileMenu);
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(quitItem);
        quitItem.addActionListener(new QuitListener());
        // Color Menu
        JMenu colorMenu = new JMenu("Color");
        mbar.add(colorMenu);
        colorMenu.add(new ColorMenuItem(Color.RED, "red", this));
        colorMenu.add(new ColorMenuItem(Color.GREEN, "green", this));
        colorMenu.add(new ColorMenuItem(Color.BLUE, "blue", this));
        JMenu positionMenu = new JMenu("Position");
        // Position Menu
        mbar.add(positionMenu);
        positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
        positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
        positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
    }
    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

6.8.2 ColorPanel.java

This is the class for the three panels that fill the content pane.

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

public class ColorPanel extends JPanel
{
    Color color;
    public ColorPanel()
    {
        super();
        color = Color.WHITE;
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(color);
```

```
        g.fillRect(0,0,getWidth(), getHeight());
    }
    public void setColor(Color c)
    {
        color = c;
    }
}
```

6.8.3 QuitListener.java

This is the part of the controller that quits when the quit item is selected from the File menu.

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

public class QuitListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

6.8.4 ColorMenuItem.java

This is the class for the menu items in the Color menu.

```
import javax.swing.JMenuItem;
import java.awt.Color;


public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    private final Tricolor tc;
    public ColorMenuItem(Color _color, String _colorName, Tricolor _tc)
    {
        super(_colorName);
        color = _color;
        tc = _tc;
        addActionListener(new ColorMenuItemListener(tc, color));
    }
}
```

6.8.5 ColorMenuItemListener.java

This is the part of the controller that handles menu selections from the Color menu.

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

public class ColorMenuItemListener implements ActionListener
{
    private final Tricolor tc;
    private final Color color;
    public ColorMenuItemListener(Tricolor _tc, Color _color)
    {
        tc  = _tc;
        color = _color;
    }
    public void actionPerformed(ActionEvent e)
    {
        //tell the current panel to change color
        //System.out.println("fooment");
        tc.getCurrentPanel().setColor(color);
        tc.repaint();
    }
}
```

6.8.6 PositionMenuItem.java

This is the class for menu items determining which panel is to be colored.

```
import javax.swing.JMenuItem;

public class PositionMenuItem extends JMenuItem
{
    public final Tricolor tc;
    public final ColorPanel attachedPanel;
    public PositionMenuItem(Tricolor _tc, ColorPanel _attachedPanel,
        String pos)
    {
        super(pos);
        tc = _tc;
        attachedPanel = _attachedPanel;
        addActionListener(new PositionMenuItemListener(tc, attachedPanel));
    }
```

```
}
```

6.8.7 PositionMenuItemListener.java

This is the controller for the position menu items.

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

public class PositionMenuItemListener implements ActionListener
{
    private final Tricolor tc;
    private final ColorPanel attachedPanel;
    public PositionMenuItemListener(Tricolor _tc, ColorPanel _attachedPanel)
    {
        tc = _tc;
        attachedPanel = _attachedPanel;
    }
    public void actionPerformed(ActionEvent e)
    {
        //set the current panel to the selected value
        tc.setCurrentPanel(attachedPanel);
    }
}
```

Programming Exercises

1. Add a fourth panel to your app. How much code do you need to do this?

Chapter 7

Inner Classes, Anonymous Classes and Java GUIs

7.0 What is ahead?

Java allows you to create classes inside of other classes and, even inside of methods. Creating classes in these places gives us access to the outer class's state variables. We will find that this technique is very useful for GUI programming and for creating our own *data structures*. While we are in this chapter, we learn about Java's graphics libraries and we will make our programs responsive to mouse and keyboard activity.

7.1 Improving Tricolor

We are going to study the `Tricolor` application and improve it via the use of inner classes.

Let us begin with the quit menu item. We have a whole external class in our project that is used in one place: on the quit menu item. Can we get rid of this complexity and place the code necessary to drive the menu item inside of `Tricolor.java`?

The answer is, "yes." What we will do is to create an *anonymous inner class*. This is a class with no name. Go into the `makeMenus` method of `Tricolor.java` and find the lines on which the quit item is created and its action listener attached.

```
JMenuItem quitItem = new JMenuItem("Quit");
```

```
    fileMenu.add(quitItem);
    quitItem.addActionListener(new QuitListener());
```

We will now obviate the need for the external `QuitListener` class. Change the code as follows.

```
    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
    quitItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    });
```

Then by add these two import statements to the beginning of the `Tricolor.java`.

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

Now close the `QuitListener` class, compile, and run. You will see that the quit menu item is still working.

7.2 Deconstructing this Arabesque

You can see we have added some very mysterious code to `quitItem`. What does it all mean. Notice what is inside of the parentheses.

```
                              new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    }
```

You see a call to `new`, so you know an object is being created. You also know that `ActionListener` is an interface, so we can never create an instance of `ActionListener`.

So, what is happening? This is an example of an *anonymous inner class*. It is a class with no name. What you are saying here is, "make an instance of an `ActionListener` with this `actionPerformed` method." Since we never refer to it after we add it to the `quitItem` menu item object, we never do need to name it.

Anonymous inner classes provide a quick and easy way to attach actions to menu item or buttons. This is especially true if the menu item performs an isolated function, such as shutting an application down. The entire anonymous inner class is an argument you are sending to the call `quitItem.addActionListener()`!

We have now obviated the need for the `QuitListener` class, which just adds another name for a single-use object.

Take note of the way we formatted the code.

```
quitItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
```

Observe that the entire class declaration

```
new ActionListener()
{
   public void actionPerformed(ActionEvent e)
   {
        currentColor = Color.red;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
}
```

is part of the argument in the call `quitItem.addActionListener(....);`. This explains the seemingly strange construct `\tt });`. In one shot we have implemented an object's class and instantiated it as well.

We will refer to this funny closing object,`\tt });`, as "sad Santa." If you format this way, you should see Sad Santa as the last line of an anonymous listener class. It is very important to be fanatically consistent in this matter. It helps you avoid mysterious error messages that will vex and confuse.

7.3 Hammertime

Just because you have a shiny new hammer the entire world *does not* become a nail. You might be tempted to do this. And if you succomb to this impulse, it is going to work. You should go ahead and try it!

```
red = new ColorMenuItem(Color.RED, "red", this);
```

```
red.addActionListener(new ActionListener()
   {
       public void actionPerformed(ActionEvent e)
       {
            currentColor = Color.red;
            currentPanel.setColor(currentColor);
            currentPanel.repaint();
        }
   });
```

You can now repeat this procedure for the green and blue menus as follows.

```
green = new ColorMenuItem(Color.GREEN, "green", this);
green.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        currentColor = Color.green;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
});
blue = new ColorMenuItem(Color.BLUE, "blue", this);
blue.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        currentColor = Color.blue;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
});
```

But you can see this is the sort of violation of the 11th commandment that you were earlier warned of.

7.4 Using Inner Classes to Improve our Design

The implementaton we just showed works, but it exhibits duplicate and slack code in the implementation of the action listeners. We begin to think: *Can we attach the action listener directly to the color menu item?*. Let us return to our original design for the `ColorMenuitem` We can convert it to being an inner class to `Tricolor` and obviate the need for the `Tricolor tc` state variable. So we begin with this.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    private final Tricolor tc;
    public ColorMenuItem(Color _color, String _colorName, Tricolor _tc)
    {
        super(_colorName);
        color = _color;
        tc = _tc;
        addActionListener(new ColorMenuItemListener(tc, color));
    }
}
```

Now we trim this down to be an inner class. Both `JMenuItem` and `Color` are imported so we can lop the imports off.

Since we are going to make this into an inner class of `Tricolor`, we will have access to `Tricolor`'s state variables. As a result, we will get rid of the instance of `Tricolor` that is a state variable inside of the class. We will also get rid of the call to `ColorMenuItemListener`. This will be replaced by an inner class.

```
class ColorMenuItem extends JMenuItem
{
    private final Color color;
    public ColorMenuItem(Color _color, String _colorName)
    {
        super(_colorName);
        color = _color;
        addActionListener(); TODO:  Write action listener!
    }
}
```

Now let us pop this inside of `Tricolor`. We will need to do some adjustment before this will compile. First, chage the lines in `makeMenus`

```
colorMenu.add(new ColorMenuItem(Color.RED, "red", this));
colorMenu.add(new ColorMenuItem(Color.GREEN, "green", this));
colorMenu.add(new ColorMenuItem(Color.BLUE, "blue", this));
```

to

```
colorMenu.add(new ColorMenuItem(Color.RED, "red"));
```

```
colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
```

Also add these two imports.

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

Our class now looks like this. You can close the `QuitListener` and `ColorMenuItem` classes. These are now obviated.

```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


public class Tricolor extends JFrame implements Runnable
{
    ColorPanel leftPanel;
    ColorPanel rightPanel;
    ColorPanel middlePanel;
    ColorPanel currentPanel;

    public Tricolor()
    {
        super("Tricolor");
        leftPanel = new ColorPanel();
        middlePanel = new ColorPanel();
        rightPanel = new ColorPanel();
        currentPanel = leftPanel;
    }
    public ColorPanel getCurrentPanel()
    {
        return currentPanel;
    }
    public void setCurrentPanel(ColorPanel c)
    {
        currentPanel = c;
    }
    public void run()
```

```
{
    setSize(500,500);
    makeMenus();
    //install Color Panels
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,3));
    c.add(leftPanel);
    c.add(middlePanel);
    c.add(rightPanel);
    setVisible(true);
}
private void makeMenus()
{
    //make and install menu bar
    JMenuBar mbar = new JMenuBar();
    setJMenuBar(mbar);
    //File menu
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
    quitItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    });
    // Color Menu
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new ColorMenuItem(Color.RED, "red"));
    colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
    colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
    JMenu positionMenu = new JMenu("Position");
    // Position Menu
    mbar.add(positionMenu);
    positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
    positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
    positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
}
class ColorMenuItem extends JMenuItem
{
    private final Color color;
    public ColorMenuItem(Color _color, String _colorName)
    {
        super(_colorName);
```

```
            color = _color;
            //addActionListener(); //TODO:  Write action listener!
        }
    }
    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

Now let us get the action listener working. We will attach this as an anonymous inner class inside of `ColorMenuItem`. Begin by creating a shell for it.

```
public ColorMenuItem(Color _color, String _colorName)
{
    super(_colorName);
    color = _color;
    addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
        }
    });
}
```

Before doing anything else, compile and make sure you have all of your formatting ducks in a row. Now we write the body. What do we want to happen? The current panel should be set to this menu item's color. We should then update the graphics. You now insert this code.

```
public ColorMenuItem(Color _color, String _colorName)
{
    super(_colorName);
    color = _color;
    addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            currentPanel.setColor(color);
            repaint();
        }
    });
}
```

Now you run this shiny new code and uh oh... we appear to have a repaint error. What happened? Only the menu item is repainting itself! There are two ways to handle this. One is to tell the current panel to repaint. Change

```
repaint();
```

to

```
currentPanel.repaint();
```

Another way is to tell the whole app to repaint. You do have access to the `this` of an enclosing class. In this case just use

```
Tricolor.this.repaint();
```

We use the former solution, since it does the minimum work needed and accomplishes the goal. The color menu is now completely operational as it was before. We have relieved our code of some complexity. Since the state variables of the enclosing class are visible we can shorten our constructor and the complexity of the code inside of the `ColorMenuItem`. We make the listener a totally anonymous class, and cut complexity there too. You no longer need the two external classes that controlled the Color menu.

7.5 The Position Menu

Now we are ready to adjust the position menu. Here is the current state of `PositionMenuItem.java`.

```
import javax.swing.JMenuItem;
public class PositionMenuItem extends JMenuItem
{
    public final Tricolor tc;
    public final ColorPanel attachedPanel;
    public PositionMenuItem(Tricolor _tc, ColorPanel _attachedPanel,
        String pos)
    {
        super(pos);
        tc = _tc;
        attachedPanel = _attachedPanel;
        addActionListener(new PositionMenuItemListener(tc, attachedPanel));
    }
}
```

We now slim this down to be an inner class.

```
public class PositionMenuItem extends JMenuItem
{
```

```
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        //TODO: write new listener
        //addActionListener(new PositionMenuItemListener(tc, attachedPanel));
    }
}
```

Now go into `makeMenus` and change

```
positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
```

to

```
positionMenu.add(new PositionMenuItem(leftPanel, "left"));
positionMenu.add(new PositionMenuItem(middlePanel, "middle"));
positionMenu.add(new PositionMenuItem(rightPanel, "right"));
```

Once you do this, the program will compile. You no longer need either external class controlling the position menu. Close them.

Next we write the listener as an anonymous inner class. Begin by making the shell.

```
class PositionMenuItem extends JMenuItem
{
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
            }
        });
    }
}
```

What needs to happen when a new position is selected? We just change the current panel. No graphical change is needed. So let's add the code.

```
class PositionMenuItem extends JMenuItem
{
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                currentPanel = attachedPanel;
            }
        });
    }
}
```

7.6 Cruft Patrol!

It seems some getter and setter methods in `Tricolor` are obviated. What can we trim out? Comment these out.

```
public ColorPanel getCurrentPanel()
{
    return currentPanel;
}
public void setCurrentPanel(ColorPanel c)
{
    currentPanel = c;
}
```

Everything still works, so you can get rid of them.

7.7 The Product

We are left with two classes. Here is `ColorPanel.java`, which we never changed

```
import javax.swing.JPanel;
import java.awt.Color;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
```

```
    {
        color = Color.white;
    }
    public void setColor(Color _color)
    {
        color = _color;
    }
    public Color getColor()
    {
        return color;
    }
}

import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


public class Tricolor extends JFrame implements Runnable
{
    ColorPanel leftPanel;
    ColorPanel rightPanel;
    ColorPanel middlePanel;
    ColorPanel currentPanel;

    public Tricolor()
    {
        super("Tricolor");
        leftPanel = new ColorPanel();
        middlePanel = new ColorPanel();
        rightPanel = new ColorPanel();
        currentPanel = leftPanel;
    }
    public void run()
    {
        setSize(500,500);
        makeMenus();
        //install Color Panels
        Container c = getContentPane();
        c.setLayout(new GridLayout(1,3));
```

```
        c.add(leftPanel);
        c.add(middlePanel);
        c.add(rightPanel);
        setVisible(true);
    }
    private void makeMenus()
    {
        //make and install menu bar
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        //File menu
        JMenu fileMenu = new JMenu("File");
        mbar.add(fileMenu);
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(quitItem);
        quitItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                System.exit(0);
            }
        });
        // Color Menu
        JMenu colorMenu = new JMenu("Color");
        mbar.add(colorMenu);
        colorMenu.add(new ColorMenuItem(Color.RED, "red"));
        colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
        colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
        JMenu positionMenu = new JMenu("Position");
        // Position Menu
        mbar.add(positionMenu);
        positionMenu.add(new PositionMenuItem(leftPanel, "left"));
        positionMenu.add(new PositionMenuItem(middlePanel, "middle"));
        positionMenu.add(new PositionMenuItem(rightPanel, "right"));
    }
    class ColorMenuItem extends JMenuItem
    {
        private final Color color;
        public ColorMenuItem(Color _color, String _colorName)
        {
            super(_colorName);
            color = _color;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    currentPanel.setColor(color);
                    currentPanel.repaint();
```

```
                }
            });
        }
    }

    class PositionMenuItem extends JMenuItem
    {
        public final ColorPanel attachedPanel;
        public PositionMenuItem(ColorPanel _attachedPanel, String pos)
        {
            super(pos);
            attachedPanel = _attachedPanel;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    currentPanel = attachedPanel;
                }
            });
        }
    }

    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

Programming Exercises

1. Add a new item to the color menu that sets the current panel to white.
2. Add a new panel and modify the position menu to accomodate it.

7.8 Inner Classes in General

We have inner classes in several guises during the study of the tricolor application. Let us now summarize and consolidate what we have seen.

Inner classes can be static. Such classes have access to private methods and varibles of the enclosing class, but only via an instance of that class. In the code, shown here, note the illegal and legal examples.

```
public class Outer
```

```
{
    int x;
    int y;
    public Outer(... )
    {
        //constructor code
    }
    private foo(...)
    {
        //code
    }
    static class Inner
    {
        public void go()
        {
            x = 5  //Illegal
            Outer o  new Outer(...);
            o.x = 5 //OK
            foo(....);  // Illegal
            o.foo(...)  //OK
        }
    }
}
```

Remember, non-static portions of a class have access to static portions, but direct access (not via an instance) is not allowed. If you are declaring an inner class `static` you should closely consider making it a separate class, unless it relieves a lot of complexity.

This is in contrast to the non-static inner classes we used to control the menus. These were created so as to have access to the state variables of the enclosing outer class. We created the named inner classes `PositionMenuItem` and `ColorMenuItem`.

The anonymous class we created as an event handler for `quitItem` is an example of a *local class*, since it is created inside of a method of the enclosing class. Local classes do not have access to the local variables of their enclosing method, unless the variable is declared `final`. Accessing a local nonfinal variable is a compiler error.

We will return to this topic when we discuss Java collections; inner classes can play an important role in the creation of data structures.

7.9 Adding and Deleting Components from a `JFrame`

We will show a program that adds and removes buttons from a window "on the fly." Begin by creating a graphical shell.

```
import javax.swing.JFrame;

public class Adder extends JFrame implements Runnable
{
    public Adder()
    {
        super("Add and Remove Buttons Demo");
    }
    public void run()
    {
        setSize(600,600);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Adder a = new Adder();
        javax.swing.SwingUtilities.invokeLater(a);
    }
}
```

Run and compile this; the result is an empty window with a title in the title bar specified by the constructor.

Next, on the top of the window, we will add buttons named `"Add"` and `"Remove"`. To do this we will take take the following steps.

1. Make a JPanel with a 1 row 2 column grid layout.
2. Add the two buttons to it
3. Add it to the north side of the content pane using the `BorderLayout` static constant `NORTH`.

We modify the `run()` method as follows.

```
public void run()
{
    setSize(600,600);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
    JPanel topPanel = new JPanel(new GridLayout(1,2));
    JButton addButton = new JButton("Add");
    topPanel.add(addButton);
    JButton removeButton = new JButton("Remove");
    topPanel.add(removeButton);
    getContentPane().add(BorderLayout.NORTH, topPanel);
    setVisible(true);
}
```

Do not neglect to add these imports.

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

You will see that the add and remove buttons now appear at the top of the content pane.

Next, we shall make an inner class that extends `JPanel` and which overrides its `paintCompnent` method. We will put at 10×10 grid layout in this panel.

Here it is. Do not forget to import `java.awt.Graphics`.

```
class ButtonPanel extends JPanel
{
    public ButtonPanel()
    {
        super(new GridLayout(10,10));
    }
    @Override
    public void paintComponent(Graphics g)
    {
    }
}
```

This is the panel in which we will keep our buttons. Let's make an array list of buttons and make it a state variable. We will also make the button panel a state variable and add it to the window. We will initialize it in the constructor. Add this to your the top of the class

```
private ArrayList<JButton> buttons;
private ButtonPanel bp;
```

Inside the constructor add

```
buttons = new ArrayList<JButton>();
bp = new ButtonPanel();
```

To your imports add

```
import java.util.ArrayList;
```

Now let us add code to cause pushing the Add button to addd buttons to the array list `buttons`. We do this by creating an action listener and attaching it to `addButton` in the `run` method as follows. Note the use of the if statement to prevent overpopulation.

```
addButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        int n = buttons.size();
        if(n < 100)
        {
             buttons.add(new JButton("" + buttons.size()));
             repaint();
        }
    }
});
```

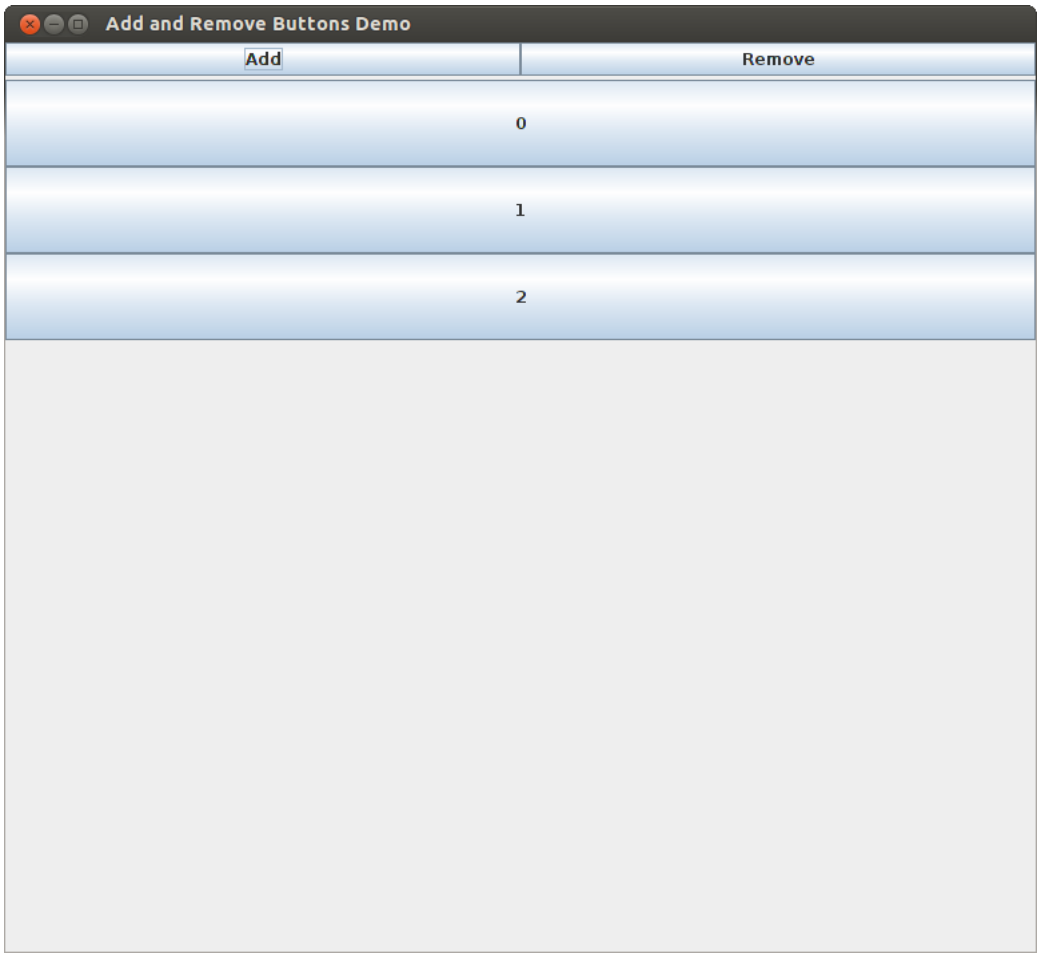
If you hit the add button, you will not see any buttons added to the screen. Stick in this line of code to see the listener is working.

```
System.out.printf("buttons.size() = %s\n", buttons.size());
```

Hit the add button and see this in the console.

```
buttons.size() = 1
buttons.size() = 2
buttons.size() = 3
buttons.size() = 4
buttons.size() = 5
buttons.size() = 6
```

The array list is being populated. However, we are not seeing the buttons appear on in the lower part of the console. Let us get them in using the `repaint()` method. Before proceeding, delete the print line you just inserted. Now we are going to test this out. Run the app and hit the add button three times. You will see nothing has happened. Now maximize or resize the window. That triggers a repaint. You will see this.



Now hit the add button five more times. You will see no changes. Then trigger a repaint by resizing or maximizing. You will then see what you expect, which is this.

The window does not behave as expected. You have to trigger a repaint to see the added buttons. We don't want this. Notice that containers are smart, if you add the same widget several times, it will only appear once.

Next, let us try to remove the last button we put in. Add this listener. Note the use of the if statement to prevent a "blood from a turnip" situation that could cause a nasty exception.

```
removeButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        int n = buttons.size();
        if(n > 0)
        {
            buttons.remove(n - 1);
```

```
            repaint();
        }
    }
});
```

Now run the app and click the add button five times. Resize or maximize the window. The five buttons then suddenly appear. Now click the remove button once and resize. You will see this.

It is whacko and it is not what we expect. What is the remedy for resetting the components in a `JPanel`? Modify your `paintComponent` as follows.

```
@Override
public void paintComponent(Graphics g)
{
    removeAll();
    for(JButton b: buttons)
    {
        add(b);
```

```
    }
    revalidate();
}
```

This evicts all items from the window, adds back in the ones that belong, and then `revalidate()` ensures that the compoents themselves refresh. This goes deeper than just repainting. Your add and remove features now work as expected. Now keep clicking until the whole window fills up with buttons.

Chapter 8

Exception Handling

8.0 Introduction

The next major component of the Java language will allow us to write programs that handle errors gracefully. Java provides a mechanism called *exception handling* that provides a parallel track of return from functions so that you can avoid cluttering the ordinary execution of code with endless error-handling routines.

Exceptions are objects that are "thrown" by varous methods or actions. In this chapter we will learn how to handle (catch) an exception. By so doing we allow our program to recover and continue to work. Failure to catch and exception results in a flood of nasty red text from Java (a socalled "exploding heart"). Crashes such as these shoud be extremely rare in productionquality software. We can use exceptions as a means to protect our program from such dangers as user abuse and from such misfortures as crashing whilst attempting to gain access to an nonexistent or prohibited resource. Many of these hazards are beyond both user and programmer control.

When you program with files or with socket connections, the handling of exceptions will be manditory; hence the need for this chapter before we begin handling files.

8.1 The `Throwable` Subtree

Go to the Java API guide and pull up the class `Exception`. The family tree is is as follows.

```
java.lang.Object
java.lang.Throwable
```

`java.lang.Exception`

The class name `Throwable` is a bit strange; one would initially think it were an interface. It is, however, a class. The class `java.lang.Exception` has a sibling class `java.lang.Error`.

When objects of type `Error` are thrown, it is not reasonable to try to recover. These things come from problems in the Java Virtual Machine, bad memory problems, or problems from the underlying OS. We just accept the fact that they cause program death. Continuing to proceed would just lead to a chain of ever-escalating problems.

Objects of type `Exception` are thrown for more minor problems, such as an attempt to open a non-existent file for reading, trying to convert an unparseable string to an integer, or trying to access an entry of a string, array or array list that is out of bounds.

Let us show this mechanism at work. For example, if you attempt to execute the code

```
int foo = Integer.parseInt("gobbledegook");
```

you will be rewarded with an exception. To see what happens, create this program `MakeException.java`.

```
public class MakeException
{
    public static void main(String[] args)
    {
        int foo = Integer.parseInt("gobbledegook");
    }
}
```

This program compiles happily. You will see that the infraction we have here is a run-time error, as is any exception.

When you run the program you will see this in the interactions pane.

```
java.lang.NumberFormatException: For input string: "gobbledegook"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at MakeException.main(MakeException.java:5)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
    at java.lang.reflect.Method.invoke(Method.java:601)
    at edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand(JavacCompiler.java:2
```

The exploding heart you see here shows a *stack trace*. This shows how the exception propagates through the various calls the program makes. To learn what you did wrong, you must look in this list for your file. You will see the offending line here.

```
at MakeException.main(MakeException.java:5)
```

You are being told that the scene of the crime is on line 5, smack in the middle of your `main` method. The stack trace can yield valuable clues in tracking down and extirpating a run-time problem such as this one.

We have seen reference to "throw" and "throws" before. Go into the API guide and bring up the class `String`. Now scroll down to the method summary and click on the link for the familiar method `charAt()`. You will see this notation.

Throws:

> `IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of the string.

Let us now look up this `IndexOutOfBoundsException`. The family tree reveals that this class extends the `RuntimeException` class. The purpose of this exception is to create an opportunity to gracefully get out of an exceptional situation and to avoid having your program simply crash. Incidentally, this an an opportunity we will not always avail ourselves of, especially when it arises because of a programmer error.

8.2 Checked and Run-Time Exceptions

There are two types of exceptions that exist: `RuntimeExceptions` and all others, which are called *checked exceptions*. Generally a run-time exception is caused by *programmer* error. Programmers should know better, and it is probably good for them to have their programs using your class die with an exploding heart as a reward for writing rotten code. Such programmers can read the stack trace and lick thier wounds. How do you know if an exception is a `RuntimeException`? Just look up its family tree and see if it is a descendant of `RuntimeException`. So far in our study of Java, we have only seen runtime exceptions.

Checked exceptions, on the other hand, are usually caused by situations beyond programmer control. Suppose a user tries to get a program to open a file that does not exist, or a file for which he lacks appropriate permissions. Another similar situation is that of attempting to create a *socket*, or a connection

to another computer. That computer may disallow such connections, it could be down, or it could even be nonexistent. These situatoins are not necessarily the user's or programmer's fault.

Checked exceptons must be *handled*; this process entails creating code to tell your program what to do in the face of these exceptions being thrown. It is entirely optional to handle a runtime exception.

Sometimes a runtime exception will be caused by user error; in these cases it is appropriate to use exception handling to fix the problem. For example if a user is supposed to enter a number into a `JOptionPane` dialog and enters a string that is not numeric, your program might try to use `Integer.parseInt` to convert it into an integer. Here we see a problem created by an end-user. This user should be protected and this error should be handled gracefully so that (bumbling) user can go about his business. You always want to protect the end-user from exceptions if it is at all feasible or reasonable.

8.2.1 Catching It

Java provides a parallel track of execution for handling exceptions gracefully. Suppose you are writing a program that displays a color in response to a hex code entered by a (very dumb) end-user of Your Shining Program. The user enters something like `ffgg00`; this is a situation that you, the programmer do not control. The Java exception mechanism would allow you to cleanly punt and reset your color calculator to some state, such as white, and display the appropriate hex code, `0xFFFFFF`.

Some resourceful hackish readers might think, "Here is a new and useful way to get unwedged from a bad situation." This is a mistake. Only use exception handling for error situations beyond programmer control. Do not use them for the ordinary execution of your programs.

8.3 A Simple Case Study

Let us write a simple color calculator. Our application is to have three graphical elements. It will have a color panel to display the color sample which will occupy most of the frame. On the top of the frame we will place a `JButton` and a `JTextField`. The user types into the `JTextField` and hits enter or hits the button and the color is shown.

We shall immediately bring on our existing `ColorPanel` class and recycle it shamelessly. This should give you the idea that you want to design plenty of reusable classes that can be helpful in a variety of situations.

```
import javax.swing.JPanel;
```

```
import java.awt.Color;
import java.awt.Graphics;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
    {
        super();
        color = Color.white;
    }
    public void setColor(Color _color)
    {
        color = _color;
    }
    public Color getColor()
    {
        return color;
    }
    public void paintComponent(Graphics g)
    {
        g.setColor(color);
        g.fillRect(0,0,getWidth(), getHeight());
    }
}
```

Now let us begin by building the frame. We block in the three graphical elements as state variables

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import java.awt.Color;
import java.awt.Container;
public class SimpleColorCalc extends JFrame implements Runnable
{
    final ColorPanel cp;
    final JButton show;
    final JTextField hexCode;

    public SimpleColorCalc()
    {
        super("Simple Color Calculator");
        cp = new ColorPanel();
    }
    public static void main(String[] args)
    {
```

```
        SimpleColorCalc scc = new SimpleColorCalc();
        javax.swing.SwingUtilities.invokeLater(scc);
    }
    public void run()
    {
        setSize(400,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container c = getContentPane();
        setVisible(true);
    }

}
```

Now we shall write the balance of the constructor During this process, we estbllish the basic properties of the graphical widgets. The color panel will be white. The text field will show the hex code for white, `0xffffff`. The `JButton` gets created and labeled. We also set fonts for the button and the text field to enhance the appearance of the app.

```
public SimpleColorCalc()
{
    super("Simple Color Calculator");
    cp = new ColorPanel();
    show = new JButton("Show the Color");
    hexCode = new JTextField("0xffffff");
    hexCode.setFont(new Font("Monospaced", Font.BOLD, 12));
    hexCode.setHorizontalAlignment(JTextField.RIGHT);
    show.setFont(new Font("Monospaced", Font.BOLD, 12));
}
```

Enter this code and compile. Add the necessary impoorts.

Next we add these lines to the `run()` method. Here we create a `JPanel` and pop the button and text field into it. We then add the panel to the top of the frame. We next place the color panel in the center, where it will occupy the biggest space, as we had planned.

```
JPanel top = new JPanel();
c.add(BorderLayout.NORTH, top);
top.setLayout(new GridLayout(1,2));
top.add(show);
top.add(hexCode);
c.add(BorderLayout.CENTER, cp);
```

Make sure you put the `setVisible(true)` call last. Run this code and you will see a button in the upper-left, a text field with `"ffffff"` emblazoned on it in the upper right, and a color panel filled with white.

The next logical step is to make the button and text fields live. When the button is pushed or enter is hit in the text field , we want the following to happen.

1. Get the text from the `JTextField`. It comes in as a string.
2. Turn it into a hex code.
3. Get the color for the hex code
4. Have the color panel paint itself that color.

To this end, we will create an action listener. Since the text field will issue an action event when enter is hit in it, we can use the same listener class for the text field and the button.

Begin with a shell we place inside of our class.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
    }
}
```

Go to the API guide and look up `JTextField`. Look in the constructor summary and see the constructor we used. Now do a search for `getText()`. This is a method inherited from the parent class `javax.swing.JTextComponent`. It returns the text, as a string, residing in the `JTextField`. Let us now test this using our action listener. Make sure you do the includes for the action listener and the action event.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        System.out.println(text);
    }
}
```

Compile and run and you should see that when you click on the button, the text in the `JTextField` is put to `stdout`. Thusfar, we have waded in and learned about a new widget. How about an exception here?

What is beyond programmer control? The user just might do something stupid like enter `cowpie` in the `JTextField`. As it stands now, our program will happily put that to stdout. However, we want to convert this value to a hex

code. To perform the conversion, we use the `Integer` wrapper class. Bring it up in your API guide. Now find the method `parseInt(String s, int radix)` and click on its link. The word *radix* is just another word for number base. Since we are trafficking in hex codes here, we shall use base 16. You can also see that it throws a `NumberFormatException`. Go to the API page for this exception; you can do so by clicking on the link shown.

This is a runtime exception, as you can see by looking up the family tree. However, it is triggered by an end-user's blunder, so we shall deal with it gracefully. We will just reset our color calculator to its original white.

The tasks confronting us here are: get the hex code, convert it into a hex number, but if an illegal value is entered, reset everything to white. First here is the naked call to the static method `Integer.parseInt()`. We know this method is dangerous: it throws an exception.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        int colorcode = Integer.parseInt(text, 16);
    }
}
```

Imagine you are about to get into a tub full of water. Do you just plunge in? This is not the recommended course of action if you desire continued and comfortable existence. You first dip a finger or toe in the water. If it's too hot or too cold, you throw a `BadTubwaterTemperatureException`. You can recover from such an error. If the tub is too cold, add hot water until the desired temperature is reached. If the tub is too hot, let it cool or add cold water until the water is at a suitable temparature.

Java provides a mechanism called the the `try-catch` sequence to handle the exception. We now insert this and explain it.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorcode = Integer.parseInt(text, 16);
        }
        catch(NumberFormatException ex)
        {
```

```
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
            repaint();
        }
    }
}
```

You place the "dangerous code" you are using inside of the `try` block. In this case, the danger is generated by a number format exception triggered by the abuse of our innocent program. If the user enters a legal hex code, the `catch` block is ignored; if not, the `catch` block executes. This precludes the occurrence of an exploding heart caused by an end–user abusing `Integer.parseInt()`. Once we have circumnavigated the danger, we can go about our business of obtaining a color and coloring the panel. Notice that all of the code using the integer variable `colorCode` is placed in the `try` block. This is because an exception will cause the `try` block to abort immediately, and the variable you wish to use will never be created during an error state.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorCode = Integer.parseInt(text, 16);
            Color enteredColor = new Color(colorCode);
            cp.setColor(enteredColor);
            repaint();
        }
        catch(NumberFormatException ex)
        {
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
            repaint();
        }
    }
}
```

Pssssst.... Confidentially..... Duplicate code! We see that `repaint()` should occur regardless of whether an exception is thrown or not. The `try`–`catch` apparatus has one more item, the `finally` block. This block is carried out whether an exception occurs or not. Here we see it at work.

```
class ColorChangeListener implements ActionListener
```

```
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorCode = Integer.parseInt(text, 16);
            Color enteredColor = new Color(colorCode);
            cp.setColor(enteredColor);
        }
        catch(NumberFormatException ex)
        {
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
        }
        finally
        {
            repaint();
        }
    }
}
```

Now finish by attaching this listener to the button and text field. Go into the run method and add these lines just before the `setVisible` line.

```
show.addActionListener(new ColorChangeListener());
hexCode.addActionListener(new ColorChangeListener());
```

This is a ready–for–prime–time program that functions robustly. Exception handling gives it fault-tolerance that makes it deployable in a realistic situation.

8.4 All Code Shown

Here is the complete program `SimpleColorCalc.java`. Make sure you have the `ColorPanel` in the same directory when compiling.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.awt.Color;
import java.awt.Font;
import java.awt.Container;
```

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

public class SimpleColorCalc extends JFrame implements Runnable
{
    final ColorPanel cp;
    final JButton show;
    final JTextField hexCode;
    public SimpleColorCalc()
    {
        super("Simple Color Calculator");
        cp = new ColorPanel();
        show = new JButton("Show the Color");
        hexCode = new JTextField("0xffffff");
        hexCode.setFont(new Font("Monospaced", Font.BOLD, 12));
        hexCode.setHorizontalAlignment(JTextField.RIGHT);
        show.setFont(new Font("Monospaced", Font.BOLD, 12));
    }
    public static void main(String[] args)
    {
        SimpleColorCalc scc = new SimpleColorCalc();
        javax.swing.SwingUtilities.invokeLater(scc);
    }
    private static String presentHexCode(String s)
    {
        s = s.toLowerCase();
        if(! s.substring(0,2).equals("0x"))
            s = "0x" + s;
        return s;
    }
    public void run()
    {
        setSize(400,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container c = getContentPane();
        //Make a space for the text field and button.
        JPanel top = new JPanel();
        top.setLayout(new GridLayout(1,2));
        top.add(show);
        top.add(hexCode);
        //add the top and the color panel to the content pane
        c.add(BorderLayout.NORTH, top);
        c.add(BorderLayout.CENTER, cp);
        show.addActionListener(new ColorChangeListener());
```

```
        hexCode.addActionListener(new ColorChangeListener());
        setVisible(true);
    }
    class ColorChangeListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            String text = hexCode.getText();
            try
            {
                int colorCode = Integer.parseInt(text, 16);
                Color enteredColor = new Color(colorCode);
                hexCode.setText(presentHexCode(text));
                cp.setColor(enteredColor);
            }
            catch(NumberFormatException ex)
            {
                hexCode.setText("0xffffff");
                cp.setColor(Color.white);
            }
            finally
            {
                repaint();
            }
        }
    }
}
```

8.5 Exception Handling, In General

Now that we have seen a simple case study, we can begin to understand the whole business of handling exceptions. You will have already noticed some things.

Exceptions are objects; they are instances of classes. The class `Throwable` is the root class. The class `Throwable` has two children, `Exception` and `Error`. As we said before, we shall concentrate on exceptions here, so you may actually treat `Exception` as the root class.

8.5.1 Can you have several `catch` blocks?

The answer to this is yes. A `try` block must be followed immediately by at least one `catch` block. A `finally` block is an optional block that occurs after a `try` block and one or more `catch` blocks; this block executes whether or not

an exception is thrown. Do not place other code between the succeeding `try`, `catch` and `finally` blocks. Each `catch` block can catch a different type of exception.

The general syntax looks like this.

```
try
   {
   //some dangerous code that throws various exceptions
   }
catch(FairlySpecificException e)
   {
   //code to handle a fairly specific exception.
   }
catch(LessSpecificExceptoin e)
   {
   //code to handle a less specific exception
   }
.
.
.
catch(LeastSpecificException e)
   {
   //code to handle the most general possible exception
   }
finally
   {
   //do this no matter what
   }
```

8.5.2 The Bucket Principle

It is possible that code in a `try` block might throw several types of exceptions. Think of the catch blocks as being buckets and the exceptions as "dropping out" from your code. The most general type of exception is `Exception`; if you do the following, you will be in for a surprise.

```
try
   {
   //code that throws a NumberFormatException
   //and some other exceptions
   }
catch(Exception e)
   {
   //handle an exception
```

```
   }
catch(NumberFormatExcepton e)
   {
   //handle a number format exception
   }
```

The second catch block is dead code! Think of the catch blocks as being buckets for catching exceptions. A catch block with a more general type of exception is a bigger bucket. Once a bucket catches the exception, it is handled and execution skips to the `finally` block if one exists. If not, you continue execution at the end of the `try`–`catch` progression. Were a number format exception to be thrown in our code here, the big bucket at the top, the `Exception` bucket, would catch any number format exception. The moral of this tale is: Place more specialized exceptions in earlier catch blocks and the more general ones at the bottom. If two types of exception are independent, they represent independent buckets. An example of such exceptions is `NumberFormatException` and `FileNotFoundException`. Catch blocks for these would represent independent, nonoverlapping buckets. Look in the API guide to see that they are unrelated, and their lowest common ancestor is the root exception class `Exception`.

Reasoning by analogy is dangerous, but we plunge ahead insouciantly nonetheless. The behavior if `try`–`catch` progression is much like an `if--else if--else` progression in that the first active block executes. The analogy, however, is incomplete because the `finally` block executes in any event, unlike the `else` block.

Arrange your buckets so you get the desired action in recovering from your error. Also engineer your buckets to make them leak–proof: remember, an uncaught exception will crash your program. You should aim to catch as narrowly as possible.

Check the methods you are using in the `try` block for the types of exceptions they can throw. Then aim to catch just these.

8.6 Mr. Truman, We Must Pass the Buck!

Go into the API guide to `Integer` and get the method detail for `parseInt()`. Here is its method header.

```
public static int parseInt(String s) throws NumberFormatException
```

You see a new keyword, `throws`. This declaration in the function header is a tocsin to the client programmer: *Beware, this method can generate, or throw, a* `NumberFormatException`. The creator of this method is "passing the buck" and forcing the client to handle this exception. The penalty for failing to do so is the possibility of an uncaught exception and ugly program death.

8.6.1 Must I?

We have said that, when an exception is caused by programmer error, you probably should not catch it, unless there is a compelling reason of cost or operational practicality. Handling runtime exceptions is *optional*. Handling checked exceptions is mandatory, unless you pass the buck by using the `throws` keyword in the method header. In this case, you force the caller to handle the exception.

Let us look at such an example. Go into the API guide and bring up the `FileReader` class. The method detail for the first constructor reads as follows.

```
public FileReader(String fileName) throws FileNotFoundException
```

Click on the link for `FileNotFoundException`. Since `RuntimeException` is not in the family tree of `FileNotFoundException`, we see that `FileNotFoundException` is a checked exception. This would be appropriate since there is no programmer control over the file system where the program is being used. You will see plenty of examples of checked exception in the next chapter on fileIO.

Reading the preamble, we see this exception is thrown if you attempt to open a nonexistent file for reading or you try to open a file for which you do not have read permission. If you are going to instantiate a `FileReader` in a class method, you have two choices. You can handle the exception in the method, or you can add `throws FileNotFoundException` to the method header. Ultimately, any client code invoking this method must either handle the exception or pass the buck.

Here is how we handle it.

```
public void processFile(String filename)
{
    try
    {
        FileReader f = new FileReader(fileName);
        //code that processes the file
    }
    catch(FileNotFoundException e)
    {
        //code to bail out of the wild goose chase
    }
}
```

Here is how to pass the buck.

```
public void processFile(String filename) throws FileNotFoundException
{
```

```
    FileReader f = new FileReader(fileName);
    //code that processes the file
}
```

Passing the buck forces the caller to handle the exception. You should also note that you must add the import statement

```
import java.io.FileNotFoundException
```

to use a `FileNotFoundException` because it is not part of the `java.lang` pacakage.

8.7 Can I Throw an Exception?

In short, the answer is "yes." Let us show an example. Suppose you are writing a fraction class.

```
public class Fraction
{
    int num;
    int denom;
    public Fraction(int _num, int _denom)
    {
        num = _num;
        denom = _denom;
        //more code
    }
    public Fraction()
    {
        this(0,1);
    }
    //more code
}
```

Your program should get annoyed if some foolish client does something like this.

```
Fraction f = new Fraction(3,0);
```

You do not want people abusing your code by programming with zero-denonomiator functions. Modify your code as follows to punish the offender. We shall throw an `IllegalArgumentException`.

```
public class Fraction
{
    int num;
    int denom;
    public Fraction(int _num, int _denom)
        throws IllegalArgumentException
    {
        if(_denom == 0)
             throw new IllegalArgumentException("Zero Denominator");
        num = _num;
        denom = _denom;
        //more code
    }
    public Fraction()
    {
        this(0,1);
    }
```

The `IllegalArgumentException` is a runtime exception (check its family tree), so it is optional for the caller to check it. You can leave the issue to the caller's judgement. As soon as the exception is triggered, the execution of the constructor is called to a halt. The exception is thrown to the caller; if the caller does not handle it, the caller is rewarded with an exploding heart.

8.7.1 Can I make my own exceptions?

Yes, all you need do is extend an existing exception class. For example we might want to have a special exception for our `Fraction` class. We create it as follows.

```
public class ZeroDenominatorException extends RuntimeException
{
}
```

We extended the `RuntimeException`, so our exception is a runtime exception. You may now throw `ZeroDenominatorExceptions`. You could choose to extend `IllegalArgumentException`, since this exception results from passing illegal arguments to a constructor.

However, you should strive to use standard exceptions wherever possible. The use of the `IllegalArgumentException` in this constructor was probably the best call.

8.8 Summary

Java provides an exception handling apparatus that allows you to handle various common error states without cluttering up the main line of execution of your code by error handling routines. You can create new types of exceptions by extending existing ones. Exceptions propagate through the stack via the buck–passing mechanism. If they are unhandled the the program crashes. You should not use exceptions for the ordinary course of your code as an alternative branching mechanism. This is an abuse of excepton handling.

Chapter 9

Text File IO

9.0 Introduction

A fundamental operation of computer applications is that of reading from and writing to files. This allows us to create permanent records of our activity on the disk and to reopen it later for further editing or use. We will concern ourselves chiefly with two major types of files: text files and files that serialize, or pack away, objects which can later be deserialized, or unpacked.

We will begin, in this chapter, with text files, and do a case study of creating a simple text editor that opens text files for editing in a simple GUI. Along the way we shall meet a new and very useful widget, the `JFileChooser`. Common to both of these streams is the `File` class; we shall begin our exposition with that.

9.1 The File Class and Paths

The File class belongs to the package `java.io`; it is used to represent locations in your file system. The File class does not play a role in the actual reading or writing of data to a file. Instances of the `File` class can point at files or directories stored on your system. There are other objects that handle the actual mechanics of file IO.

Let us explore this class and see what it does. Open a DrJava interactive session, and the Java API guide. Let us begin by looking at the Field Summary in the guide. It features four fields. Really, they harbor two pieces of data, the path separator and the separator character. This character can be yielded up as a character or a string. Hence the existence of four constants. Notice that these constans are static, so we can call them by class name.

```
> import java.io.File;
> File.pathSeparator
":"
> File.pathSeparatorChar
:
> File.separator
"/"
> File.separatorChar
/
>
```

The purpose of the separator character is to separate files in a path. This character is a on Windoze systems, and it is a / on a UNIX system. The notion of path is common to all operating systems. Recall a path consists of a sequence of directories followed by a directory or file. The separator character can be expanded to "and then into." Only the last item in a path can be a regular file; all others must be directories.

For example the path

```
animals/mammals/tapir.html
```

specifies a file tapir.html that lives inside of directory mammals, which in turn lives inside of directory animals. In Windoze, this path is specified by

```
animals\mammals\tapir.html
```

Common to the command line interfaces of Windoze and UNIX is the notion of `search path`. In UNIX if you enter the command `ls`, it not in your `cwd`. Therefore UNIX checks your search path, which is a list of paths to directories for the presence of `ls`. It checks this list in order; if it finds the command in some directory, it immediately executes it. Since `ls` lives in `/bin`, the directory `/bin` must be in your path for `ls` to run. Fortunately, this is done for you by default. Windoze also has a search path mechanism that works in an identical way. Let us show the path on both systems. First on UNIX, we see the path by entering `echo $PATH`at the command prompt.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$
```

In Windoze, bring up a console window by going to the run menu item in the Start menu, and type `cmd` into the text slot. Then, in this little black window, type `PATH` at the prompt.

```
Microsoft Windows XP [Version 5.1.2600]
```

```
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\John Morrison>PATH
PATH=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
```

In both systems, there is an environment variable containing the search path. In UNIX this is `$PATH`, in Windoze it is `PATH`. You will also notice that the search path is a list of absolute paths. Absolute paths in Windoze begin with a drive letter such as `C:`; in UNIX they begin with a `/`. Observe that the directory `/bin` is listed on the path of this machine, so ls there is found and run. If you type an command that does not live in your path, you will get an error message, as seen here.

```
$ ontv
bash: ontv: command not found
$
```

We sent UNIX on a wild goose chase and we get rewarded with a nastygram. If you are running a MAC you will see the same things. In Windoze, you will see this, informing us it has been sent on a similarly futile wild goose chase.

```
C:\Documents and Settings\John Morrison>ontv
ontv is not recognized as an internal or external command,
operable program or batch file.
```

We see that the path separator, which is : in UNIX and ; in Windoze, separates directories in the search path.

9.2 Constructors and Methods

We shall most commonly use the constructor

```
public File(String pathname)
```

to create instances of `File` objects. Such an object may point at a directory or a regular file. This is not surprising to UNIX users, since they know that directories are just special files that contain an index to their contents. You may use an absolute path name, or a relative path name. A relative path name will be relative to the `cwd` of the java program when it is running on the users machine.

The `exists()` method can be used to see if a given file already exists on the users system. Here we show a brief example.

```
> f = new File(".");
> f.exists()
true
> g = new File("someFileThatDoesNotExist")
> g.exists()
false
>
```

We began by making f point at the Java programs `cwd`. Naturally, this must exist. We then deliberately chose a file that does not exist in the programs cwd, and we see that exists() discerns its nonexistence. This method can be very useful for performing file IO; you can use it to avoid clobbering an existing (valuable) file on the (hapless) users system.

The `canRead()`, `canWrite()`, and `canExecute()` methods are self-explanatory. They come in handy: you can check to see if you have permission for gaining access to a file prior to charging forth. This can save the throwing of an exception.

The methods `getPath()` and `getAbsolutePath()` will return the string representation of the path to a file from the program's `cwd`. You should make a directory, place a few unvaluable files in it, and experiment with the methods in the API. You can remove files, make new directories, and do all manner of file management with this class. Do the simple exercises below in a program or in an interactive DrJava session.

Programming Exercises

1. Make a new `File` object and use it to determine the absolute path of your `cwd`.
2. Perform ls -l on your cwd. Try resetting permission bits on one of your files by creating a file object. Verify what you did using ls -l, and by using methods from the `File` class.
3. Make a `File` object in the interactive prompt and change its `cwd` to various places. Check for existence and nonexistence of various files. See if you can determine what permissions you have for the files. Can you check if a file is a directory?
4. See if you can write a program that takes a file or directory as command line argument and which imitates the action of the UNIX command `ls`.

9.3 A Simple Case Study: Copying a File

During this section you will see how to read from and write to a text file. We shall emulate the action of the cp command in UNIX. The usage for our program

will be

```
$ java Copy donorFile recipientFile
```

and its action will be to copy the contents of the donor file into the recipient file. It will clobber any recipient file that already exists, just as the UNIX `cp` command does. You may enter this command in the DrJava interactions pane to execute it as well as at the command prompt.

Let us begin by creating the class `Copy`. We will make our method be a static method inside of this class. Here is the start.

```
public class Copy
{
    public static void copy(String donor, String recipient)
    {
    }
}
```

This file compiles happily. Now, inside of the `copy` method, add this code.

```
    donorFile = new File(donor);
    recipientFile = new File(recipient);
```

Also, add this import statement at the top of the program.

```
import java.io.File;
```

to avoid angry yellow. The resulting code will compile. Our `File` objects just point to paths in the file system, which might or might not exist. Next, let us open the donor file for reading and the recipient file for writing. To do so, we use a `FileReader` as follows.

```
FileReader fr = new FileReader(donorFile);
```

and a `FileWriter` as follows.

```
FileWriter fw = new FileWriter(donorFile);
```

Take a trip to the API guide for the `FileReader` class. We are using the constructors

```
FileReader(File file)
```

and

```
FileWriter(File file)
```

to create the code above.

Next, we will read each line from the donor file, then write them to the recipient file. Adding in the appropriate exception handling yields the following class. Notice how the `copy` method passes the buck and forces the caller to handle any exception it generates.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Copy
{
    public static void copy(File donorFile, File recipientFile)
       throws IOException
    {
        FileReader fr = new FileReader(donorFile);
        FileWriter fw = new FileWriter(recipientFile);
    }
    public static void main(String[] args)
    {
        try
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

9.3.1 A Programming Idiom

We now need to read the contents of the donor file. We shall read it, a line at at time, and in turn write each line to the donor. You might think, "How do I iterate through a file? It was so easy in Python with a `for` loop!" Here is an

idiom that does exactly that.

```
int ch;
while( (ch != fr.read()) != -1)
{
    //Process each character of the file.
}
```

What we are doing here is to write each character of the donor to the recipient in the loop. We next close each file so they are properly saved and so system resource are liberated.

```
int ch;
while( (ch = fr.readLine()) != -1)
{
    fr.write(ch);
}
fw.close();
fr.close();
```

Putting it all together we get this class.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Copy
{
    public static void copy(File donorFile, File recipientFile)
        throws IOException
    {
        FileReader fr = new FileReader(donorFile);
        FileWriter fw = new FileWriter(recipientFile);
        int ch;
        while( (ch = fr.read()) != -1 )
        {
            fw.write(ch);
        }
        fw.close();
        fr.close();
    }
    public static void main(String[] args)
```

```
    {
        try
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

To run this program in DrJava with command line arguments, just use the following command.

```
> java Copy donorFile recipientFile
```

Likewise, at the UNIX prompt, we an use the same command as follows.

```
$ java Copy donorFile recipientFile
```

9.3.2 Buffered FileIO

When you read a character from a file using a `FileReader`, you are making a request of the operating system to see in that file and fetch that character. This is a fairly costly proceeding. There are times when you are programming with devices such as terminals when you want to do this. However, with what we are doing with text files, this sort of character-by-character retrieval is unnecessary and wasteful of system resources.

We make our appliction much faster by using *buffered* fileIO. A buffer is simply a temporary storage space. Your refrigirator acts as a buffer. Periodically, you go to the grocery store, fetch what you need and store it in the 'fridge. This saves time and money since you do not have to go to the store every time you need a food item. Your fridge is, effectively, a food intake buffer. You also likely have a recycling bin in the garage. You place recyclables in the bin, which is either periodically collected or which you periodically empty at the recycling center as it fills. Buffers ease the transfer of stuff.

Java has two standard libary classes for buffered fileIO, `java.io.BufferedReader` and `java.io.BufferedWriter`. These fetch bytes from a file one disk sector

(usually 4K) at a time, and then you can read from the buffer. When the buffer empties, another request is made to the operating system to refill it, until you come to the end of the file. All of this happens behind the scenes, so you need not worry about it.

Here is a code snippet that creates a buffered reader.

```
BufferedReader bf = new BufferedReader(new FileReader(someFile));
```

You pass the buffered reader a file reader that is connected to some file. As the buffered reader reads from the file, it can throw various IOExceptions. Said exceptions must be handled.

Analoglously, a buffered writer is created as follows.

```
BufferedWriter bw = new BufferedWriter(new FileWriter(someFile));
```

We will now create a new class for copying files that uses buffered reading and writing.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BufferedCopy
{
    public static void copy(File donorFile, File recipientFile)
       throws IOException
    {
        String line;
        BufferedReader fr = new BufferedReader(new FileReader(donorFile));
        BufferedWriter fw = new BufferedWriter(new FileWriter(recipientFile));
        while( (line = fr.readLine()) != null)
        {
            fw.write(line + "\n");
        }
        fr.close();
        fw.close();
    }
    public static void main(String[] args)
    {
        try
```

```
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

Notice that when using `write` we had to furnish an end-of-line character. This is because the `readLine()` method strips off the end-of-line character. This was not a worry in unbuffered IO since all characters, including end-of-line characters are extracted from the donor and put to the recipient.

We show a performance comparision. You can see that the buffered version is quite a bit faster. We created a file called `megazero.txt` which contains 12500 lines, each containing 79 zeroes and one newline character. We remove the recipient file in between the two tests to ensure "fairness."

```
$ time(java Copy megazero.txt foo.txt)
real    0m0.552s
user    0m0.640s
sys 0m0.060s
$ rm foo.txt
$ time(java BufferedCopy megazero.txt foo.txt)
real    0m0.225s
user    0m0.260s
sys 0m0.040s
$
```

Programming Exercises Now it's time to write some programs and practice what you have seen.

1. Write a program called `Cat.java` that takes a list of regular files as arguments and which puts them to `stdout` *in seratum.*
2. Write a program called `Ls.java` that lists the files in the directory passed it as a command–line argument in long format.
3. Add a feature to `Copy.java` that checks if the recipient file exists and pops up a `JOptionPane` asking the user if he wants to overwrite the recipient.

9.4 Opening a File in a GUI Window

Let us begin by making the usual graphical shell. Compile this and make sure it runs for you. Running it should reveal an empty GUI window.

```
import javax.swing.JFrame;
public class FileGUI extends JFrame implements Runnable
{
    public void run()
    {
        setSize(600,800);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        FileGUI fg = new FileGUI();
        javax.swing.SwingUtilities.invokeLater(fg);
    }
}
```

We now add some features. First, we shall make a `File` be a state variable, and add a constructor that initializes the file and which displays its locatoion in the title bar. This will be the file that will be displayed in the GUI window. We shall also include the stuff needed to make FileIO work. Notice how we set things up so the file's absolute path is displayed in the title bar of our app.

```
import javax.swing.JFrame;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class FileGUI extends JFrame implements Runnable
{
    File file;
    public FileGUI(String s)
    {
        super("FileGUI:  " + (new File(s)).getAbsolutePath());
        file = new File(s);
    }

    public void run()
    {
        System.out.println(file);
```

```
        setSize(600,800);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        FileGUI fg = new FileGUI(args[0]);
        javax.swing.SwingUtilities.invokeLater(fg);
    }
}
```

You run this with the command

```
> java FileGUI someFile.txt
```

9.4.1 Designing the Application

Here is an outline of what we need to do. We used top-down design to break out the steps into codeable pieces. We now integrate the process. Along the way we meet some new standard library classes.

`StringBuffer` This is a form of mutable string. The characters it accumulates are not pooled. You extract its contents by calling its `toString()` method. The default constructor creates an empty `StringBuffer`.

`javax.swing.JTextArea` This is a box into which text can be placed. It can be added to any container class.

`javax.swing.JScrollPane` This is a container class into which you can place a `JTextArea`. If the text is too large to show in its entirety, then scroll bars automatically appear.

Now we make a plan as to how to proceed.

1. Create a `StringBuffer`.
2. Open the file.
3. Suck the file into a `StringBuffer`, for safekeeping before displaying it.
4. Close the file.
5. Display the file.
   (a) Make a `JTextArea`
   (b) Put it in the content pane.

(c) Put the `StringBuffer`'s contents into the `JTextArea`

6. Give the `JTextArea` scrollbars (the file may be too big).
7. Run and compile the first part and check for errors.

```
import javax.swing.JFrame;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class FileGUI extends JFrame implements Runnable
{
    File file;
    public FileGUI(String s)
    {
        super("FileGUI:  " + (new File(s)).getAbsolutePath());
        file = new File(s);
    }

    public void run()
    {
        System.out.println(file);
        setSize(600,800);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        //Create the StringBuffer.
        StringBuffer sb = new StringBuffer();
        try
        {
            //temporary code to make sure file is found
            System.out.println(file.getAbsolutePath());
            //read file and create BufferedReader
            FileReader fr = new FileReader(file);
            BufferedReader r = new BufferedReader(fr);
        }
         catch(FileNotFoundException e)
        {
            System.err.printf("File %s was not found", file.getAbsolutePath());
        }
        catch(IOException e)
        {
            System.err.println("IO Exception occurrred");
        }


        setVisible(true);
    }
```

```
    public static void main(String[] args)
    {
        FileGUI fg = new FileGUI(args[0]);
        javax.swing.SwingUtilities.invokeLater(fg);
    }
}
```

And now for the rest

```
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import java.io.File;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class FileGUI extends JFrame implements Runnable
{
    File file;
    public FileGUI(String s)
    {
        super((new File(s)).getAbsolutePath());
        file = new File(s);
    }

    public void run()
    {
        System.out.println(file);
        setSize(600,800);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        //Create the StringBuffer
        StringBuffer sb = new StringBuffer();;
        try
        {
            //open the file and create a BufferedReader
            FileReader fr = new FileReader(file);
            BufferedReader r = new BufferedReader(fr);

            String buf ="";
            //suck the file into the StringBuffer
            while( (buf = r.readLine())  != null)
            {
                sb.append(buf + "\n");
            }
```

```
            //Close the file
            r.close();
        }
        catch(FileNotFoundException e)
        {
            System.err.printf("File %s was not found", file.getAbsolutePath());
        }
        catch(IOException e)
        {
            System.err.println("IO Exception occurrred");
        }
        //make the JTextArea; don't forget the import
        JTextArea jta = new JTextArea();
        //Make a JScrollPane and do the import.
        //Put the JTextArea inside of the JScrollPane
        JScrollPane jsp = new JScrollPane(jta);
        //Put the JScrollPane into the Content Pane.
        getContentPane().add(jsp);
        //Put the contents of the String buffer inot the JTextArea.
        jta.setText(sb.toString());
        setVisible(true);
    }
    public static void main(String[] args)
    {
        FileGUI fg = new FileGUI(args[0]);
        javax.swing.SwingUtilities.invokeLater(fg);
    }
}
```

Programming Exercises

1. Put a `JTextField` at the top of this application, in which the user is to enter a file name. Attach an action listener which causes the file entered to be displayed. If the end-user goofs up, put the message "No such Annie–Mule" into the `JTextArea` and clear the `JTextField`.
2. Can you make the `JTextField` uneditable? Look in the API guide.

9.5 Swing's `ImageIO` Class

We will show how to obtain an image from a file and how to draw it in a `JPanel`. Let us begin with this shell program.

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

```
import javax.swing.JPanel;

public class ImageFrame extends JFrame implements Runnable
{
    Image image;
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new ImagePanel());
        setVisible(true);
    }
    public static void main(String[] args)
    {
        ImageFrame iframe = new ImageFrame();
        javax.swing.SwingUtilities.invokeLater(iframe);
    }
    class ImagePanel extends JPanel
    {
    }
  }
```

Run this and you will just see a blank window. Now place an image file in the same directory as this program. I have the image `myPic.jpg`. This technique works for all standard image formats including such formats `.gif` and `.png`.

We will be using the class `javax.swing.ImageIO`. It has a static method `read` which accepts an image file as an argument and which produces an instance of `java.awt.Image`. This can be put to the `ImagePanel` using the inherited `drawImage` method as follows. Noice the additional imports that become necessary, and the needed `try-catch` sequence.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.io.IOException;
import java.io.File;
import javax.imageio.ImageIO;
import java.awt.Image;
import java.awt.Graphics;

public class ImageFrame extends JFrame implements Runnable
{
    Image image;
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new ImagePanel());
```

```
        setVisible(true);
    }
    public static void main(String[] args)
    {
        ImageFrame iframe = new ImageFrame();
        javax.swing.SwingUtilities.invokeLater(iframe);
    }
    class ImagePanel extends JPanel
    {
        public void paintComponent(Graphics g)
        {
            try
            {
                image = ImageIO.read(new File("myPic.jpg"));
                g.drawImage(image, 100, 100, null);
            }
            catch(IOException ex)
            {
                System.err.printf("File %s could not load\n", "myPic.jpg");
            }

        }
    }
}
```

Programming Exercises

1. Add a File menu to this application
2. Add items `open` and `quit`.
3. Make the `quit` item live.
4. Put a `JTextField` in the top of the `ImageFrame`. Attach an action listener so that the text in it is grabbed and the image file placed in it is opened.
5. Use appropriate exception handling; if the user goofs, leave the existing image in the frame.

Chapter 10

The NitPad Case Study

10.0 Case Study: NitPad: A Text Editor

Now we shall look at a simple, full-featured application that will be quite similar to the Notepad application you see on certain dark side machines.

We are going to think about this program from the standpoint of the user. We must be vigilant and protect the user from data loss. Since he is a paying customer, we must see carefully to his needs.

Let us begin by creating a graphical shell

```
import javax.swing.JFrame;
public class Nitpad extends JFrame implements Runnable
{
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
    }
    public void run()
    {
        setSize(600,600);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Run this code and a blank window should appear on your screen with the string `"Nitpad: Unsaved *"` emblazoned on the title bar. Let us begin by making some design decisions. For the main text area, we will use a `JEditorPane`; this is a flexible widget in which you may type text. We will put this inside of a `JScrollPane` and add it to the content pane. Placing it in a `JScrollPane` causes scrollbars to materialize when the text is too large to display in the content pane. Let us begin by doing that.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JEditorPane;
public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Why is the `JEditorPane` a state variable? Open Notepad and look at it. You can see that its file and edit operations are driven by menus. Since this is to happen, it will be helpful to be able to communicate with `jep` throughout the entire program. Notice that we initialized this state variable in the second line of the constructor.

Run the program; observe that you can type characters into the `JEditorPane`.

10.0.1 Laying out Menus

Our program will have two menus, `File` and `Edit`. The file menu should have the standard list of menu items: New, Open, Save, Save As, and Quit. The Edit

menu will feature Cut, Copy, Paste and Select All. Let us now create the menus and menu items so we can see them. First, remember we need these includes.

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

We then remember to do the following.

1. Nail in the menu bar with this code.

   ```
   JMenuBar mbar = new JMenuBar();
   setJMenuBar(mbar);
   ```

2. Create and add in the two menus.

   ```
   JMenu fileMenu = new JMenu("File");
   JMenu editMenu = new JMenu("Edit");
   mbar.add(fileMenu);
   mbar.add(editMenu);
   ```

3. Populate the menus.

   ```
   JMenuItem newItem = new JMenuItem("New");
   JMenuItem openItem = new JMenuItem("Open");
   JMenuItem saveItem = new JMenuItem("Save");
   JMenuItem saveAsItem = new JMenuItem("Save As...");
   JMenuItem printItem = new JMenuItem("Print");
   JMenuItem quitItem = new JMenuItem("Quit");
   fileMenu.add(newItem);
   fileMenu.add(openItem);
   fileMenu.add(saveItem);
   fileMenu.add(saveAsItem);
   fileMenu.add(printItem);
   fileMenu.add(quitItem);
   JMenuItem copyItem = new JMenuItem("Copy");
   JMenuItem pasteItem = new JMenuItem("Paste");
   JMenuItem cutItem = new JMenuItem("Cut");
   JMenuItem selectAllItem = new JMenuItem("Select All");
   editMenu.add(copyItem);
   editMenu.add(pasteItem);
   editMenu.add(cutItem);
   editMenu.add(selectAllItem);
   ```

Here is our latest result. Run it and see all of the menus and menu items being displayed. We have created much of the view for this application and a very little of the model. However, we should be pleased by the app's appearance. We also added here code to set the font in the constructor. Notice the appearance of another import.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JEditorPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.Font;
public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
        jep.setFont(new Font("Monospaced", Font.PLAIN, 12));
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        makeMenus();
        setVisible(true);
    }
    public void makeMenus()
    {
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        mbar.add(fileMenu);
        mbar.add(editMenu);
        JMenuItem newItem = new JMenuItem("New");
        JMenuItem openItem = new JMenuItem("Open");
        JMenuItem saveItem = new JMenuItem("Save");
        JMenuItem saveAsItem = new JMenuItem("Save As...");
        JMenuItem printItem = new JMenuItem("Print");
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(newItem);
        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.add(saveAsItem);
        fileMenu.add(printItem);
        fileMenu.add(quitItem);
        JMenuItem copyItem = new JMenuItem("Copy");
        JMenuItem pasteItem = new JMenuItem("Paste");
        JMenuItem cutItem = new JMenuItem("Cut");
```

```
        JMenuItem selectAllItem = new JMenuItem("Select All");
        editMenu.add(copyItem);
        editMenu.add(pasteItem);
        editMenu.add(cutItem);
        editMenu.add(selectAllItem);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

10.0.2 Getting a File to Save via Menus

We begin by attaching empty action listeners to each `File` menu item. For example, in the `New` menu item, we proceed as follows.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
```

Do not forget to add these imports.

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

Proceed to attach an empty action listener to each `Edit` menu item.

Before going any further, we must make a plan for how these items are to work. *We must think about the user experience and design appropriately.* It is likely and desirable that we will develop some private methods that will be used by the listeners to do their jobs. Once we make a plan, it will be clear what functions will be executed by more than one listener. Said function should be coded as private methods.

Now we must think first about the state of our program. So far, it has a `JEditorPane` which can accept typed text. However, this pane does not know what to do with the text. What other state variables do we need?

Open Notepad on a Windoze machine. Any instance of Notepad can open exactly one file. Either we are typing in an unsaved window or a window pointing at a file. This tells us we need a `File` object representing our current file. When

Nitpad is first started, we will make it a `null` object. Add a state variable as follows.

```
private File currentFile;
```

Add this line to the constructor.

```
currentFile = null;
```

Let us begin by trying to save a file. We need to plan the action of the listener attached to the `Save` menu item. Let us now write a plan in comments in its code.

```
saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            //if currentFile is null, prompt to select
            //a file as the current file

            //write to the current file

            //place the absolute path of the current file
            //in the title bar.
        }
    });
```

Now let us look at `Save As` before moving ahead.

```
saveAsItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            //prompt to select
            //a file as the current file to save to.

            //write to the current file

            //place the absolute path of the current file
            //in the title bar.
            }

        }
    });
```

You can see some commonalities in the functions of the two menu item's action listeners. Both prompt to select a current file. Both write to the current file.

Both update the title bar. This points up the need for three private methods. Notice how we pass the buck with the `IOException` in the first method. You can add these method stubs to your class. Make sure you import the class `java.io.IOException`. Note we also devleop a function to select a currentFile, because it will be a common operation

```
private void  writeToCurrentFile() throws IOException{}
private void fixTitleBar(){}
private void promptToSave() {}
private void selectCurrentFile() {}
```

Let us try coding up these methods, beginning with `writeToCurrentFile()`. rere are the basic steps.

1. Create a buffered writer linked to the current file.
2. Extract the text from the editor pane.
3. Write the text to the file
4. Close the file connection so the file is properly saved.

Now we code it up.

```
public void  writeToCurrentFile() throws IOException
{
   //Create a buffered write linked to the current file.
    BufferedWriter bw = new BufferedWriter(
        new FileWriter(currentFile));
    //Extract the text from the editor pane.
    String blob = jep.getText();
    //Write the text to a file.
    bw.write(blob);
    //Close the file coonection.
    bw.close();
}
```

You might properly ask: *What if the current file is null?* We have blithely ignored that possibility. Coördinating that problems is best left to the action listener. We are adhering to the design prinicple of *atomicity of purpose*; i.e. our function does exactly one thing. Other objects will orchestrate its behavior. It is a precondition of this function that a valid current file be selected.

It is now time to test this out. We are going to show a primitive version of our app that just saves to a file named `hammer.txt`. Here is the file with the additions we have made. We must now place code in the `Save` action listener to get things going.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JEditorPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.io.File;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    private File currentFile;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
        jep.setFont(new Font("Monospaced", Font.PLAIN, 12));
        currentFile = new File("hammer.txt");  //temporary code to
        //test the writeToCurrentFile method.  TODO: Get rid of this.
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        makeMenus();
        setVisible(true);
    }
    public void makeMenus()
    {

        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        mbar.add(fileMenu);
        mbar.add(editMenu);
        JMenuItem newItem = new JMenuItem("New");
        JMenuItem openItem = new JMenuItem("Open");
        JMenuItem saveItem = new JMenuItem("Save");
        JMenuItem saveAsItem = new JMenuItem("Save As...");
```

```
JMenuItem quitItem = new JMenuItem("Quit");
fileMenu.add(newItem);
fileMenu.add(openItem);
fileMenu.add(saveItem);
fileMenu.add(saveAsItem);
fileMenu.add(quitItem);
JMenuItem copyItem = new JMenuItem("Copy");
JMenuItem pasteItem = new JMenuItem("Paste");
JMenuItem cutItem = new JMenuItem("Cut");
JMenuItem selectAllItem = new JMenuItem("Select All");
editMenu.add(copyItem);
editMenu.add(pasteItem);
editMenu.add(cutItem);
editMenu.add(selectAllItem);
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
openItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file

        //place the absolute path of the current file
        //in the title bar.
    }
});
saveAsItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //prompt to select
        //a file as the current file to save to.

        //write to the current file

        //place the absolute path of the current file
        //in the title bar.
```

```
            }
        });
        quitItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
            }
        });

    }
    private void fixTitleBar(){}
    private void promptToSave(){}
    private void selectCurrentFile(){}

    private void writeToCurrentFile() throws IOException
    {
       //Create a buffered write linked to the current file.
        BufferedWriter bw = new BufferedWriter(
            new FileWriter(currentFile));
        //Extract the text from the editor pane.
        String blob = jep.getText();
        //Write the text to a file.
        bw.write(blob);
        //Close the file connection.
        bw.close();
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Let us now code this action listener. This is very minimal, but all we want to do is to test and see if the current file is getting written to `hammer.txt`. Add this to your class.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file
        try
        {
            writeToCurrentFile();
```

```
        }
        catch(IOException ex)
        {
            System.err.println("Not happy");
        }

        //place the absolute path of the current file
        //in the title bar.
    }
});
```

Now place text in the `JEditorPane` and select `Save` from the `File` menu. Close the application. You should have a file `hammer.txt` with the text you typed in it.

10.0.3 Is the Window Saved?

If we quit or application or navigate away from a file, we need to know if the the contents of the window are saved. This is necessary becasue we must protect the user from inadvertant data loss. Hence, we will add the state variable

```
private boolean saved;
```

In the constructor, we initialize as follows.

```
saved = false;
```

This brings up a whole set of problems we have to deal with.

- The `saved` state variable must be updated whenever the contents of the window change. How do we do that?
- As part of our interface, we will put a * at the end of the title bar when the contents of the window are not saved. How do we make this happend?
- Where are the perils of data loss? How do we prevent them? We must think that whenever we display a new file in the window, the changes to the old one must be saved.
- If the current file is null, we must act as if the window is unsaved.

Let us begin by getting up a mechanism to point to a current file. For now, you will need to add this empty methd which we will fill soon.

```
private void writeToCurrentFile() throws IOException(){}
```

Next, let us dispose of the matter of selecting the current file. Let us return `true` if a file is selected and `false` otherwise.

```
private boolean selectCurrentFile()
{
    boolean chosen = false;
    JFileChooser jfc = new JFileChooser();
    int willSave = jfc.showSaveDialog(Nitpad.this);
    if(willSave == JFileChooser.APPROVE_OPTION)
    {
        currentFile = jfc.setSelectedFile();
        chosen = true;
    }
    return chosen;
}
```

Note that we have changed the return type of this function, since we have seen a need to do so. The caller will need to know if a file is selected and saved to so it can act accordingly. Note the use of `JOptionPane`'s static constants to do the job. These are the integers that are returned by this method.

```
private int promptToSaveWindow()
{
    int choice = JOptionPane.showConfirmDialog(Nitpad.this,
        "Save document in window?");
    boolean selected = false;
    if(choice == JOptionPane.YES_OPTION)
    {
        selected = selectCurrentFile();
    }
    try
    {
        if(selected)
        {
            writeToCurrentFile();
        }
        else
        {
            choice = JOptionPane.CANCEL_OPTION;
        }

    }
    catch(Exception ex)
    {
        //bail out conservatively
        choice = JOptionPane.CANCEL_OPTION;
```

```
    }
    return choice;
}
```

Now we shold get rid of the temporay refrence to `hammer.txt` at the beginning and alter the constructor to read

```
currentFile = null;
```

Another matter of housekeeping is the title bar. Let's get it at least partially working.

```
    private void fixTitleBar()
    {
        //TODO  fix star
        String currentFileString = (currentFile == null)? "Unsaved":
            currentFile.getAbsolutePath();
        setTitle("Nitpad:  " + currentFileString);
    }
```

10.0.4 Getting Save and Save As to Work

Let us now turn our attention to the `actionPerformed` method in the `saveItem` action listener. Here is what we have so far.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file
        try
        {
            writeToCurrentFile();
        }
        catch(IOException ex)
        {
            System.err.println("Not happy");
        }

        //place the absolute path of the current file
        //in the title bar.
    }
});
```

If the current file is null we can select as follows

```
boolean saveWanted = false;
if(currentFile == null)
    saveWanted = selectCurrentFile();
```

Now we can make writing to the current file conditional on the save being wanted.

```
if(saveWanted)
{
    writeToCurrentFile();
    fixTitleBar();
    saved = true;
}
```

Now the method looks like this.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file
        boolean saveWanted = false;
        if(currentFile == null)
            saveWanted = selectCurrentFile();
        //write to the current file
        try
        {
            if(saveWanted)
            {
                writeToCurrentFile();
                //place the absolute path of the current file
                //in the title bar.
                fixTitleBar();
                saved = true;
            }
        }
        catch(IOException ex)
        {
            System.err.printf("Could not open %s\n",
                currentFile.getAbsolutePath());
        }
    }
});
```

Now we can write the `Save As` action listener pretty easily.

```
saveAsItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //prompt to select
        //a file as the current file to save to.
        boolean saveWanted = false;
        saveWanted = selectCurrentFile();
        //write to the current file and
        //place the absolute path of the current file
        //in the title bar.
        try
        {
            if(saveWanted)
            {
                writeToCurrentFile();
                fixTitleBar();
                saved = true;
            }
        }
        catch(IOException ex)
        {
            System.err.printf("Could not open %s\n",
                currentFile.getAbsolutePath());
        }
    }
});
```

10.0.5 Getting the `File` Menu in Order

Now we will turn to writing the rest of the action listeners for the `File` menu. This will require some careful design on our part to prevent data loss for the user. Let us begin with `New`.

If the current file is unsaved and `New` is selected, we must offer the user the opportunity to save his file before he navigates away from the window and loses his work. We force the user to deliberately abandon the contents of the window.

Once this is done, we set `currentFile` to `null` and we clear all text from the `JEditorPane`.

Be warned, we do not have the feature in place that monitors whether the text in the window is saved in full working order. We will do that after we get the menu items working. Let us plunge ahead recklessly, assuming that this desired feature is working. We begin with `New`; first we show the outline in

comments.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //If the contents of the window are unsaved,
        //prompt the user to save them.
        //Then, clear the JEditorPane and set the current
        //file to null.
    }
});
```

We now insert the code to make this work. Notice how we protect the user from losing data in the window and how we give lots of opportunity to back out whilst doing nothing.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //If the contents of the window are unsaved,
        //prompt the user to save them.
        int saveWindowChoice = -1;
        int saveFileChoice = -1;
        if(!saved)
        {
           saveWindowChoice = promptToSave();
            if(saveWindowChoice == JOptionPane.CANCEL_OPTION)
                ;//do nothing
            else if(saveWindowChoice == JOptionPane.NO_OPTION)
            {
                currentFile = null;
                jep.setText("");  //abandon changes
            }
            if(saveWindowChoice == JOptionPane.YES_OPTION)
            {
                if(currentFile == null)
                {
                    saveFileChoice = selectCurrentFile();
                    if(saveFileChoice)
                    {
                        writeToCurrentFile();
                    }
                }
            }
        }
        //Then, clear the JEditorPane and set the current
```

```
        //file to null.
    }
});
```

Chapter 11

The UniDraw Case Study and Serialization

11.0 Introduction

During this chapter we will create a program named `UniDraw`, which will be a simple drawing program with a color pen whose color and width are controlled by menus. The user will draw by dragging the mouse in the drawing panel, thereby creating a mark on the drawing.

We will be able to save our resulting creations for later viewing or editing. What is new here is that we will learn how to *serialize*, or pack away, objects into files and how to reconstitute them and restore the program's state at the time it was saved. This will be a full-featured event-driven graphical application.

11.1 Representing Curves

Let us begin by sketching in a bare-bones shell for our program. We will create two classes. We need a means with which to represent a curve in the drawing. To decide how to do this, let's be more specific about the application's functionality.

When the user presses the mouse, that triggers a `MouseEvent`; if the name of the event is `e`, we get the point at which it occurred by calling `e.getPoint()`. So, when the mouse is pressed, we should create a new curve and it should begin at the point we retrieved.

Now the user drags the mouse to draw. As this occurs, the mouse is *polled* and it periodically fires off a `MouseEvent`. Each event can tell us where it occurred; we will need to store all of those points in our curve.

Finally, when he mouse is released, the drag is over. We add the point where it is released, and then we wait for the next pressing of the mouse. Notice that every curve is *guaranteed* to have at least two points.

What does a curve need to know? It needs to know the list of points it accrues during the press/drag/release sequence. It also needs to know, according to our specification, its width and its color.

There is a vary natural way to achieve this. Let us make our curve class extend `ArrayList<Point>`. This class has an `add` method that will easily add points to the class. We will also make the curve capable of drawing itself in a graphical window. So we begin like so. Observe that a curve's color and width will never change, so we make these state variables `final`.

```
import java.awt.Color;
import java.awt.Point;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
}
```

Next we add a constructor to intilaize the state variables.

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.util.ArrayList;
public class Curve extends ArrayList<Point> //implements Serializable
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color = color;
        this.width = width;
    }
}
```

Finally, we will place a `draw` method in our `Curve` class.

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

```
import java.awt.Graphics;
import java.util.ArrayList;

public class Curve extends ArrayList<Point> //implements Serializable
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;
    }
    public void draw(Graphics g)
    {
    }
}
```

We now have a starting point for the `Curve` class. Place this code in a file named `Curve.java` and compile it. The remarks we made in this session about the mouse will help us to write the mouse event handlers needed to make the application work.

11.2 Getting Started on the Application

We will begin with a standard shell for a graphical application.

```
import javax.swing.JFrame;
public class DrawFrame extends JFrame implements Runnable
{
    public DrawFrame()
    {
        super("UniDraw:  Untitled");
    }
    public void run()
    {
        setSize(600,600);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        DrawFrame df = new DrawFrame();
        javax.swing.SwingUtilities.invokeLater(df);
    }
```

Place this code in a file named `DrawFrame.java`. Compile it and then run it. You should see a window appear on the screen with a title in the title bar.

Now it is time to think about what we want. We need a space for drawing. To do this, we need to subclass `JPanel` and then override `public void paintComponent(Graphics g)` to tell the drawing area how to render itself.

The drawing area must respond to the pressing and releasing of the mouse, so it must be a `MouseListener`. It must respond to the dragging of the mouse, so it must be a `MouseMotionListener`.

Communication will flow in ths way. The user will have selected a background color, a pen color and a width using the menus; these will all be given default values to start. These values need to be known by the application and will be determined by menu choices. The panel will need to have access to these values, so it is a good idea to make the panel an inner class. So our panel will have to extend `JPanel` and implement `MouseListener` and `MouseMotionListener`. We will need to implement the methods in the two interfaces so our class will compile. Now our code will look like this. We will add in the menu items, too, and make the `JMenuBar` a state variable so that we can separate the making of the various menus into separate functions. This will make managing our code simpler. We placed a `paintComponent` method in the draw panel so it would be easy to see that is got placed in the content pane.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.Color;
import java.awt.Graphics;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
    private JMenuBar mbar;
    public DrawFrame()
    {
        super("UniDraw:  Untitled");
        dp = new DrawPanel();
        mbar = new JMenuBar();
        setJMenuBar(mbar);
    }
```

```
public void run()
{
    setSize(600,600);
    makeFileMenu();
    makeColorMenu();
    makeBackgroundMenu();
    makeWidthMenu();
    getContentPane().add(dp);
    setVisible(true);

}
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
}
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Pen Color");
    mbar.add(colorMenu);
}
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
}
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
    mbar.add(widthMenu);
}
public static void main(String[] args)
{
    DrawFrame df = new DrawFrame();
    javax.swing.SwingUtilities.invokeLater(df);
}
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mouseEntered(MouseEvent e){}
```

```
        public void mouseExited(MouseEvent e){}
        public void mousePressed(MouseEvent e){}
        public void mouseReleased(MouseEvent e){}
        public void mouseClicked(MouseEvent e){}
        public void mouseMoved(MouseEvent e){}
        public void mouseDragged(MouseEvent e){}
        @Override
        public void paintComponent(Graphics g)
        {
            g.setColor(new Color(0xabcdef));
            g.fillRect(0,0,getWidth(), getHeight());
        }
    }
}
```

If you run this you should see the menus in the menu bar and a Carolina blue square in the content pane.

11.3 Deciding State in the Application

If we think about the user experience, figuring out state should be fairly easy. Lurking in the future is the issue of what we need to store in a file to reconstitute our drawing session. This informatoin needs to be included in the state variables.

We will begin by focusing on the mechanism of drawing in the draw panel. To draw the panel it seems we need to know the color for the background and the color the pen is now coloring. Hence we create these variables.

```
private Color background;//color of background
private Color foreground;//color of current curve
```

We also need to know the width of the pen. We create that variable too.

```
private int width;
```

Finally, we must deal with the drawing. We will record it as an array list of curves. We declare it as follows.

```
private ArrayList<Curve> drawing;
```

Once we create these variables, they must be initialized in the constructor.

```
public DrawFrame()
{
```

```
    super("UniDraw:  New File");
    dp = new DrawPanel();
    mbar = new JMenuBar();
    setJMenuBar(mbar);
    background = new Color(0xabcdef);
    foreground = new Color(0x228800);
    width = 1;
    drawing = new ArrayList<Curve>();
}
```

When we do this, let us update the `paintCompoent` method in the `DrawPanel` as follows. We want the chosen background color to color the background, not just `0xabcdef`.

```
@Override
public void paintComponent(Graphics g)
{
    g.setColor(background);
    g.fillRect(0,0,getWidth(), getHeight());
}
```

Drawing the drawing is simple. Since the `Curve` class has a `draw` method, we just tell each curve to draw itself using a collections `for` loop like so.

```
@Override
public void paintComponent(Graphics g)
{
    g.setColor(background);
    g.fillRect(0,0,getWidth(), getHeight());
    for(Curve c:drawing)
    {
        c.draw(g);
    }
}
```

Finally add this to your imports

```
import java.util.ArrayList;
```

and your progam will compile and run. It, however will not render curves in the drawing because we have not told the curves how to draw themselves. Our call to `s.draw(g)` does nothing. Here is the current state of our `DrawFrame` class.

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

```
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
    private JMenuBar mbar;
    private Color background;//color of background
    private Color foreground;//color of current curve
    private int width;
    private ArrayList<Curve> drawing;

    public DrawFrame()
    {
        super("UniDraw:  New File");
        dp = new DrawPanel();
        mbar = new JMenuBar();
        setJMenuBar(mbar);
         setJMenuBar(mbar);
        background = new Color(0xabcdef);
        foreground = new Color(0x001a57);
        width = 1;
        drawing = new ArrayList<Curve>();
    }

    public void run()
    {
        setSize(600,600);
        makeFileMenu();
        makeColorMenu();
        makeBackgroundMenu();
        makeWidthMenu();
        getContentPane().add(dp);
        setVisible(true);

    }
    public void makeFileMenu()
    {
```

```
        JMenu fileMenu = new JMenu("File");
        mbar.add(fileMenu);
    }
    public void makeColorMenu()
    {
        JMenu colorMenu = new JMenu("Pen Color");
        mbar.add(colorMenu);
    }
    public void makeBackgroundMenu()
    {
        JMenu backgroundMenu = new JMenu("Background");
        mbar.add(backgroundMenu);
    }
    public void makeWidthMenu()
    {
        JMenu widthMenu = new JMenu("Width");
        mbar.add(widthMenu);
    }
    public static void main(String[] args)
    {
        DrawFrame df = new DrawFrame();
        javax.swing.SwingUtilities.invokeLater(df);
    }
    class DrawPanel extends JPanel
        implements MouseListener, MouseMotionListener
    {
        public DrawPanel()
        {
            addMouseListener(this);
            addMouseMotionListener(this);
        }

        public void mouseEntered(MouseEvent e){}
        public void mouseExited(MouseEvent e){}
        public void mousePressed(MouseEvent e){}
        public void mouseReleased(MouseEvent e){}
        public void mouseClicked(MouseEvent e){}
        public void mouseMoved(MouseEvent e){}
        public void mouseDragged(MouseEvent e){}
        @Override
        public void paintComponent(Graphics g)
        {
            g.setColor(new Color(0xabcdef));
            g.fillRect(0,0,getWidth(), getHeight());
            for(Curve c: drawing)
            {
```

```
                c.draw(g);
            }
        }
    }
}
```

11.4 Getting the Curves to Draw: Getting `Curve.java` ready

Open the file `Curve.java`; here is its current state

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;

    }
    public void draw(Graphics g)
    {
    }
}
```

Notice that the draw method is unimplemented. Here is what we propose to do. For now we will sidestep the question of controlling width. The default width is 1; we shall begin by getting curves to draw on the screen.

1. Set the pen color to the color of the curve.
2. Connect each dot in the curve to its successor.

Our curve is an array list. This means it can learn its size by calling `size()`. We gain access to the points via the method `Point get(int index)`. Let us declare the following.

```
int n = size();
```

Now supppose `k` iterates through the list. We need to connect `get(k)` to `get(k+1)`. How do we do this? You must ask who is doing it. The pen has that job an it is an object of type `Graphics`, so go to the `Graphics` API page. Here is the method detail for `lineTo`

drawLine

```
public abstract void drawLine(int x1,
                              int y1,
                              int x2,
                              int y2)
```

Draws a line, using the current color, between the points `(x1, y1)` and `(x2, y2)` in this graphics context's coordinate system.

Parameters:

`x1` - the first point's x coordinate.
`y1` - the first point's y coordinate.
`x2` - the second point's x coordinate.
`y2` - the second point's y coordinate.

Now we begin to code our loop to draw the cure. We change the color of the pen to our color and then we connect each dot to its successor. Note the use of the `n -1`. Failure to do that will cause the endpoint to connect to a nonexistent sucessor. The result of that mistak would be an ugly `IndexOutOfBoundsException` being thrown.

```
public void draw(Graphics g)
{
    int n = size();
    g.setColor(color);
    for(int k = 0; k < n - 1; k++)
    {
        g.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
    }
}
```

Also, notice that there is no provision for connecting points so we had to break out each coördinate separately. Compile and run now. To finish, we must now make the panel's mouse methods work. The action of the mouse is not yet populating our curves or our drawings.

11.5 Getting the Curves to draw: Enabling the Panel

This is the current state of our `DrawPanel`.

```
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
    public void mouseDragged(MouseEvent e){}
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(new Color(0xabcdef));
        g.fillRect(0,0,getWidth(), getHeight());
        for(Curve c: drawing)
        {
            c.draw(g);
        }
    }
}
```

Let's step through what we had said before about the mouse and carefully outline what is to happen. Note that we will ignore most of the mouse methods.

1. When the user presses the mouse, create a new curve and add the point to it where the press occurred. Also, add the curve to the drawing.
2. When the mouse is dragged, accumulate the points where the mouse events are being fired during the polling process. As each mouse event is fired, repaint the panel so it stays up to date.
3. When the mouse is released, add the point of release to the curve, and repaint.

This means we will use the methods `mousePressed`, `mouseReleased` and `mouseDragged`. The rest get ignored. Let us begin with `mousePressed`. We must make a new curve and then add the point of the press to it. Remember, a mouse event has a `getPoint()` method that does the job.

```
public void mousePressed(MouseEvent e)
{
    Curve c = new Curve(color, width);
    c.add(e.getPoint());
    drawing.add(c);
}
```

When the mouse is released, we get the point, add it to the curve, and repaint.

```
public void mouseReleased(MouseEvent e)
{
    c.add(e.getPoint());
    repaint();
}
```

When the mouse is dragged, we get the point and repaint.

```
public void mouseDragged(MouseEvent e)
{
    c.add(e.getPoint());
    repaint();
}
```

Now compile. You will get this.

```
$ javac *.java
DrawFrame.java:87: error: cannot find symbol
            Curve c = new Curve(color, width);
                                ^
  symbol:   variable color
  location: class DrawFrame.DrawPanel
DrawFrame.java:92: error: cannot find symbol
            c.add(e.getPoint());
            ^
  symbol:   variable c
  location: class DrawFrame.DrawPanel
DrawFrame.java:97: error: cannot find symbol
            c.add(e.getPoint());
            ^
  symbol:   variable c
```

```
  location: class DrawFrame.DrawPanel
3 errors
$
```

That is ugly indeed. What happened? The variable `c` we created to point at the curve is local to `mousePressed`. Lift it up and make it a state variable and all will work. Now your class will look like this.

```
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    Curve c;            // c is now a state varible
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        c = null;  //initialize
    }
    //ignored mouse events
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
    //active mouse events
    public void mousePressed(MouseEvent e)
    {
        //remove "Curve" on this first line.
        c = new Curve(foreground, width);
        c.add(e.getPoint());
        drawing.add(c);
    }
    public void mouseReleased(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    public void mouseDragged(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(background);
        g.fillRect(0,0,getWidth(), getHeight());
```

```
            for(Curve c: drawing)
            {
                c.draw(g);
            }
        }
    }
```

11.6 Creating and Enabling the Color Menus

Next we need to get the background and foreground color menus working. We shall give each menu the Roy G. Biv colors; you will see that other colors caan be added easily; each new color can be added with a single line of code! We will also meet the `JColorChooser` widget, which will allow the user to pick a custom 24-bit color. It has the addional feature of allowing the user to choose *alpha*, or opacity of colors.

Alpha is determined by a numerical scale of 0 through `0xff`. The alpha value of 0 gives a perfectly transparent (non-)color. The alpha value of `0xff` gives a totally opaque color. Intermediate values yield transparaent layers of color. All colors are completely determined by a 32 bit integer, one byte for red, another for green, another for blue, and one for alpha.

Now we resume our main thread. We will create innner classes for the pen color menu items and the background color menu items. We begin with the pen color menu, making each menu item know its color.

```
public class PenColorItem extends JMenuItem
{
    private final Color color;
    public PenColorItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
    }

}
```

We will also attach an action listener which will set the pen color to the menu item's color. Note that the changing of the pen color does not require an update of the `DrawPanel`, so we do not have to repaint the panel.

```
public class PenColorItem extends JMenuItem
{
    private final Color color;
    public PenColorItem(Color _color, String colorName)
```

```
        {
            super(colorName);
            color = _color;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    foreground = color;
                }
            });
        }
    }
```

Now add these two lines and compile

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

Now let's build the pen color menu. We go into the pen color menu and populate it thusly with a red menu item. We will test this before proceeding with the rest.

```
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new PenColorItem(Color.red, "red"));
}
```

Drag the mouse in the window and see the red color drawn. We will now complete Roy G. Biv; note the hex codes used for indigo and violet.

```
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new PenColorItem(Color.red, "red"));
    colorMenu.add(new PenColorItem(Color.orange, "orange"));
    colorMenu.add(new PenColorItem(Color.yellow, "yellow"));
    colorMenu.add(new PenColorItem(Color.green, "green"));
    colorMenu.add(new PenColorItem(Color.blue, "blue"));
    colorMenu.add(new PenColorItem(new Color(0x2E0854), "indigo"));
    colorMenu.add(new PenColorItem(new Color(0x7D26CD), "violet"));
}
```

We still have 16,777,209 colors to go. How do we get them? We use a `JColorChooser` to choose a custom color. We will avail ourselves of its static

`showDialog` method to do the job. The constructor needs three arguments, a `JComponent`, a message (a string will do), and a default color. We will use our draw frame as the component, `"Choose Pen Color"` as our message, and the `foreground` color as our default color. By default, the color does not change. This is in keeping with minimizing user surprise.

Since there is some *ad hoc* procedure here, we will attach an anonymous inner class as a listener. The `JColorChooser` is goof-proof and requires no exception handling. Begin with an import.

```
import javax.swing.JColorChooser;
```

Now add the listener. at the end of the menu.

```
JMenuItem custom = new JMenuItem("Custom...");
colorMenu.add(custom);
custom.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        foreground = JColorChooser.showDialog(DrawFrame.this,
               "Choose Background Color", background);
    }
});
```

Select the new item from the menu and watch the (very cool) color chooser dialog pop up and offer you all of those choices.

Now wwe show the code for the background color. It is very similar, except that we must repaint when we change its color.

```
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
    backgroundMenu.add(new bgMenuItem(Color.red, "red"));
    backgroundMenu.add(new bgMenuItem(Color.orange, "orange"));
    backgroundMenu.add(new bgMenuItem(Color.yellow, "yellow"));
    backgroundMenu.add(new bgMenuItem(Color.green, "green"));
    backgroundMenu.add(new bgMenuItem(Color.blue, "blue"));
    backgroundMenu.add(new bgMenuItem(new Color(0x2E0854), "indigo"));
    backgroundMenu.add(new bgMenuItem(new Color(0x7D26CD), "violet"));
    JMenuItem custom = new JMenuItem("Custom...");
    backgroundMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
```

```
            background = JColorChooser.showDialog(DrawFrame.this,
                   "Choose Background Color", background);
            repaint();
        }
    });
}
```

Now we add the needed inner class to drive this.

```
class BackgroundMenuItem extends JMenuItem
 {
     final Color color;
     public BackgroundMenuItem(Color _color, String colorName)
     {
         super(colorName);
         this.color = _color;
         addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent e)
             {
                 background = color;
                 dp.repaint();
             }
         });
     }
 }
```

11.7 Consructing the Width Menu

We will construct the width menu and get it to change the state variable `width`, but we will also have to look at how curves draw themselves to get proper performance out of this menu. As of now, our pen only draws a path one pixel wide.

The inner class needed has one small surprise.

```
class WidthMenuItem extends JMenuItem
{
    final int width;
    public WidthMenuItem(int _width)
    {
        super("" + _width);
        width = _width;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
```

```
            {
                DrawFrame.this.width = width;
            }
        });
    }
}
```

Note the use of `DrawFrame.this` to specify that we want the ennclosing class's `width` variable to have a new value assigned to it.

We then turn to constructing the menu. Begin by doing this import

```
import javax.swing.JOptionPane;
```

Now modify `makeWidthMenu()` as follows.

```
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
    mbar.add(widthMenu);
    widthMenu.add(new WidthMenuItem(1));
    widthMenu.add(new WidthMenuItem(2));
    widthMenu.add(new WidthMenuItem(5));
    widthMenu.add(new WidthMenuItem(10));
    widthMenu.add(new WidthMenuItem(20));
    widthMenu.add(new WidthMenuItem(50));
    JMenuItem custom = new JMenuItem("custom...");
    widthMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            width = JOptionPane.showInputDialog(DrawFrame.this,
            }

        }
    });
}
```

Extreme Danger! What happens if the user types in something that is not a valid integer. Let's see what happens when the user types in `"cows"`. We get this horrid thing.

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException:
    For input string: "cows"
    at java.lang.NumberFormatException.forInputString
```

```
          (NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at DrawFrame$3.actionPerformed(DrawFrame.java:117)
               .
               .
          (much horrid stuff)
               .
               .
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:139)
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:97)
```

It is not acceptable for a user goof to produce this. We see that an unhandled `NumberFormatException` has bubbled up and caused havoc. We remedy this by placing an appropriate catch-try sequence in the listener as follows.

```
custom.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        String  buf = JOptionPane.showInputDialog(DrawFrame.this,
           "Specify width");
        try
        {
            width = Integer.parseInt(buf);
        }
        catch(NumberFormatException ex)
        {
            JOptionPane.showMessageDialog(DrawFrame.this,
               "String \"" + buf + "\" is not an integer.");
        }

    }
});
```

No action will be taken in the draw panel, but the user will be alerted to his mistake. The menu does what it is supposed to, but we still have no control of the pen's actual width. For this, we must learn about a new and helpful class.

11.8 Graphics2D

If you scour the page for `java.awt.Graphics`, you will find no facility for controlling the pen's width. So, when we get desperate what do we do? We look in the ancestor classes. These are barren of anything useful. Where now? We look

in some child classes. The `Graphics` class has child class `Graphics 2D`, which will help solve our problem.

If you scroll down on the API page, you will see a section entitled "Default Rendering Attributes. It says this.

> `Paint`
> The color of the Component.
> `Font`
> The Font of the Component.
> `Stroke`
> A square pen with a linewidth of 1, no dashing, miter segment joins and square end caps.
> `Transform`
> The getDefaultTransform for the GraphicsConfiguration of the Component.
> `Composite`
> The AlphaComposite.SRC_OVER rule.
> `Clip`
> No rendering Clip, the output is clipped to the Component.

What is important to us is that the stroke of the pen has a line width of 1. It also says there are "miter segment joins" and "square end caps." We also see that there is a `setStroke` method. Here is its method detail.

> `public abstract void setStroke(Stroke s)`
> Sets the Stroke for the Graphics2D context.
> **Parameters:**
> - the Stroke object to be used to stroke a Shape during the rendering process
> **See Also:**
> `BasicStroke, getStroke()`

We see that the classes `Stroke` and `BasicStroke` might also be of importance. We learn that `Stroke` is an interface, and that `BasicStroke` is a class that implements it. We can create a `BasicStroke` and use the `setStroke` method for the `Graphics2D` pen to obtain a pen whose with is larger than 1.

Inside of a `Graphics` lurks a `Graphics2D` You are guaranteed to be able to make the cast

```
(Graphics2D) g
```

if `g` is a `Graphics` object. You always have access to a `Graphics2D` if you are rendering inside of a `Swing` object. This is the type of object the `Swing` paint engine uses.

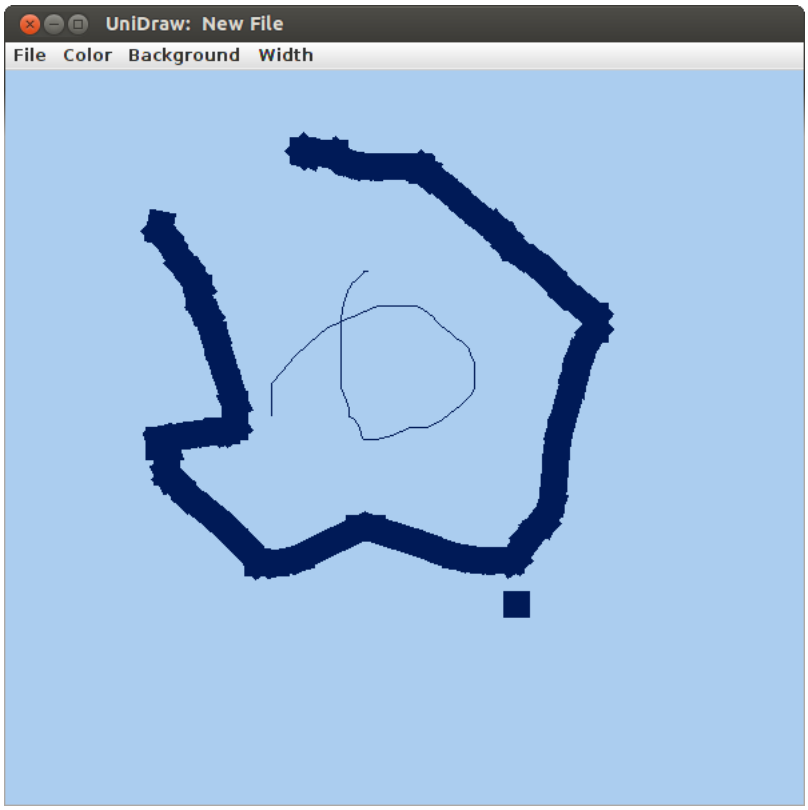
Now go to the `BasicStroke` constructor list. There is a constructor that will accept a `float`, and therefore which accepts our `width`. We will now use this to upgrade the `draw` method of the `Curve` class. We begin by inserting the needed cast.

```
public void draw(Graphics g)
 {
     Graphics2D g2d = (Graphics2D) g;
     g2d.setColor(color);
     int n = size();
     for(int k = 0; k < n - 1; k++)
     {
         g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
     }
 }
```

Compile and run. You will see that nothing has changed. We will now call `g2d`'s `setStroke` method and feed it a `BasicStroke` with `width`.

```
public void draw(Graphics g)
 {
     Graphics2D g2d = (Graphics2D) g;
     g2d.setStroke(new BasicStroke(width));
     g2d.setColor(color);
     int n = size();
     for(int k = 0; k < n - 1; k++)
     {
         g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
     }
 }
```

Now let's draw in the window. Choose a width of 20. The result looks like this.



The fat curve looks as if it were painted by watery, spattery paint. You can see where we drew a dot; it shows a square pen tip. We are not happy with this result and need to improve it.

How do we improve its appearance? This is where we need to look at the static constants in the `BasicStroke` class.

`BasicStroke.CAP_BUTT`	This ends a path with a semicircle.
`BasicStroke.CAP_ROUND`	This ends a path with a semicircle.
`BasicStroke.CAP_SQUARE`	This ends a path with a half a square.
`BasicStroke.JOIN_SQUARE`	This joins path segments by connecting the outer corners of their wide outlines wiht a straight segment.
`BasicStroke.JOIN_MITER`	This joins path segments by extending their outside edges until they meet.
`BasicStroke.JOIN_ROUND`	This joins path segments by rounding off the corner at a radius half of line width.

Use them in the constructor `BasicStroke(float width, int cap, int join)`.
You will select one `CAP` constant and one `JOIN` constant. We will use `BasicStroke.CAP_ROUND`

and `BasicStroke.JOIN_ROUND`. Here is the new appearance for our `draw` method in the `Curve` class.

```
public void draw(Graphics g)
{
    Graphics2D g2d = (Graphics2D) g;
    g2d.setColor(color);
    int n = size();
    g2d.setStroke(new BasicStroke(width, BasicStroke.CAP_ROUND,
         BasicStroke.JOIN_ROUND));
    for(int k = 0; k < n - 1; k++)
    {
        g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
    }
}
```

Now run again and compile. Notice the spiffy appearance of the fat curve. Also notice that clicking in a spot yields a circular blob of Dook-blue ink. The use of the cap and join parameters cleans things up and gets rid of the splatter.

Let us now collect the current state of the files. First, `Curve.java`.

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.BasicStroke;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;

    }
    public void draw(Graphics g)
    {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setColor(color);
        int n = size();
        g2d.setStroke(new BasicStroke(width, BasicStroke.CAP_ROUND,
             BasicStroke.JOIN_ROUND));
        for(int k = 0; k < n - 1; k++)
        {
            g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
        }
    }
}
```

Next, `DrawFrame.java`.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JColorChooser;
import javax.swing.JOptionPane;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

```
import java.awt.event.MouseMotionListener;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
    private JMenuBar mbar;
    private Color background;//color of background
    private Color foreground;//color of current curve
    private int width;
    private ArrayList<Curve> drawing;

    public DrawFrame()
    {
        super("UniDraw:  New File");
        dp = new DrawPanel();
        mbar = new JMenuBar();
        setJMenuBar(mbar);
         setJMenuBar(mbar);
        background = new Color(0xabcdef);
        foreground = new Color(0x001a57);
        width = 1;
        drawing = new ArrayList<Curve>();
    }

    public void run()
    {
        setSize(600,600);
        makeFileMenu();
        makeColorMenu();
        makeBackgroundMenu();
        makeWidthMenu();
        getContentPane().add(dp);
        setVisible(true);

    }
    public void makeFileMenu()
    {
        JMenu fileMenu = new JMenu("File");
        mbar.add(fileMenu);
    }
```

```
    public void makeColorMenu()
    {
        JMenu colorMenu = new JMenu("Color");
        mbar.add(colorMenu);
        colorMenu.add(new PenColorItem(Color.red, "red"));
        colorMenu.add(new PenColorItem(Color.orange, "orange"));
        colorMenu.add(new PenColorItem(Color.yellow, "yellow"));
        colorMenu.add(new PenColorItem(Color.green, "green"));
        colorMenu.add(new PenColorItem(Color.blue, "blue"));
        colorMenu.add(new PenColorItem(new Color(0x2E0854), "indigo"));
        colorMenu.add(new PenColorItem(new Color(0x7D26CD), "violet"));
        JMenuItem custom = new JMenuItem("Custom...");
        colorMenu.add(custom);
        custom.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                foreground = JColorChooser.showDialog(DrawFrame.this,
                       "Choose Background Color", foreground);
            }
        });
    }
    public void makeBackgroundMenu()
    {
        JMenu backgroundMenu = new JMenu("Background");
        mbar.add(backgroundMenu);
        backgroundMenu.add(new BackgroundMenuItem(Color.red, "red"));
        backgroundMenu.add(new BackgroundMenuItem(Color.orange, "orange"));
        backgroundMenu.add(new BackgroundMenuItem(Color.yellow, "yellow"));
        backgroundMenu.add(new BackgroundMenuItem(Color.green, "green"));
        backgroundMenu.add(new BackgroundMenuItem(Color.blue, "blue"));
        backgroundMenu.add(new BackgroundMenuItem(new Color(0x2E0854), "indigo"));
        backgroundMenu.add(new BackgroundMenuItem(new Color(0x7D26CD), "violet"));
        JMenuItem custom = new JMenuItem("Custom...");
        backgroundMenu.add(custom);
        custom.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                background = JColorChooser.showDialog(DrawFrame.this,
                       "Choose Background Color", background);
                repaint();
            }
        });
    }
    public void makeWidthMenu()
    {
        JMenu widthMenu = new JMenu("Width");
```

```
        mbar.add(widthMenu);
        widthMenu.add(new WidthMenuItem(1));
        widthMenu.add(new WidthMenuItem(2));
        widthMenu.add(new WidthMenuItem(5));
        widthMenu.add(new WidthMenuItem(10));
        widthMenu.add(new WidthMenuItem(20));
        widthMenu.add(new WidthMenuItem(50));
        JMenuItem custom = new JMenuItem("custom...");
        widthMenu.add(custom);
        custom.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                String  buf = JOptionPane.showInputDialog(DrawFrame.this,
                   "Specify width");
                try
                {
                    width = Integer.parseInt(buf);
                }
                catch(NumberFormatException ex)
                {
                    JOptionPane.showMessageDialog(DrawFrame.this,
                       "String \"" + buf + "\" is not an integer.");
                }

            }
        });

    }
    public static void main(String[] args)
    {
        DrawFrame df = new DrawFrame();
        javax.swing.SwingUtilities.invokeLater(df);
    }
    /*********************************************************************
    *
    *            Width Menu Items
    *
    *********************************************************************/
    class WidthMenuItem extends JMenuItem
    {
        final int width;
        public WidthMenuItem(int _width)
        {
            super("" + _width);
            width = _width;
            addActionListener(new ActionListener(){
```

```
            public void actionPerformed(ActionEvent e)
            {
                DrawFrame.this.width = width;
            }
        });
    }
}

/*****************************************************************
*
*            Color Menu Items
*
*****************************************************************/
public class PenColorItem extends JMenuItem
{
    private final Color color;
    public PenColorItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                foreground = color;
            }
        });
    }

}
class BackgroundMenuItem extends JMenuItem
{
    final Color color;
    public BackgroundMenuItem(Color _color, String colorName)
    {
        super(colorName);
        this.color = _color;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                background = color;
                dp.repaint();
            }
        });
    }
}
/*****************************************************************
```

```
*
*           The Draw Panel
*
*****************************************************************/
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    Curve c;            //now a state varible
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        c = null;  //initialize
    }
    //ignored mouse events
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
    //active mouse events
    public void mousePressed(MouseEvent e)
    {
        //remove "Curve" on this first line.
        c = new Curve(foreground, width);
        c.add(e.getPoint());
        drawing.add(c);
    }
    public void mouseReleased(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    public void mouseDragged(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(background);
        g.fillRect(0,0,getWidth(), getHeight());
        for(Curve c: drawing)
        {
            c.draw(g);
        }
```

```
        }
    }
}
```

Programming Exercises

1. Experiment with the other cap and join costants to observe the effects they cause on your drawing.

11.9 FileIO for Objects and Serialization

Now we well learn how to save our drawings in a file. This will involve several new classes, as well as the familar `File` class, which represents locations in our file system. We will now place two items in our `File` menu, `open` and `save`. Choosing the `open` menu item will open a drawing file and display it to the screen. Note that there are a lot of parallels to the `NitPad` case study here.

Let us begin by adding some features to our code. First of all, we will add a state variable to the `DrawFrame` class.

```
private File currentFile;
```

You also will need this import

```
import java.io.File;
```

In the constructor, do this

```
currentFile = new File("test.unid"); //TODO set this to null when menus work
```

We will begin by getting objects into and out of this fixed file `test.unid`. Later, we will get the full `File` menu working we will choose the current file from the file system using the `JFileChooser` widget.

FileIO will be controlled by listeners attached to items in the `File` menu. Let us begin with the process of saving a drawing.

This is done by creating an `ObjectOutputStream`. You can do this as follows.

```
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream(currentFile);
```

This process, if you check the constructor, can throw an `IOException`. Remember, we will adhere to the convention that we should handle any `FileNotFoundException` in our exception handling code.

You will need the following imports.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
```

What can be stored via an `ObjectOutputStream`? You can store any primitive type, or any object type that implements the interface `java.io.Serializable`. This makes sense, since the process of storing an object in a file in this manner is called *serialization*. The process reduces the object to its bytes and packs it in a non-human-readable binary format.Go to the API page and look up this interface. You will see that this is a "bundler interface" that has no required methods. If you attempt to serialize an object that does not implement this interface, you will be the unhappy recipient of a `NonserializableExcepton`. This is not a runtime exception, but it is a subclass of `IOException`. We can ferret these out by using the `printStackTrace()` method for `IOException`s.

Here we see the current state of `makeFileMenu`.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
}
```

Now let us add the save menu item and attach an action listener shell to it. You can put statemnt to print some string to `stdout` to test that it works. Run that, test it, and then discard it.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem saveItem = new JMenuItem("Save");
    fileMenu.add(saveItem);
    saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            saveWindowToCurrentFile();
        }
    });
}
```

Recalling what we know from `NitPad`, we know that there will be a need to write a file from several places in the menu. Therefore we will calll a helper

method, `private void saveWindowToCurrentFile()`. You should make an empty shell for this method and compile. If you run the program, you will see a save item in the `File` menu.

We will now take care of the implementing our helper method. To get started on this, we open an object output stream to the current file. We will also set up the exception-handling code.

```
private void saveWindowToCurrentFile()
{
    try
    {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(currentFile));
    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Now what do we save? Our rule is *Stuff the State!* Our state variables need to contain all information necessary to reconstruct the fine work of art we are creating. We are going to stuff the state of our app in the file. However, we do not need all of the elements to do this. For example we do not need the draw panel or the current file.

What methods are pertinent. You can write any primitive type. For example,

```
oos.writeInt(5);
```

will write the integer 5 to the object output stream. You can look in the API guide page to see all of the methods for writing primitive data. Any object you write must implement `Serializable`. If the object is a container, such as an `ArrayList<T>`, the type `T` of the entries, must also be serializable. Remember, if an ancestor class implements `Serializable`, you are in business.

We now look at all of our state variables.

```
private DrawPanel dp;
```

```
private JMenuBar mbar;
private Color background;//color of background
private Color foreground;//color of current curve
private int width;
private ArrayList<Curve> drawing;
private File currentFile;
```

We ask, "What is needed ot reconstruct our drawing?" We will not need the draw panel or the menu bar. We ditch them. We do need `foreground`, `background` and `width` to save the state of our session properly. Since `Color` implements `Serializable`, we are OK here. You can see this on the `java.awt.Color` page.

What abut the drawing? It is an `ArrayList` which is serializable. What about the entries? These are of type `Curve`, which is a child class of `ArrayList<Point>`. Finally, note that `java.awt.Point` implements `Serializable`, so we are in the clear. We can safely deflate all of our objects.

Here is our implementation.

```
private void saveWindowToCurrentFile()
{
    try
    {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(currentFile));
        oos.writeObject(background);
        oos.writeObject(foreground);
        oos.writeInt(width);
        oos.writeObject(drawing);
    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Now run the app, make a drawing and save it. Change colors, use curves of various widths and fool around. Remember its appearance, because will will resurrect it. Then look at the file's properties. In Linux, we see this

```
$ ls -l test.unid
```

```
-rw-rw-r-- 1 morrison morrison 774 Jan 22 19:59 test.unid
$
```

Our drawing occupies 774 bytes. If you open it with a text editor, you will see gibberish. So now let's see if we can resurrect our drawing. You can do this in Windoze by right clicking to see file properties.

Let us begin by creating an open item in the menu and attaching an action listener to it. We will create a helper method named `private void readFromCurrentFile()` to do the dirty work.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem saveItem = new JMenuItem("Save");
    fileMenu.add(saveItem);
    saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            saveWindowToCurrentFile();
        }
    });
    JMenuItem openItem = new JMenuItem("Open");
    fileMenu.add(openItem);
    openItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            readFromCurrentFile();
        }
    });
}
```

Now we are going to resurrect our drawing. You will not be surprised to learn we use an `ObjectInputStream` to do it. When you serialize, you have a responsibility. You must

- Retrieve the items you stored in the order in which you stored them
- Remember their types. All of this information is lost.
- If you used `writeObject`, you must use `readObject()` and *cast to the correct type*. If you used `writeInt`, you use `readInt()` to retrieve the integer. This same rule applies for all primitive types.
- Handle a `ClassCastException`. This is not a runtime exception but it is most likely a programmer error. Print a stack trace in this `catch` block to see where the error occurred.

Suitably warned, we sally forth with the customary insouciance.

```
private void readFromCurrentFile()
{
    try
    {
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(currentFile));
        background = (Color) ois.readObject();
        foreground = (Color) ois.readObject();
        width = ois.readInt();
        drawing = (ArrayList<Curve>) ois.readObject();
        ois.close();
        dp.repaint();

    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
    catch(ClassNotFoundException ex)
    {
        ex.printStackTrace();
    }
}
```

We have met our responsibilities. Now compile, select the open menu item and your drawing will reappear!

11.10 Making the `File` Menu

We need to design the file menu with care. Remember: *Data loss to the user is a cardinal sin!* We must be very careful to think through the user experience very carefully to get this complex process right. So let us skech out each item's action prior to coding and determine what helper methods are needed.

New If you have a drawing in the window already, we need to offer to save it. If the user says yes, save the file. In any event, blank the window and set everything to default.

open If you have a drawing in the window already, we need to offer to save it. If the user says yes, save the file. In any event, have the user select a file to open and then place it in the window.

Save If the current file is null, have the user choose a file. If the file is saved, do nothing. If not, write the contents of the window to the current file.

Save As... This will be a menu. It will offer two menu items: to save as a drawing, a `.jpeg` or a `.png`.

Save As → drawing Fire up a file chooser to choose where to save. Then, if the user opts to save, save it there.

Save As → JPEG Fire up a file chooser to choose where to save. Then, if the user opts to save it, save it as a `.jpeg`

Quit Offer to save any window conents, then quit. Cancel if the user declines when asked if he really wants to quit.

Now we examine these menu items. We should write helper methods that are atomic, i.e. these methods accomplish a single task.

Chapter 12

Collections: from the Inside

12.0 Data Strucures

Recall that a data structure is a container object that stores related objects under a single name. These structures differ in the way they access, manage, acquire, and present data. We have seen them before. Let's round up a few examples from the Python and Java worlds.

Python has these key data structures.

1. `list` This data structure is a mutable heterogeneous sequence type. It holds a sequence of objects of any type. It uses the [] operator for access to items and to sublists.
2. `tuple` This data structure is an immutable version of the `list` data structure.
3. `dict` This data structure creates an associative table of key-value pairs. The keys and values must be hashable objects in Python.

So far, we have seen these in Java.

1. `ArrayList<E>` This data structure is a mutable homogeneous sequence type. It holds a sequence of objects type `E`. It uses the get() method for access to items.
2. `arrays` These data structures are fixed-sized homogenous mutable containers. Access to entries is given by the [] operator. `list` data structure.

We are now going to focus on data structures in Java. We will learn about the Java Collections Framework. But first, we will learn about data structures from the inside by creating a simple data structure ourselves, the stack.

12.1 What is a Stack?

Think about a stack of books; we will be limited to interacting with and seeing the top book on the stack only. This idea should be familiar from your early encounters with functions. When a function in Python is called, its *stack frame* that contains its parameters and local variables is placed on the top of the call stack. When the function returns, its frame is removed from the top of the call stack and it is destroyed.

We are going to create a stack via two means. In one, we will use an array and in the other we will create a *linked data structure*. Both of these will grow dynamically to accomodate a large number of elements without the user having do deal with any resizing operation.

12.2 The Link Class

Our first version of a class will be based upon a objects called *links*; each link will contain a datum and a pointer to the next link. Our stack will have a bottom link (which will point at `null`). Each link will point down at the next lower item on the stack.

We begin by creating a shell for the class. Notice the use of the type parameter, since we are creating a generic class.

```
public class Stack<E>
{
}
```

We won't be able to do much until we build the link class. So we begin on that. Each link holds an element of type `E` and a pointer to the next link.

```
public class Stack<E>
{
    Link<E> top;
}
class Link<E>
{
    private E datum;
    private Link<E> next;
}
```

Now we produce two constructors for the link class to initialize the state variables.

```
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E _datum, Link<E> _next)
    {
        datum = _datum;
        next = _next;
    }
    public Link(E _datum)
    {
        this(_datum, null);
    }
}
```

Finally, we make getters and setters to manipulate the state of a link.

```
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E _datum, Link<E> _next)
    {
        datum = _datum;
        next = _next;
    }
    public Link(E _datum)
    {
        this(_datum, null);
    }
    public E getDatum()
    {
        return datum;
    }
    public Link<E> getNext()
    {
        return next;
    }
    public boolean hasNext()
    {
        return next != null;
    }
    public void setNext(Link<E> newNext)
    {
        next = newNext;
    }
```

```
    public void setDatum(E newDatum)
    {
        datum = newDatum;
    }
}
```

12.3 The Stack Interface

We will now determine what methods our stack data structure will have and how they will work. Both stack data structures we create will have exactly this interface.

- `boolean isEmpty()` This evaluates to `true` if the stack has no elements in it.
- `E peek()` This returns the top item on the stack. Notice, it returns the *datum*, not the link containing it. We throw an `EmptyStackException` if the client attempt to peek on an empty stack.
- `E pop()` This returns the top item on the stack and remove it. This entails getting rid of the link at the top. We throw an `EmptyStackException` if the client attempt to pop from an empty stack.
- `void push(T newItem)` This wraps the new item in a link and places it on the top of the stack.
- `void seymour()` This will print the stack from top to bottom.

Let us begin by putting in appropriate method stubs. Remember, we want our program to stay in an condition in which it compiles. By doing this, we flesh out our class's interface. Note the state variable `top` that will point to the top link on the stack.

```
public class Stack<E>
{
    Link<E> top;
    public Stack()
    {
    }
    public void push(E newItem)
    {
    }
    public E pop()
    {
        return null;
    }
```

```
    public E peek()
    {
        return null;
    }

    public boolean isEmpty()
    {
        return false;
    }
    public int size()
    {
        return -3;
    }
    public void seymour()
    {
    }
}
```

When we combine this and the link class in the file `Stack.java`, we have the shell of our program.

12.4 Implementing the Link-Based Stack

Now let us turn our attention to our skeleton stack class. It needs to know about the top of the stack, so we insert the state variable

```
    Link<E> top;
```

to point at the top of the stack. Let us agree that when the stack is empty, we will have `top` set to `null`. We also give our class an appropriate constructor.

```
    public Stack()
    {
        top = null;
    }
```

Our stacks will be born empty.

We now know when a stack is empty; this is so when `top == null`. We now implement the method `isEmpty()`

```
    public boolean isEmpty()
    {
        return top==null;
    }
```

Next, we will work on the `push` method. This method takes an object `newItem` of type `E`. Our stack does not accept bare items; these items must be clothed in a link. So we begin by doing that.

```
public void push(E newItem)
{
    Link<E> newLink = new Link<E>(newItem);
}
```

If the stack is empty, we just make `top` point at it and we are done. Adding that code, we have the following. Note that we blocked in the else block. Also observe that `newLink`'s `getNext()` returns `null`, indicating that it is at the bottom of the stack.

```
public void push(E newItem)
{
    Link<E> newLink = new Link<E>(newItem);
    if(isEmpty())
    {
        top = newLink;
    }
    else
    {
    }
}
```

Now we concern ourselves with adding to a nonempty stack. When we are done, `top` must point at the `newLink` and `newLink` must have as its next link the the rest of the stack.

As of now, `newLink`'s next link is `null`. We now do this

```
newLink.setNext(top);
```

Ths ensures that `newLink`'s next link is the existing stack. To finish, set

```
top = newLink;
```

and now `top` is pointing at the new link, which has the previous members of the stack as its successors. We now insert this code and we have `push`.

```
public void push(E newItem)
{
    Link<E> newLink = new Link<E>(newItem);
    if(isEmpty())
```

```
        {
            top = newLink;
        }
        else
        {
            newLink.setNext(top);
            top = newLink;
        }
    }
```

What is needed next is for us to be able to see that stuff is being pushed onto the stack. The need to implement `seymour` arises. Here is the idea. Start at the top. If it is null, we do nothing. Otherwise we get the datum, print it, and go to the next link. We stop when we encounter a `null` link. Notice that we make a copy of `top` to iterate with; we do not want `top` itself iterating to the bottom and losing our stack. This is an accessor method and should not change the state of the stack.

```
    public void seymour()
    {
        Link<E> foo = top;
        while(foo != null)
        {
            System.out.println(foo.getDatum());
            foo = foo.getNext();
        }
    }
```

Let us now create a driver class to test our `Stack` class. Create this program, `Driver.java`.

```
public class Driver
{
    public static void main(String [] args)
    {
        System.out.println("Stack Class Driver");
        Stack<String> s = new Stack<String>();
        s.push("a");
        s.push("b");
        s.push("c");
        s.push("d");
        s.seymour();
    }
}
```

You shoud see this result.

```
d
C
b
a
```

The items are "backwards" because a stack is a LIFO (last in first out) data structure.

Now let us peek. If a stack is empty, and you peek, this is likely a programmer goof. If a hapless client attepts to peek at an empty stack, we shall hand him an `EmptyStackException` for his troubles. Next time he will have the manners to us `isEmpty()` for its intended purpose. This exception we will be throwing will be a runtime exception. We need a new class. Create the file `EmptyStackException.java` and place this code in it.

```
public class EmptyStackException extends RuntimeException
{
}
```

We now begin our `peek` method.

```
    public E peek() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        return null;
    }
```

Peeking is easy. If the stack is not null, you just get the top's datum. We insert this in the `return` statement.

```
    public E peek() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        return top.getDatum();
    }
```

Once we get `peek`, it is staighforward to get `pop`. We will throw an excepton if we see an attempt to pop from an empty stack. Then we will save the top element, remove it, and then finally return it.

```
    public E pop() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        E out = top.getDatum();//fetch top item
        top = top.getNext();  //amputate
        return out;//return top item
    }
```

Now let us add a little torture test in a driver class to see if this works.

```
public class Driver
{
    public static void main(String[] args)
    {
        Stack<String> s = new Stack<String>();
        for(int k = 0; k < 10; k++)
        {
            s.push("" + k);
        }
        s.seymour();
        for(int k = 0; k < 10; k++)
        {
            s.pop();
        }
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

Now run.

```
morrison@odonata:~/book$ java Driver
9
8
7
6
5
4
3
2
1
0
PASS
```

Et Voila!

12.5 Iteraor and Iterable in the Link-Based Stack

Wouldn't it be cool if you could use a collections `for` loop on a stack? It would be helpful if we could create our own data structure and have a `for` loop walk through them cleanly. This feature exists for the standard `ArrayList`. Why not for our custom structures.

To this requires two simple steps. Your class must implement the interface `Iterable<E>`. We look in the API guide and we see that one method is reqired.

```
public Iterator<E> iterator();
```

Now we ask, "What is an iterator?" We look in the API guide and see that, it too, is an interface. This interface has three methods.

```
public Iterator<E>
{
    public E next();
    public boolean hasNext();
    public E remove();
}
```

It is an accepted standard that `remove()` is optional. We can render it unappetizing by throwing an `UnsupportedOperationException()`. That is the tack we shall take here.

To get our stack class ready we modify its header to read

```
public class Stack<E> implements Iterable<E>
```

We can keep it compiling by adding the iterator method as follows.

```
public iterator()
{
    return null;
}
```

We will implement our iterator as an inner class. We begin by making a hollow class with the required methods. While we are here, let us also adminisster the horse-kick if the hapless client calls `remove()`.

```
public class Navigator implements Iterator<E>
```

```
{
    private Link<E> nav;
    public Navigator()
    {
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
    public boolean hasNext()
    {
    }
    public E next()
    {
    }
}
```

Place this inner class inside of your Stack class and it will compile. You will need to add the import statement

```
import java.util.Iterator;
```

The job of `nav` is to navigate through our stack. It is easy to write `hasNext()`

```
    public boolean hasNext()
    {
        return nav != null;
    }
```

When writing `next()`, we must return the datum then move to the next node. It goes like this.

```
    public E next()
    {
        E out = nav.getDatum();
        nav = nav.getNext();
        return out;
    }
```

You now have your iteraor. Now return to the `iterator` method for the enclosing `Stack` class and you have

```
public Iterator<E> iterator()
{
    return new Navigator();
}
```

We now show the entire program; note that we added a recursive method for computing the size of the stack.

```
import java.util.Iterator;
public class Stack<E> implements Iterable<E>
{
    Link<E> top;
    public Stack()
    {
        top = null;
    }
    public void push(E newItem)
    {
        Link<E> newLink = new Link<E>(newItem);
        if(isEmpty())
        {
            top = newLink;
        }
        else
        {
            newLink.setNext(top);
            top = newLink;
        }
    }
    public E pop() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        E out = top.getDatum();//fetch top item
        top = top.getNext();  //amputate
        return out;//return top item
    }
    public E peek() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        return top.getDatum();
    }

    public boolean isEmpty()
    {
        return top==null;
```

```
}
public int size()
{
    return sizeHelper(top);
}
public int sizeHelper(Link<E> s)
{
    if(s==null)
        return 0;
    return 1 + sizeHelper(s.getNext());
}
public void seymour()
{
    Link<E> foo = top;
    while(foo != null)
    {
        System.out.println(foo.getDatum());
        foo = foo.getNext();
    }
}

/*************************Public Iteration**********************/
public Iterator<E> iterator()
{
    return new Navigator();
}
class Navigator implements Iterator<E>
{
    Link<E> nav;
    public Navigator()
    {
        nav = top;
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
    public boolean hasNext()
    {
        return nav != null;
    }
    public E next()
    {
        E out = nav.getDatum();
        nav = nav.getNext();
        return out;
```

```
        }
    }
}
```

We now have a full-featured link-based stack class. Now let's test the for loop

```
public class Driver
{
    public static void main(String[] args)
    {
        Stack<String> s = new Stack<String>();

        for(int k = 0; k < 10; k++)
        {
            s.push("" + k);
        }
        System.out.println("collections for loop");
        for(String q: s)
        {
            System.out.println(q);
        }
        System.out.println("Seymour");
        s.seymour();
        for(int k = 0; k < 10; k++)
        {
            s.pop();
        }
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

Running this we see

```
$ java Driver
collections for loop
9
8
7
6
5
4
3
2
1
0
```

```
Seymour
9
8
7
6
5
4
3
2
1
0
PASS
$
```

12.6 An Array Based Stack

Recall the interface we prescribed for a stack.

```
import java.util.Iterator;
public class AStack<E> implements Iterable<E>
{
    public AStack()
    {
    }
    public void push(E newItem)
    {
    }
    public E pop()
    {
        return null;
    }
    public E peek()
    {
        return null;
    }

    public boolean isEmpty()
    {
        return false;
    }
    public int size()
    {
        return -3;
    }
```

```
    public void seymour()
    {
    }
    public Iterator<E> iterator()
    {
        return null;
    }
}
```

We are going to create a stack class for this interface, but instead of using links, we will use an array to hold our elements. This presents a challenge. We must beware of overflowing the array and we must resize it when it gets nearly full. Along the way, we will have an encounter with the phenomenon of *type erasure*, which will render our challenge more difficult. Nevertheless, we shal circumvent this will a clever arabesque. Now strap on your seat belt.

Let us begin by creating two integer state variables. We will use `capacity` to refer to the size of the private array we shall create. We will create the private variable `size` to point just above the top of the stack. So, when the stack is empty, we make `size` 0. Now we have a some reasonable state variables, so we will create this and make the constructor.

```
    private int capacity;
    private int size;
    private E[] entries;
    public AStack(int _capacity)
    {
        capacity = _capacity;
        if(capacity < 1)
            throw new IllegalArgumentException();
        entries = new E[capacity];
        size = 0;
    }
    public AStack()
    {
        this(10);
    }
```

Notice we created two constructors. The default gives you an initial capacity of 10. This imitates the action of Java's array list type. The other allows you to specify a capacty of any positive integer size. We also burn the inept with an exception when they do something stupid, like make a zero or negative capacity.

Let us compile our masterpiece. We find ourselves on the business end of compiler ire.

```
$ javac AStack.java
```

```
AStack.java:13: error: generic array creation
        entries = new E[capacity];
                  ^
1 error
$
```

It is time for an arabesque. As it turns out, we cannot create arrays of generic type. We defer the disucssion of the reason until we discuss type erasure. If you can't bear the suspense, flip ahead and take a peek. Otherwise calmy accept the upcoming turn of events.

Insted of an `E()`, let us us an array of `Objects`. So we modify our code as follows.

```
private int capacity;
private int size;
private Object[] entries;
public AStack(int _capacity)
{
    capacity = _capacity;
    if(capacity < 1)
        throw new IllegalArgumentException();
    entries = new Object[capacity];
    size = 0;
}
public AStack()
{
    this(10);
}
```

All traces of compiler dyspepsia now vanish. We will have to enforce the type restriction on items being added to this array ourselves, but you will see that the interface will do this for us nicely. We go for the easy pickings first. When is the stack empty? When its size is zero. What is its size? We know that. So here we implement those two methods with one-liners.

```
public boolean isEmpty()
{
    return size == 0;
}
public int size()
{
    return size;
}
```

Now let us write `peek`. If the stack is empty, we throw an exception. Otherwise, we just look at the top item in the stack. Remember, `size` points just aboe the

stack, so note our use of `size - 1` to get it correct.

```
public E peek()
{
    if(isEmpty())
        throw new EmptyStackException();
    return (E) entries[size - 1];
}
```

Now compile. See the following squawling.

```
$ !javac
javac AStack.java
Note: AStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$
```

We are doing something kind of edgy but we know what we are doing, so we will add this annotation

```
@SuppressWarnings("unchecked")
```

above the `peek` method. The compiler will no longer grumble. This grumbling was triggered by the cast to `E` in the return statement.

Our push method will check the type of items placed on the stack. It will be the only entry point for new items to go onto the stack. As a result, we have a cast-iron guarantee that only items of typpe `E` will ever appear on our stack. That is what makes it safe to use the `@SuppressWarnings("unchecked")` annotation.

Now we shall pop. This requires us to keep track of the item being popped, then we can lower the size by 1. We could leave the popped item in the array, but we will set it to null to encourage garbage collection.

Next we will push new items onto the stack. The push method is the gatekeeper. We shall insist all items we push be type `E`. This is checked at compile time by the `push` method. If you attempt to push an item of the wrong type on the stack, the comipiler will issue forth with error messages and disapprobation.

So heere is how we will approach it. If the client attempts to pop from an empty stack, an exception will be thrown. We then make a record of the top item on the stack. Next, we will set the entry on the stack referring to the top item to `null` to encourage garbage collection. Finally, we decrement size and return the item we recorded. Here is the implementation

```
@SuppressWarnings("unchecked")
```

```
    public E pop()
    {
        if(isEmpty())
            throw new EmptyStackException();
        E out = (E) entries[size - 1];
        entries[size - 1] = null; //encourage garbage collection
        size--;
        return out;
    }
```

We need the annotation because of the cast. We are guaranteed, because of the contract we have for `push` that only items of type `E` will be placed on the stack.

Now let us implement `seymour`. That's easy.

```
    public void seymour()
    {
        int k;
        for(k = size - 1; k >= 0; k--)
        {
            System.out.println(k);
        }
    }
```

We now administer a torture test.

```
public class ADriver
{
    public static void main(String[] args)
    {

        AStack<String> s = new AStack<String>();
        for(int k = 0; k < 10; k++)
            s.push("" + k);
        s.seymour();
        for(int k = 0; k < 10; k++)
            s.pop();
        if(s.isEmpty())
            System.out.println("PASS");

    }
}
```

Here is the result.

```
$ java ADriver
```

```
9
8
7
6
5
4
3
2
1
0
PASS
```

Finally, we now write the iterator. It will be an inner class. We also modify the iterator method so it is not returning a null object. If you forget you will get a `NullPointerException` when you test the collections `for` loop.

```
public Iterator<E> iterator()
{
    return new Navigator();
}
/*******************iterator class*******************************/
class Navigator implements Iterator<E>
{
    int nav;
    public Navigator()
    {
        nav = size;
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
    public boolean hasNext()
    {
        return nav != 0;
    }
    @SuppressWarnings("unchecked")
    public E next()
    {
        nav--;
        return (E) entries[nav];
    }
}
```

We now show the class in its entirety.

```
import java.util.Iterator;
public class AStack<E> implemnets Iterable<E>
{

    private int capacity;
    private int size;
    private Object[] entries;
    public AStack(int _capacity)
    {
        capacity = _capacity;
        if(capacity < 1)
            throw new IllegalArgumentException();
        entries = new Object[capacity];
        size = 0;
    }
    public AStack()
    {
        this(10);
    }
    public void push(E newItem)
    {
        entries[size] = newItem;
        size++;
    }
    @SuppressWarnings("unchecked")
    public E pop()
    {
        if(isEmpty())
            throw new EmptyStackException();
        E out = (E) entries[size - 1];
        entries[size - 1] = null; //encourage garbage collection
        size--;
        return out;
    }
    @SuppressWarnings("unchecked")
    public E peek()
    {
        if(isEmpty())
            throw new EmptyStackException();
        return (E) entries[size - 1];
    }

    public boolean isEmpty()
    {
        return size == 0;
    }
```

```
    public int size()
    {
        return size;
    }
    public void seymour()
    {
        int k;
        for(k = size - 1; k >= 0; k--)
        {
            System.out.println(k);
        }
    }
    public Iterator<E> iterator()
    {
        return Navigator;
    }
}
```

Modify the driver by removing the call to `seymour` and using the collections `for` loop as follows.

```
public class ADriver
{
    public static void main(String[] args)
    {

        AStack<String> s = new AStack<String>();
        for(int k = 0; k < 10; k++)
            s.push("" + k);
        for(String q: s)
        {
            System.out.println(q);
        }
        for(int k = 0; k < 10; k++)
            s.pop();
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

This will work just as it did before we put the iterator in place.

12.7 Some Perspective

We have just implemented the same interface, that of a stack, in two very different ways. Each of these two methods has its strengths and weaknesses. In the next chapter we look at the performance of algorithms performed inside of data structures. By analyzing the performance characteristics of a particular implementatoin of an algorthm, we can understand when it is appropriate to choose a particular implementation of an interface.

Before we begin our analysis of algorithms in the next chapter we will explain some technical matters involved in generic programming. There are big benefits to this approach, but there are also hazards it is wise to be aware of.

12.8 A Roundup of Basic Facts about Generics

We have been be writing classes for varous data structures, which house collections of objects that are called *elements*. Each data structure has rules for item access and for adding items to the collection.

Recall that if we wanted an array list of strings we declared as follows.

```
ArrayList<String> roster = new ArrayList<String>();
```

Informally we would say that the object pointed at by `roster` is an "`ArrayList` of strings." The conents of `<..>` constitute the *type parameter* of the `ArrayList`. We can use any object type as a type parameter for an `ArrayList`. We can also use the wrapper types if we wish to have an array list of a primitive type.

The type `ArrayList` without a type parameter is called a *raw type*. When writing new code, you should not use raw types. We shall see that raw types can be used.

Let us proceed with an example. Consider this little program.

```
import java.util.ArrayList;

public class Blank
{
  public static void main(String [] args)
  {
    ArrayList foo = new ArrayList();
    foo.add("goo");
  }
}
```

Compilation of this code results in this compiler warning.

```
1 warning found:
File: /Users/morrison/book/javaCode/j10/Blank.java  [line: 8]
Warning: /Users/morrison/book/javaCode/j10/Blank.java:8: warning:
[unchecked] unchecked call to add(E) as a member of the raw type
java.util.ArrayList
```

Java wants to to specify a type parameter for your array list, hence this warning. Raw types work to ensure that Java is backwards-compatiable. Prior to Java 5, you never specified a type parameter; you only worked with raw types. Objects stored in an `ArrayList` were... just `Objects`.

Now watch what happens when we try to gain access to the contents of our `ArrayList`.

```
import java.util.ArrayList;

public class Blank
{
  public static void main(String [] args)
  {
    ArrayList foo = new ArrayList();
    foo.add("goo");
    System.out.printf("The length of %s is %d\n", foo.get(0),
        foo.get(0).length());

  }
}
```

We get a warning for using a raw type, and then we also get some angry yellow from the compiler.

```
1 error and 1 warning found:
-------------
*** Error ***
-------------
File: /Users/morrison/book/javaCode/j10/Blank.java  [line: 9]
Error: /Users/morrison/book/javaCode/j10/Blank.java:9:
cannot find symbol
symbol  : method length()
location: class java.lang.Object
-------------
** Warning **
-------------
File: /Users/morrison/book/javaCode/j10/Blank.java  [line: 8]
Warning: /Users/morrison/book/javaCode/j10/Blank.java:8: warning:
[unchecked] unchecked call to add(E) as a member of the raw type
```

```
java.util.ArrayList
```

Now you might ask why we are getting an error. The call `foo.get()` returns an `Object`, not a `String`. Therefore `foo.get().length()` makes no sense; you cannot compute the length of an `Object`. You can do this for a `String`. If you fetch an object from a container of raw type, you will just get an `Object`.

What is interesting is that the call to print `foo.get(0)` will compile and run (with the warning). Try it! This is because the `Object` class has a `toString()` method. The `Object` type object `foo.get()` points an a `String` with value `"goo"`. The varible sends the message to the `String` object and it prints itself.

The `Object` prints correctly because of the Delegation Principle. The type of a variable determines what methods are visible. The execution of those methods is delgated to the object. Since `"goo"` is a `String` object, it shows as a string and not as `Object@hexCrud`.

12.8.1 Type Erasure

Prior to Java 5, storage in collections such as the `ArrayList` worked a little like `ObjectOutputStream`s. The programmer could add object of any `Object` type to an `ArrayList`. Then, when fetching those values using the `get` method, the results had to be cast to the correct type to use them in their original form. In this system, the programmer is responsible for performing the correct casts.

We see this phenomenon at work in this example.

```
> import java.util.ArrayList;
> ArrayList foo = new ArrayList();
> foo.add("syzygy");
> foo.add("muffins");
> foo.get(0).length()
Static Error: No method in Object has name 'length'
> ((String) foo.get(0)).length()
6
> System.out.println(foo.get(0))
syzygy
> System.out.println(foo.get(1))
muffins
>
```

The cast is syntactically clumsy and it is aesthetically excrable. And it was the old way of doing business.

In modern parlance, you do this instead.

```
> import java.util.ArrayList;
```

```
> ArrayList<String> foo = new ArrayList<String>();
> foo.add("syzygy");
> foo.add("muffins");
> foo.get(0).length()
6
> foo.get(1).length()
7
>
```

That ugly cast is now gone. Where'd it go? This is no mere detail. It is important you understand what is happening behind the scenes, so you fully understand the benefits, quirks and possible dangers entailed with using generics.

Again, you should always use generics when writing new code. However, it is important to understand what is happening, because you will see legacy code that does not use generics, and you need to know how to read and manage it correctly.

So how does that all work? Suppose you use an `ArrayList` with a type parameter. You compile the program. When you do, any calls that add items to the `ArrayList` are checked to see if they are adding items of the correct type. The compiler enforces this contract.

At run time all of these have the same appearance

- ArrayList¡Integer¿
- ArrayList¡ArrayList¡Integer¿¿
- the raw type `ArrayList`

Any calls that get items from the `ArrayList` have the appropriate casts added to them *by the compiler*. You are given the *Cast Iron Guarantee*, which says that all of the necessary casts will be added by the compiler.

Danger! You can void the Cast Iron Guarantee: The existence of an "unchecked warning" voids the guarantee.

If you are using generic classes, you cannot have an "unchecked warning;" this undermines the safety of your entire program. It is the responsiblity of the creator of a generic class to extirpate any of these that might crop up.

When your code is in its run-time form, all evidence of generics is gone. All this code sees is the casts created by the compiler. Your code at runtime looks like pre-5 Java code, with raw types. This approach has two imporant benefits.

Pre-5 Java code will still compile nicely. Java maintains backwards compatibility.

This erasure mechanism helps keep your byte code small. All Java `ArrayList`s look exactly the same at run time, so there is only one segment of code for `ArrayList`.

In contrast, C++ has a similar-looking mechanism called *templates*. A close analog to a Java `ArrayList` is C++'s `vector` class. If you make vectors with several different type parameters, object code will be generated for each one. This phenomenon is sometimes known as "code bloat." This is the opposite approach to generic programming to the type erasure approach of Java.

Be warned that there will be some pitfalls for you when writing generic code, especially if you deal with arrays. We encountered this problem in in the creation of our array based stack class. Take a look at this little class. In it, we attempt to create an array of generic type.

```
public class UglySurprise<T>
{
  T[] myArray;
  public UglySurprise(int n)
  {
    myArray = new T[n];
  }
}
```

Now compile and you see an error. Java does not permit the creation of arrays of generic type.

```
1 error found:
File: /Users/morrison/book/javaCode/j10/UglySurprise.java  [line: 6]
Error: /Users/morrison/book/javaCode/j10/UglySurprise.java:6: i
generic array creation
```

The reason for this will be revealed when we discuss subtyping and generics.

12.9 Inheritance and Generics

We will use the term *type* to refer to a class or an interface. We say `S` is a *subtype* of `T` for any of these three situations.

- `S` and `T` are interfaces and `S` is a subinterface of `T`.
- `S` and `T` is a class implementing `S`.
- `S` is a subclass of `T`.

If `S` is a subtype of `T` we will say that `T` is a *supertype* of `S`. Let us begin by creating these classes and compiling.

```
public class General
{
}

public class Specific extends General
{
}

import java.util.ArrayList;

public class ArrayTest
{
  public static void main(String[] args)
  {
    Specific [] s = new Specific[10];
    General [] g = new General[10];
    ArrayList<Specific> as = new ArrayList<Specific>();
    ArrayList<General> ag = new ArrayList<General>();
    testArray(s);
    testArray(g);
    testArrayList(as);
    testArrayList(ag);

  }
  public static void testArray(General[] g)
  {
  }
  public static void testArrayList(ArrayList<General> ag)
  {
  }
}
```

In these classes, `Specific` is a subtype of `General`. All now compiles nicely. Now go into the `main` method of the `ArrayTest` class and add these four lines of code.

```
    Specific [] s = new Specific[10];
    General [] g = new General[10];
    testArray(s);
    testArray(g);
```

Your code will still compile. This is because array types are *covariant*, i.e., that if `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`. This seems intuitive and it does not come as any kind of terrible surprise.

Let us formulate analogous code for an `ArrayList`.

```
    ArrayList<Specific> as = new ArrayList<Specific>();
    ArrayList<General> ag = new ArrayList<General>();
    testArrayList(as);
    testArrayList(ag);
```

Now compile this. Here is the result.

```
1 error found:
File: /Users/morrison/book/javaCode/j10/ArrayTest.java  [line: 13]
Error: /Users/morrison/book/javaCode/j10/ArrayTest.java:13:
testArrayList(java.util.ArrayList<General>) in ArrayTest cannot be
applied to (java.util.ArrayList<Specific>)
```

Why the difference? This is because generics are *invariant*; i.e., if `S` is a subtype of `T`, then `ArrayList<S>` is *not* a subtype of `T` unless `S` and `T` are in fact the same type.

In `java.util` there is an interface called `List`. The standard `ArrayList` and `LinkedList` classes in `java.util` implement `List` so it is possible for `List` variables to point at `ArrayList` or `LinkedList` objects.

Let us learn what subtype relationships occur here. Modify `ArrayTest.java` as follows.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayTest
{
  public static void main(String[] args)
  {
    Specific [] s = new Specific[10];
    General [] g = new General[10];
    ArrayList<Specific> as = new ArrayList<Specific>();
    ArrayList<General> ag = new ArrayList<General>();


  }
  public static void testArray(General[] g)
  {
  }
  public static void testArrayList(ArrayList<General> ag)
  {
  }
}
```

We now perform an experiment by adding this code to the `main` method.

```
List<General> lg = new ArrayList<General>();
```

This does compile. If `Foo` is a subtype of `Goo` and these two classes are generic, then `Foo<T>` is a subtype of `Goo<T>`. Notice that this breaks things.

```
List<General> lgs = new ArrayList<Specific>();
```

We show the resulting compiler error.

```
1 error found:
File: /Users/morrison/book/javaCode/j10/ArrayTest.java  [line: 13]
Error: /Users/morrison/book/javaCode/j10/ArrayTest.java:13:
incompatible types
found   : java.util.ArrayList<Specific>
required: java.util.List<General>
```

Notice that a `ArrayList<General>` is a subtype of `List<General>`, but things break because of invariance. An `ArrayList<Specific>` is not a subtype of `ArrayList<General>`

12.10 Programming Project: A Linked List

In this project, you will write a simplifed version of Java's `LinkedList` class. The purpose of this project is to acquaint you further with link-bases structures and for you to get your hands dirty with them.

Here is the interface.

Header	Action
`LinkedList<E>()`	This constructor creates an empty linked list.
`boolean addLast(E newItem)`	This places the new item on the end of the list. Return true if the item is successfully added.
`boolean addFirst(E newItem)`	This places the new item at the beginning of the list. Return true if the item is successfully added
`boolean add(int index, E newItem)`	This inserts `newItem` into the list at the prescribed index. An `IndexOutOfBoundsException` is thrown if an illegal index is used.
`void clear()`	This empties the linked list of its items.
`boolean contains(Object o)`	This returns true if the object o is present in the list.
`int size()`	This returns the number of objects present in the list.
`E get(int n)`	This returns the object in position `n`. It throws an `IndexOutOfBoundsException` if you try to return an nonexistent element
`E set(int n, E newItem)`	This replaces the item at index `n` with item `newItem`. It throws an `IndexOutOfBoundsException` if you try to return an nonexistent element
`int indexOf(Object o)`	This returns the index of the first instance of `o` in the list, or -1 if the object does not appear in the list.
`int lastIndexOf(Object o)`	This returns the index of the last instance of `o` in the list, or -1 if the object does not appear in the list.
`boolean remove(Object o)`	This removes the object `o` from the list if it is present. It returns true if the list contains the object.

We provide you with a shell for the program.

```
import java.util.Iterator;
class LinkedList<E> implements Iterable<E>
{
    private Link<E> head;
    public LinkedList()
    {
        head = null;
    }
```

```
    public boolean isEmpty()
    {
        return head == null;
    }
    public void addFirst(E newItem)
    {
        //look at the link based stack
    }
    public void seymour()
    {
    }
    public boolean contains(Object o)
    {
    }
    /***********Iteraor related stuff
    public Iterator<E> iterator()
    {
        return null;
    }
    class Navigator implements Iterator<E>
    {
        //add the necessary methods
    }
}
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E _datum, Link<E> _next)
    {
        datum = _datum;
        next = _next;
    }
    public Link(E _datum)
    {
        this(_datum, null);
    }
    public E getDatum()
    {
        return datum;
    }
    public Link<E> getNext()
    {
        return next;
    }
    public boolean hasNext()
```

```
    {
        return next != null;
    }
    public void setNext(Link<E> newNext)
    {
        next = newNext;
    }
    public void setDatum(E newDatum)
    {
        datum = newDatum;
    }
}
```

12.11 Programming Project: An Array-Based List

Header	Action
`ArrayList<E>(int _capacity)`	This constructor creates an empty Array list with the specified capacity. An `IllegalArgumentException` is thrown if the client passes a nonpositive integer.
`ArrayList<E>()`	This constructor creates an empty Array list with a capacity of 10.
`boolean addLast(E newItem)`	This places the new item on the end of the list. Return true if the item is successfully added.
`boolean addFirst(E newItem)`	This places the new item at the beginning of the list. Return true if the item is successfully added
`boolean add(int index, E newItem)`	This inserts `newItem` into the list at the prescribed index. An `IndexOutOfBoundsException` is thrown if an illegal index is used.
`void clear()`	This empties the linked list of its items.
`boolean contains(Object o)`	This returns true if the object o is present in the list.
`int size()`	This returns the number of objects present in the list.
`E get(int n)`	This returns the object in position `n`. It throws an `IndexOutOfBoundsException` if you try to return an nonexistent element
`E set(int n, E newItem)`	This replaces the item at index `n` with item `newItem`. It throws an `IndexOutOfBoundsException` if you try to return an nonexistent element
`int indexOf(Object o)`	This returns the index of the first instance of `o` in the list, or -1 if the object does not appear in the list.
`int lastIndexOf(Object o)`	This returns the index of the last instance of `o` in the list, or -1 if the object does not appear in the list.
`boolean remove(Object o)`	This removes the object `o` from the list if it is present. It returns true if the list contains the object.

We provide you with a shell for the program.

```
import java.util.Iterator;
class ArrayList<E> implements Iterable<E>
{
    private int capacity;
    private int size;
    private Object[] entries;
    public ArrayList(int _capacity)
    {
        capacity = _capacity;
        size = 0;
        entries = new Object[capacity];
    }
    public ArrayList()
    {
        this(10);
    }
    public boolean isEmpty()
    {
    }
    public void addFirst(E newItem)
    {
        //look at the array based stack
    }
    public void seymour()
    {
    }
    public boolean contains(Object o)
    {
    }
    /***********Iteraor related stuff
    public Iterator<E> iterator()
    {
        return null;
    }
    class Navigator implements Iterator<E>
    {
        //add the necessary methods
    }
}
```