JAVA SUCCINCTLY PART 2

BY **CHRISTOPHER ROSE**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Java Succinctly Part 2

By
Christopher Rose

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: John Elderkin
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Christopher Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Majestic Theatre in Pomona, Queensland.

Introduction

This is the second e-book in the two-part series *Java Succinctly*. If you have not read the first book, *Java Succinctly Part 1*, and if you are not familiar with the basics of the Java language, I strongly recommend you read that e-book first. In this volume, we will concentrate on more advanced features of Java, including multithreading, building GUI applications, and 2-D graphics/game programming. Code samples can be found here.

Programming computers requires a lot of practice, which means we will inevitably make mistakes and unintentionally cause our programs to crash, hang, and otherwise behave in ways that would make an end user ill. I recommend that you copy and paste the code samples here to get an overview of how things work. Then, go ahead and changes things later—add new functionality and features to the programs (particularly the Calculator and Space Game applications presented in Chapters 7 and 8). This will give you good insight into a computer programmer's power. Remember that 100 years ago, there was nobody on Earth who could sensibly demand a billion computations be performed in one second, yet this is trivial for a modern computer programmer. We do it all the time!

Test your programs constantly, and use the debugging features of the IDE (assert, which we will look at in this book), variable watches, and breakpoints, etc. Save your projects very frequently, too.

Without further ado, let us explore some of the powerful and practical features of Java!

Chapter 1 Packages and Assert

Packages

Packages offer a way to organize classes into groups. They allow us to have multiple classes with the same name but that belong to different packages, and they allow us to reuse code. A package is like a folder on a computer. A folder can contain multiple files, just like a package contains multiple classes. And there can be two files with exactly the same name, so long as they are in different folders. Similarly, you can have two different items with exactly the same identifier in different packages. When we use the import keyword, we can specify the packages and classes to import from.

Adding a package manually

In order to place your class into a new package, you must use the package keyword, followed by the package name, as in Code Listing 1.0.

Code Listing 1.0: Package Keyword

```
package MainPackage;

public class MainClass {
	public static void main(String[] args) {
	System.out.println("This class belongs to the MainPackage package!");
	}
}
```

You will notice that the first line of this code is underlined in red in Eclipse. This is because at the moment, there is no package called **`MainPackage`**. We can add the main package in two ways. The first is by right-clicking the **src** folder in the Package Explorer, then selecting **New** and **Package** from the context menus. This will bring up a dialog box in which you can type a name for your new package.

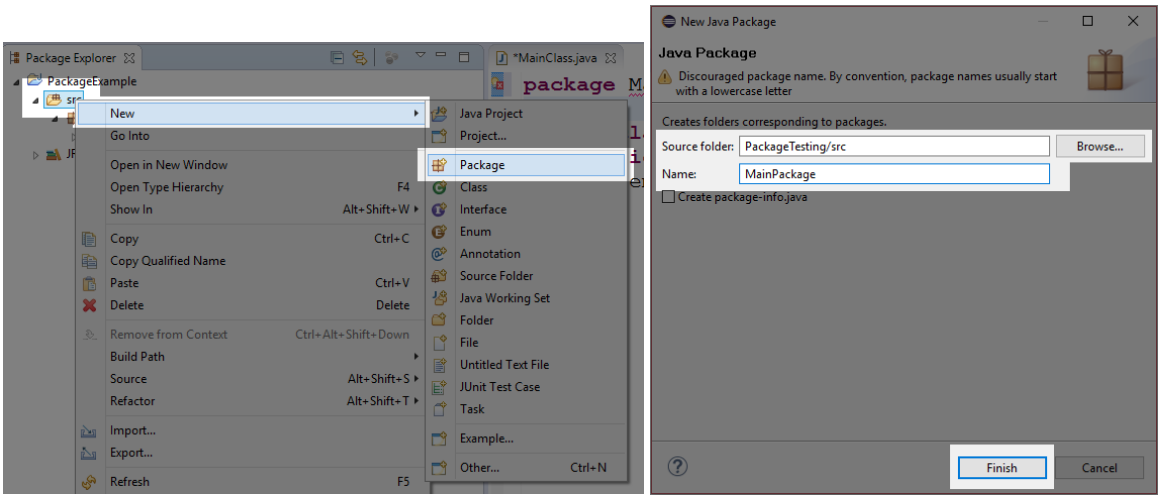


Figure 1: Adding a New Package

In the New Package box (the right window in Figure 1), you can specify a new folder for your package. I have left the folder as src. When you name the new package and click Finish, Eclipse will add the new package, but you will notice that **`MainClass`** is still not included in our new package, it remains part of the default package. You can drag the MainClass.java file into the **`MainPackage`** using the Package Explorer, as in Figure 2. When you move **`MainClass`** to the **`MainPackage`** package, you might be prompted to save the file first. When the move operation is complete, and if there are no classes in the default package, you will notice that the default package is removed from your project.

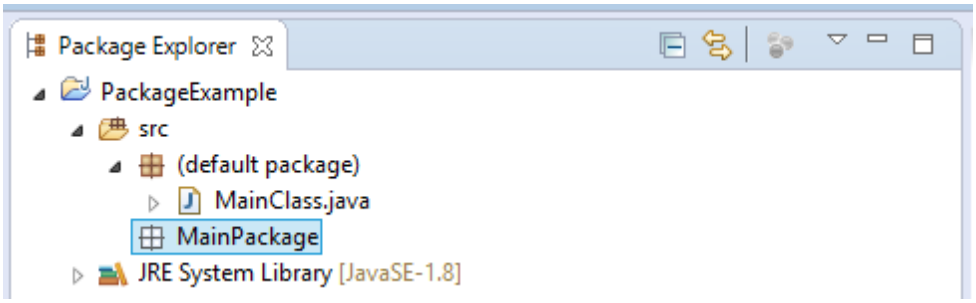


Figure 2: Drag MainClass.java into MainPackage

Adding packages using Eclipse suggestions

The second method for adding the package is to use Eclipse's built-in suggestion feature. Whenever there is an error or warning in our code, Eclipse will underline the suspect portion of code. We can hover our mouse cursor over the code, and Eclipse will pop up a box full of suggestions as to how to remedy the problem.

Note: In this e-book, I have used Eclipse as the IDE, but many other IDEs allow you to create Java applications. You might like to explore other IDEs, such as NetBeans, IntelliJ, and Android Studio (which is a version of Eclipse primarily designed to assist Android application development).

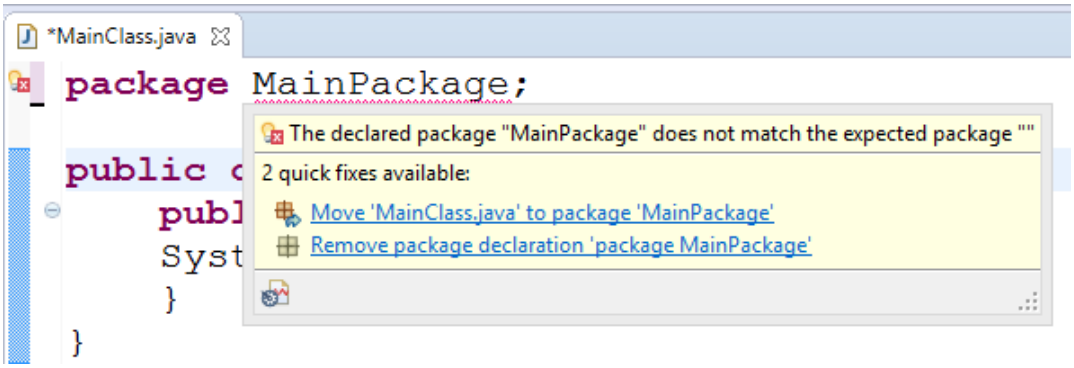


Figure 3: Eclipse's Suggestions to Add Package

Figure 3 shows two Eclipse suggestions for remedying the problem line "**`package MainPackage`**" when there is no **`MainPackage`**. The first suggestion is to move MainClass.java to package **`MainPackage`**. This is exactly what we want to do, so we click this suggestion and Eclipse will create the package in the Package Explorer, then move MainClass.java there for us.

Before we fix the problem, let's also note that there is a light bulb icon in the margin of the code window at the point where our **`MainPackage`** is underlined. You can click this icon to receive the same suggestions as you get by hovering with the mouse cursor.

Be sure to read Eclipse's suggestions very carefully—especially when dealing with potentially large-scale changes such as adding and removing classes from packages. If you are new to programming, good practice is to fix the problems manually before reverting to Eclipse's suggestions. Programming large-scale projects requires a degree of fluency that can only be obtained through practice.

Creating multiple packages

Let's add another class in a different package and see how we can import the second class into our **`MainClass`** by using the **`import`** keyword. Add a new class called **`OtherClass`**. Place **`OtherClass`** into a package called **`OtherPackage`** either using Eclipse's suggestion or by adding it to the Package Explorer, just as we did a moment ago with **`MainClass`**.

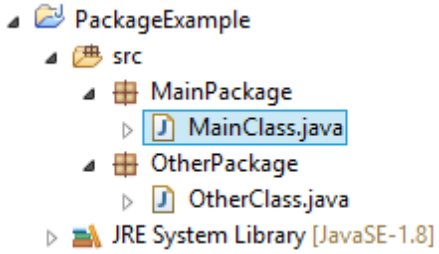


Figure 4: Two Packages

Figure 4 shows the two classes, each in a separate package. The code for the OtherClass.java file is listed in Code Listing 1.1.

Code Listing 1.1: The OtherClass

```
package OtherPackage;

public class OtherClass {
	public void SayHello() {
		System.out.println("No, say it yourself!");
	}
}
```

Code Listing 1.2: MainClass.java

```
package MainPackage;

// Import OtherPackage.*;
import OtherPackage.OtherClass;

public class MainClass {
	public static void main(String[] args) {
		OtherClass o = new OtherClass();
		o.SayHello();
	}
}
```

Code Listing 1.2 shows how we can import the package called **OtherPackage** into our MainClass.java file, create an object from the class, and call a **SayHello** method. The line that imports the package is "**import OtherPackage.OtherClass;**". We can also use the wildcard symbol, (*****), to import all classes defined as the **OtherPackage** package with the commented out line "**import OtherPackage.*;**".

We can write all the code for a program into a single package or never specify a package at all (this would mean all the classes in our project belong to the default package). But as projects become larger, we will typically collect algorithms and useful code that we can tie up in a package and reuse from project to project.

Assert

Assert is a useful debugging mechanism. To **assert** a condition in Java is to ensure that it is **true**. When we make an assertion, we are saying that if some statement is **false**, terminate the application and let us know (**assert** is meant for debugging, it is not designed for the end user or production code). We use **assert** to include tests in our program. If we are careful in designing the tests throughout our application's development, **assert** can let us know that something has gone wrong, and it can improve our ability to maintain and debug our projects. For the following, I have created a new project called **AssertTesting** and added a **MainClass**. The code for the new class is listed in Code Listing 1.3.

Code Listing 1.3: Using Assert

```
import java.util.Scanner;

public class MainClass {
	public static void main(String[] args) {
		int numerator;    // Numerator for our fraction.
		int denominator;// Denominator for our fraction.
		Scanner scanner = new Scanner(System.in);

		// Read a numerator.
		System.out.println("Please enter a numerator: ");
		numerator = Integer.parseInt(scanner.nextLine());

		// Read a denominator.
		System.out.println("Please enter a denominator: ");
		denominator = Integer.parseInt(scanner.nextLine());

		// Ensure that the denominator is not 0!
		assert(denominator != 0);

		// If the assert passed, print out some info using our
fraction:
		System.out.println(numerator + " / " + denominator + " = " +
				(numerator / denominator) + " remainder " +
				(numerator % denominator));
	}
}
```

The program in Code Listing 1.3 reads two integers from the user, a **numerator**, and a **denominator**. It is designed to divide the **numerator** by the **denominator** and output the results and remainder of the division. However, if the user inputs **0** as the **denominator**, the program cannot perform the division because division by **0** is not defined. The program uses "**assert(denominator != 0)**" to ensure that **denominator** is not zero.

Notice that the **assert** keyword has an associated **boolean** expression in brackets. If the **boolean** expression is **true**, the **assert** passes and the program continues execution normally. If the expression is **false** (i.e. the user typed **0** as the **denominator**), then the assertion failed and the program will exit. At least, that is the plan. By default, Eclipse is set to ignore assertions, and upon running the application and inputting a **denominator** of **0**, it will cause our program to crash. In order to run our application and have our assertions halt the program when they fail, we need to supply **-ea** as a command-line option to the JVM (**-ea** is short for enable assertions). In order to supply command-line arguments to the program, select **Run** from the file menu, followed by **Run Configurations**. This will open the Run Configurations dialog box, as shown in Figure 5.

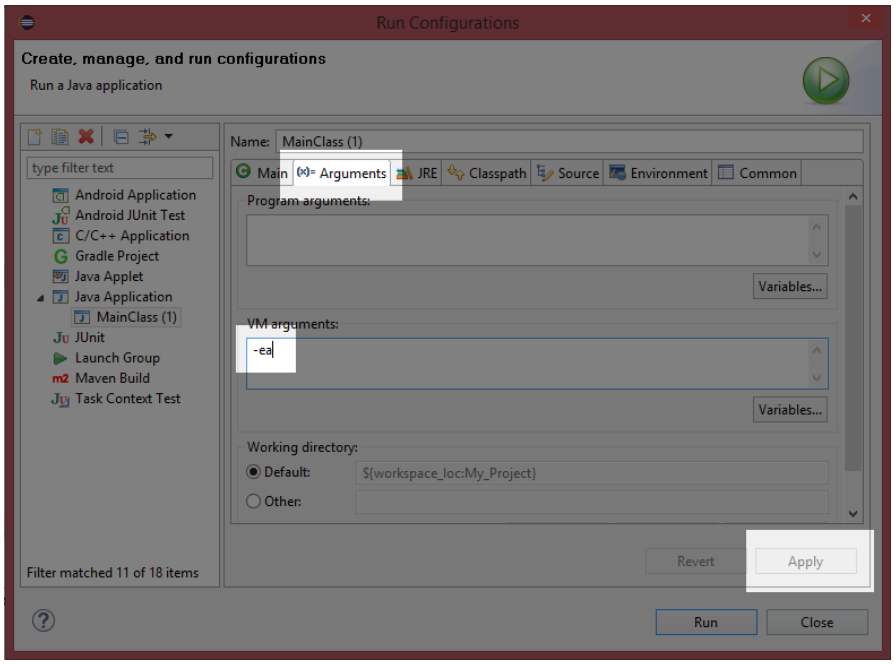


Figure 5: Specifying Command-Line Arguments to the JVM

In order to turn on assertions, select the **Arguments** tab and type **`-ea`** into the VM arguments box. Do not forget to click **Apply** after you do this. After the **`-ea`** argument is passed to the VM, we can debug our application again, and Eclipse will react more appropriately to our assertions. In order to switch the assertions off (to have the JVM ignore all assertions), remove the **`-ea`** argument from the arguments list as in Figure 5.

When an assertion fails in debugging mode, the program will pause on the assertion and highlight it so that the programmer can examine exactly what went wrong. When an assertion fails in run mode, the console window will show a message that points the programmer to the problem assertion, and to which file the assertion failed.

There are many command-line options available for the JVM and JVC. For more information on the available options, visit:

http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html

http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html#options

Chapter 2 Reading and Writing to Files

Reading and writing to files is important because files retain their information even when the computer is turned off. Files are slower to read and write than RAM (which is where variables and classes are generally stored when the program runs), but files are more permanent. Hard drives represent a memory space called nonvolatile. RAM, however, is volatile—it is cleared when the machine powers down. In addition, hard drives are generally much larger than RAM. In fact, many modern hard drives are terabytes in size, whereas the RAM in a desktop computer is often only a few gigabytes.

There are two broad file categories in Java. The difference is arbitrary, and in reality, there is no separation between the two categories, except in how we want to treat the data from the files in our code. Figure 6 shows a binary file on the left and a text file on the right. The two files have been opened in Programmer's Notepad, which is a plaintext editor (available from http://www.pnotepad.org/).

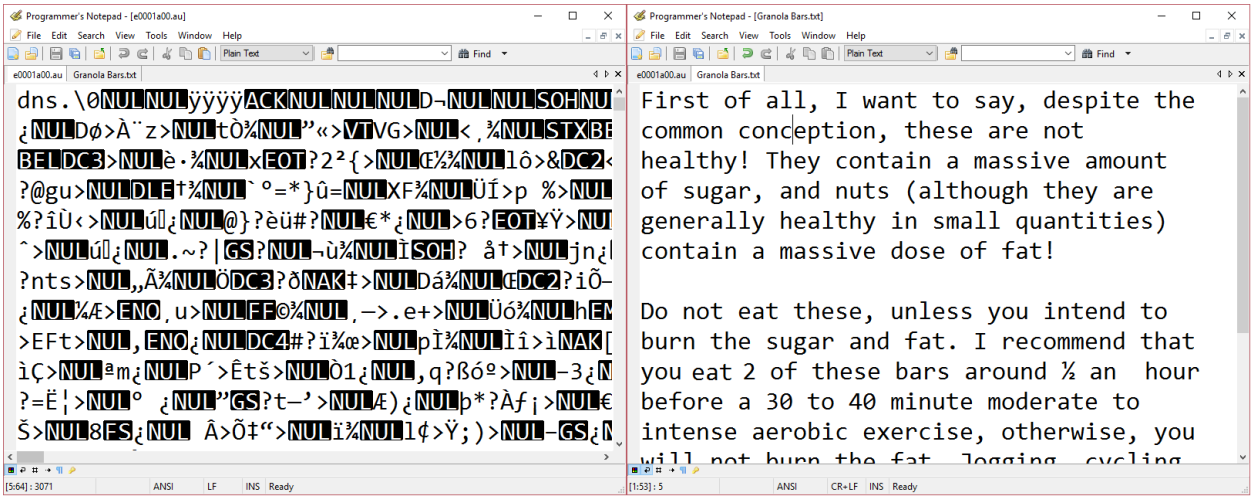


Figure 6: Binary File vs. Text File

In Figure 6, the binary file on the left is an audio file, and it looks like gibberish. It contains many strange characters and little or nothing is obviously human readable. Binary files represent data in a way that is easy for the computer to read—they are used to save information such as variables and objects in our programs. Common binary files are audio files such as WAV, MP3, etc., image files such as PNG or JPG, and files containing serialized versions of our objects (we will explore serialization in a moment).

On the right of Figure 6 is a text file. Text files consist mostly of human readable characters, such as letters of the alphabet, digits, and punctuation marks. The file on the right contains a description that can be easily read by a human but that a computer would not easily understand. Computers can read text files, but this often involves conversion, e.g., a computer can read the digits **128** from a text file but it must perform a conversion from the string **128** to the integer **128** before the number is readily useable.

In Java, when we open a file for reading or writing, we choose whether we want to open it as a text file or a binary file. The difference between the two is represented by the methods we can use to read and write data. In Java, reading and writing to text files is similar to reading from and writing to the console. Reading and writing to binary files is quite different.

Writing to a text file

Create a new project called **TextFiles**, then add a **MainClass** and a **main** method. Code Listing 2.0 shows some simple code for creating and writing a line of text to a text file.

Code Listing 2.0: Creating and Writing to a Text File

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class MainClass {
      public static void main(String[] args) throws FileNotFoundException {

            // Create a file:
            File myFile = new File("Example.txt");

            // Create a writer using the file.
            PrintWriter writer = new PrintWriter(myFile);

            // Write a line of text to the file.
            writer.println("This is some example text!");

            // Close the writer.
            writer.close();
      }
}
```

In Code Listing 2.0, the first line in the **main** method, "`File myFile = new File("Example.txt");`", creates a new **File** object called **myFile**. The **File** constructor takes a single parameter that is the path and name of the file: **Example.txt**. In our case, there is no path, so the program will create the file in the current folder (which will be the folder from which our application is running). Also, notice at the top of Code Listing 2.0, we import **java.io.File**.

The **File** object in Code Listing 2.0 is simply a filename reference. In order to write to the file, we must open it as text using a **PrintWriter**. The next line, "`PrintWriter writer = new PrintWriter(myFile);`", opens the file referenced by the **File** object for writing as text. Import **java.io.PrintWriter** in order to use the **PrintWriter** class.

If, for some reason, the file cannot be opened, the **`PrintWriter`** constructor will throw a **`FileNotFoundException`**. For this reason, I have imported **`java.io.FileNotFoundException`** and added a **`throws`** declaration to my **`main`** method.

The next line writes a line of text to our text file using the writer's **`println`** method. The **`println`** method takes a string as an argument and writes the characters to the file, appending a new line character to the end.

Finally, the writer is closed on the last line using **`writer.close()`**. We must make sure that we close every file we open with our applications because reading and writing to files with multiple programs at once is very difficult to coordinate, and often the operating system will not allow multiple programs to access a single file. Close your files in order to enable other programs (or other instances of your program) to access the file.

Upon running the application, it will appear as though nothing happened. But if you right-click on your project in the Package Explorer and select **Refresh**, as in Figure 7, you will notice that Eclipse now includes the file "Example.txt" in our package. Double-click on the file in the Package Explorer and you should see the text that we printed to the file.

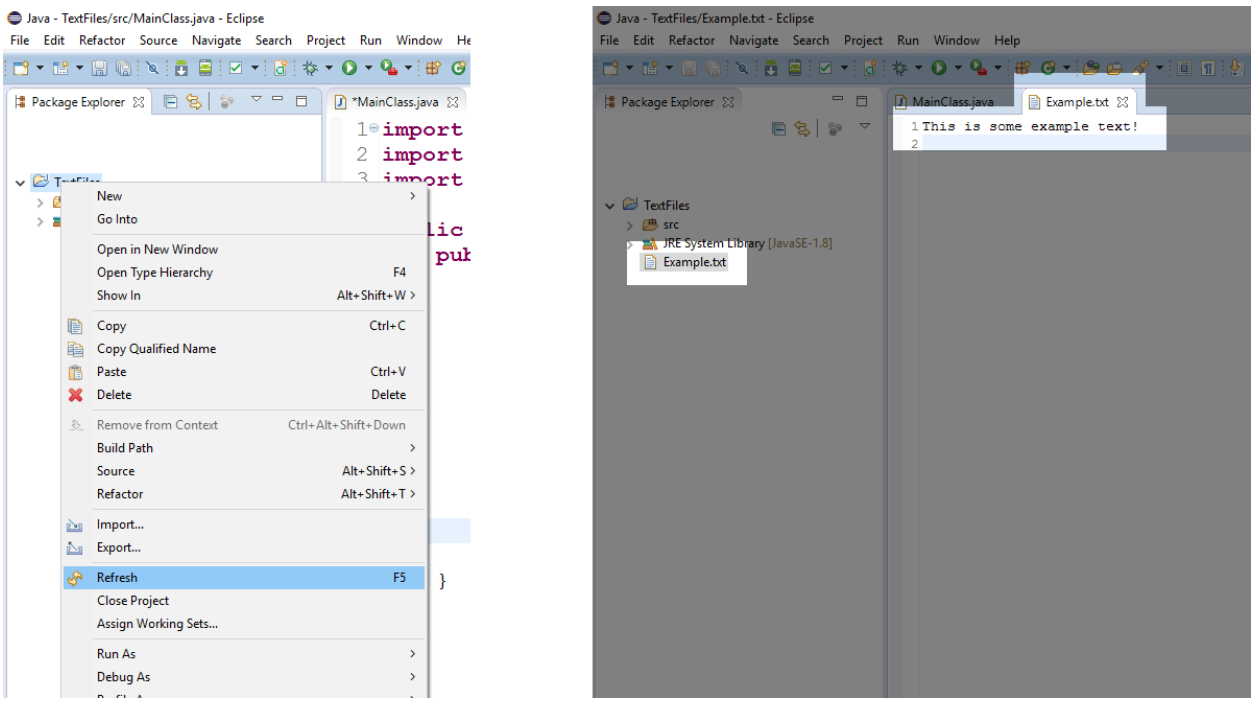


Figure 7: Refresh to Show the Example.txt File

We do not typically want to shut down our application when an error occurs, and instead of using a **`throws`** declaration for our **`main`** method, it is common to surround any code that deals with opening and saving files with try/catch blocks. Code Listing 2.1 shows the same program as above, except it uses a try/catch to respond to a **`FileNotFoundException`** more gracefully.

Code Listing 2.1: Surrounding File IO with Try/Catch

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class MainClass {
	public static void main(String[] args) {

		// Create a file.
		File myFile = new File("Example.txt");

		// Surround all file manipulation with try/catch.
		try {
			// Create a writer using the file.
			PrintWriter writer = new PrintWriter(myFile);

			// Write a line of text to the file.
			writer.println("This is some example text!");

			// Close the writer.
			writer.close();

		}
		catch (FileNotFoundException e) {
			// File could not be opened, show an error message.
			System.out.println("The file could not be opened.");

		}
	}
}
```

You might have noticed that each time you run the code from Code Listing 2.1, the data in the file is overwritten. We can also append new data to a file by using the **`FileWriter`** class and opening the file with the append parameter set to **`true`** (see Code Listing 2.2 for an example of appending text to a file). This is useful for logging purposes, when we do not want to overwrite the previously logged data each time the file is written to.

Code Listing 2.2: Appending Text to a File

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class MainClass {
	public static void main(String[] args) {
		try {
		// Create a file writer with the "append" parameter as "true":
```

```
                  FileWriter file = new FileWriter("Example.txt", true);

                  // Create a writer object from the file:
                  PrintWriter writer = new PrintWriter(file);

                  // Write some new text:
                  writer.println("This text will be added to the end!");

                  // Close the writer:
                  writer.close();
            }
            catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

We can write data and variables to a text file, but we must be aware that when we read the data, it must be parsed. Code Listing 2.3 shows an example of writing data to a text file, and we will see a much faster method for writing data in the section on serialization.

Code Listing 2.3: Writing Data/Variables to a Text File

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class MainClass {
      public static void main(String[] args) {
            File file = new File("test.txt");
            try {
                  PrintWriter out = new PrintWriter(file);
                  // Writing text
out.println("Be good. If you can't be good, be lucky!\n\t~ Alan Davis");

                  // Characters/floats/Boolean/doubles are all written in
                  // human readable form:
                  out.println( 129 ); // Integers
                  out.println( 2.7183f ); // Floats
                  out.println( true ); // Boolean
                  out.println( 1.618034 ); // Double
                  // Close writers after using them so they can be opened
by
                  // other programs:
                  out.close();
            }
            catch (FileNotFoundException e) {
                  System.out.println("File not found: " + e.getMessage());
```

```
            }
        }
}
```

Notice that upon running the application, when you refresh your project in the Package Explorer and open the file “test.txt”, the file will contain the following:

```
Be good. If you can't be good, be lucky!
      ~ Alan Davis
129
2.7183
true
1.618034
```

The numbers are human readable, i.e. the float **`2.7183f`** looks basically the same as it did in the code. This is very different from the way the computer actually stores a **`float`** in binary. Also note the use of “**`\n\t`**” in the quote from Alan Davis—this causes a new line and a tab character to be inserted into the file. These symbols are called escape sequences.

Tip: If you wish to find where the file is on your computer, navigate to your project's folder. You can navigate by right-clicking the project in the Package Explorer and selecting Properties. This will show the Project Properties dialog box. The project's folder is listed as its Location. Alternatively, you can right-click the file in the Package Explorer and select* Show in *and* System Explorer*. This will open the file’s location in the Windows Explorer.

Escape sequences

Before we go any further, let’s take a brief detour into escape sequences. When we print text to the screen, sometimes we need to use special symbols in order to add new lines, tabs, or characters that would otherwise end the string (i.e. printing the double-quote character: **`"`**). Escape sequences can be used anywhere in Java that writes strings to the screen or a file. This includes **`System.io`** and the **`PrintWriter.println`** method. Table 1 shows the escape sequences available in Java.

Note: Escape sequences are not inherent to strings. There is nothing about “\t” that makes the JVM print a tab character by itself. Escape sequences are a programmed behavior in some of the methods that deal with strings (such as `println`).

Table 1: Escape Sequences

Escape Sequence	Meaning
\t	Tab
\b	Backspace
\n	New Line
\r	Carriage Return

Escape Sequence	Meaning
\f	Form Feed
\'	Single Quote
\"	Double Quote
\\	Back Slash

Code Listing 2.4: Escape Sequence Examples

```
// \n causes a new line:
System.out.println("First line\nSecondline!");

// \t inserts a tab, i.e. a small block of whitespace.
System.out.println("This will be separated from\tThis with a tab!");

// Use \" to write " and \' to write '
System.out.println("Then Jenny said, \"It\'s above the fridge\".");

// To print a slash
System.out.println("\\ wears a top hat!");


// Some systems require \r\n in order to use a new line.
// Other systems will read this as two new lines, i.e. one
// carriage return and one new line, both of which look the same.
System.out.println("New\r\nLine!");
```

Code Listing 2.4 shows some examples of using escape sequences in our code. Note that at the end we use the pair "**\r\n**" for a single new line. The Eclipse console treats this as a new line, whereas it does not treat "**\n**" as a new line. This brings up the important point that reading escape sequences is program dependent. If we write "**\n**" to a text file, most text editors will read it as a new line. Some text editors allow us to specify whether "**\r\n**" or "**\r**" or "**\n**" should represent a new line.

Reading a text file

We can read from a text file by using a scanner. This is similar to reading from the console, except that instead of creating the scanner and passing the **System.in** parameter, we pass our file. Code Listing 2.5 shows an example of reading the text from Code Listing 2.3.

Code Listing 2.5: Reading from a Text File

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
```

```
public class MainClass {
	public static void main(String[] args) {
		File file = new File("test.txt");

		try {
			// Create a scanner from our file:
			Scanner in = new Scanner(file);

			// Read the first two lines into a string:
			String s = in.nextLine() + in.nextLine();

			// Reading variables:
			int i = in.nextInt();
			float f = in.nextFloat();
			boolean b = in.nextBoolean();
			double d = in.nextDouble();

			// Close the scanner:
			in.close();

			// Print out the results:
			System.out.println(
					"String: " + s + "\n" +
					"int: " + i + "\n" +
					"float: " + f + "\n" +
					"boolean: " + b + "\n" +
					"double: " + d );
			}
		catch (FileNotFoundException e) {
			e.printStackTrace();
		}
	}
}
```

Notice that the order in which we read the data must match the order in which we wrote it. When a numerical character is read from a file, it must match the data type or an exception will be thrown (for example, we cannot read “one” or “1” and expect that Java will automatically parse this text to the integer 1). Also, it is important to know that the reading and conversion of numerical data to numerical variables is very slow. We do not usually write variables in this manner, but instead tend to use text files mostly for reading and writing strings.

Serialization

We often want to save our objects to disk so that they can be restored later, after the machine has been switched off and on again. The act of converting an object into a format for saving to disk is called serialization. We could employ the preceding text reading/writing methods and

specify each member variable to save to a text file, but this technique is slow and requires us to specify each member variable to be saved in the classes, as well making sure to read the members in exactly the same order as we wrote them.

Instead of employing text files for our objects, we can serialize them and read/write to binary files. In order to allow our objects to be serializable, we must implement the **`Serializable`** interface. The interface requires the import of **`java.io.Serializable`**. Code Listing 2.6 shows a basic class that implements **`Serializable`**.

Serializing objects

There are many ways that objects can be saved to disk. When we serialize an object, we typically use an **`ObjectOutputStream`**, which is a class that takes an object and performs the conversion from the RAM representation of the object to the disk representation (i.e. serializes the object). Likewise, when we come to deserialize or read our objects back from the disk into our program, we usually use an **`ObjectInputStream`** in order to perform the conversion from the disk's representation of the object back to the RAM representation.

Code Listing 2.6: Implementing the Serializable Interface

```
import java.io.Serializable;

public class Animal implements Serializable {
	// Member variables
	float height;
	String name;
	boolean extinct;

	// Constructor
	public Animal(String name, float height, boolean extinct) {
		this.name = name;
		this.height = height;
		this.extinct = extinct;
	}

	// Output method
	public void print() {
		System.out.println("Name: " + name + "\n" +
			"Height: " + height + "\n" +
			"Extinct: " + extinct + "\n");
	}
}
```

In Code Listing 2.6, the only thing we must add to our class is the implement **`Serializable`** (we will look at interfaces and implements in more detail in the following chapter). Java takes care of the rest for us. Now that we have a serializable class, we need to create some objects, then

save them to disk. Code Listing 2.7 shows how to write an **`Animal`** object to disk using serialization.

Code Listing 2.7: Serializing Objects

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MainClass implements Serializable {
      public static void main(String[] args) throws FileNotFoundException,
IOException {

            // Create some animals from our Serializable class:
            Animal stego = new Animal("Stegosaurus", 12.5f, true);
            Animal croc = new Animal("Crocodile", 3.2f, false);
            Animal mozzie = new Animal("Mosquito", 0.2f, false);

            // Output to the console:
            stego.print();
            croc.print();
            mozzie.print();

            // Specify the name of our file:
            File file = new File("animals.dat");

            // Create a FileOutputStream for writing to the file.
            FileOutputStream fileOutput = new FileOutputStream(file);

            // Create object output stream to write serialized objects
            // to the file stream:
            ObjectOutputStream objectOutput = new
ObjectOutputStream(fileOutput);

            // Write our objects to the stream:
            objectOutput.writeObject(stego);
            objectOutput.writeObject(croc);
            objectOutput.writeObject(mozzie);

            // Close the streams:
            objectOutput.close();
            fileOutput.close();
      }
}
```

Code Listing 2.7 shows the steps to creating a serializable object from a class, then opening a file stream and an object stream.

If we run the program from Code Listing 2.7, then check the contents of the file (by refreshing the project in the Package Explorer, then double-clicking the file to open its contents), we will see that it no longer contains human readable data, but rather binary data (see Figure 8).

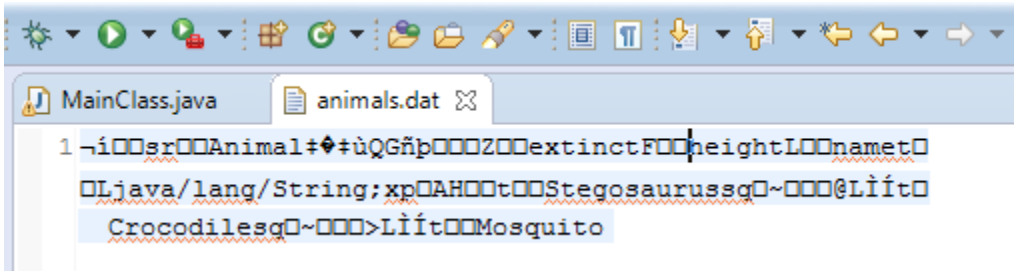


Figure 8: Serialized Objects

Figure 8 shows the contents of the file **animals.dat** after our three objects have been serialized. The contents of the file are not readable, and although there are a few scattered words, most of the file consists of nonsense characters (nonsense to humans, that is). This file presently contains data that is very fast and easy for the computer to read when we need to restore the exact values of our animals.

Reading serialized objects

Now that we have looked at how to serialize objects, let's look at how to read them back from the disk into RAM. Code Listing 2.8 shows an example of reading serialized objects from a file.

Code Listing 2.8: Reading Serialized Objects

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MainClass implements Serializable {
      public static void main(String[] args) throws FileNotFoundException,
IOException {

            // Create some animals from our Serializable class:
            Animal stego = new Animal("Stegosaurus", 12.5f, true);
            Animal croc = new Animal("Crocodile", 3.2f, false);
            Animal mozzie = new Animal("Mosquito", 0.2f, false);
```

```
            // Output to the console:
            stego.print();
            croc.print();
            mozzie.print();

            // Specify the name of our file:
            File file = new File("animals.dat");

            // Create a FileOutputStream for writing to the file.
            FileOutputStream fileOutput = new FileOutputStream(file);

            // Create object output stream to write the serialized objects
            // to the file stream:
            ObjectOutputStream objectOutput = new
ObjectOutputStream(fileOutput);

            // Write our objects to the stream:
            objectOutput.writeObject(stego);
            objectOutput.writeObject(croc);
            objectOutput.writeObject(mozzie);

            // Close the streams:
            objectOutput.close();
            fileOutput.close();


            ////////////////////////////////////////
            // Reading the objects back into RAM:
            ////////////////////////////////////////

            // Declare an array to hold the animals we read:
            Animal[] animals = new Animal[3];

            // Create a file and an object input stream:
            FileInputStream fileInput = new FileInputStream(file);
            ObjectInputStream objectInput = new
ObjectInputStream(fileInput);

            // Read the objects from the file:
            try {
                  animals[0] = (Animal) objectInput.readObject();
                  animals[1] = (Animal) objectInput.readObject();
                  animals[2] = (Animal) objectInput.readObject();

                  // Close the streams:
                  objectInput.close();
                  fileInput.close();
            }
```

```
            catch (ClassNotFoundException e) {
                  e.printStackTrace();
            }
            // Print the objects:
            System.out.println("Objects read from file: ");
            for(int i = 0; i < 3; i++) {
                  animals[i].print();
            }
      }
}
```

Code Listing 2.8 contains the code to serialize first, exactly the same as before. But the code highlighted in yellow shows how to deserialize the objects, then reads them back from disk into the array called **animals**.

Reading an unknown number of objects

If you do not know how many objects are serialized in a file, you can use a while loop without a terminating condition to read objects until an **EOFException** is thrown. **EOFException** stands for End-Of-File Exception. Code Listing 2.9 shows an example of reading the three **animals** into an **ArrayList** and catching the End-Of-File Exception. I've left out the code that serializes the three objects, but it would be exactly the same as Code Listing 2.7.

In Code Listing 2.9, we need to either catch or throw the **ClassNotFoundException**. If the file does not contain data that is serializable to our particular class, this exception will be thrown. We can either catch it or throw it. In Code Listing 2.9, I have dealt with the **ClassNotFoundException** by specifying that the **main** method throws it.

Code Listing 2.9: Reading an Unknown Number of Serialized Objects

```
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;

public class MainClass implements Serializable {
      public static void main(String[] args) throws FileNotFoundException,
IOException, ClassNotFoundException {
            // ...
            // The code above this line is the serializing code.
```

```
            // Deserializing an unknown number of objects:

            // Declare an array to hold the animals we read:
            ArrayList<Animal> animals = new ArrayList<Animal>();

            // Create a file and an object input stream:
            FileInputStream fileInput = new FileInputStream(file);

            ObjectInputStream objectInput = new
ObjectInputStream(fileInput);
            try {
                  // Read all the animals specified in the file,
                  // storing them in an array list:
                  for(;;) {
                        animals.add((Animal) objectInput.readObject());
                  }
            }
            catch (EOFException e) {
                  // We do not have to do anything here; this is the normal
                  // termination of the loop above when all objects have
                  // been read.
            }

            // Close the streams:
            objectInput.close();
            fileInput.close();
            for(Animal a: animals) {
                  a.print();
            }
      }
}
```

Chapter 3 Polymorphism

Polymorphism is a term that refers to code that might behave differently each time it is executed. There are many types of polymorphism in programming, but the term is often used to refer to a particular mechanism in object-oriented programming. Our objective is to define a parent class with some specific method or methods, then to define multiple child classes that inherit and define different code for the methods. Then, when we execute the method using the child classes, we can use the same code. However, the child classes will each perform their own specific versions of the methods.

In order to illustrate how the same code can behave differently, imagine we are creating a game (we will implement a game in the final chapter of this e-book). In a game, there is often a virtual world populated by objects, nonplayer characters (NPCs), and the player. Each object in the game is able to move, which means we can create a generic **`GameObject`** class with an abstract method called **`move`** (as per Code Listing 15).

Abstract classes

Code Listing 3.0: GameObject Class

```
// Abstract parent class:
public abstract class GameObject {
	// Abstract method:
	public abstract void move();
}
```

Notice that the **`GameObject`** class is marked with the **`abstract`** keyword. This is a safety measure—we do not want to create any instances from the generic **`GameObject`** class, so we mark it as **`abstract`** in order to prevent instances being created. Any class with one or more **`abstract`** methods must be marked as **`abstract`** itself. This class has the **`move`** method, which is **`abstract`**, so the entire class must be **`abstract`**. If you try to create an instance of the **`GameObject`** class, Eclipse will give you an error: *Cannot instantiate the type GameObject*. I call this a safety measure because we could define a body for **`move`** in **`GameObject`** and remove the **`abstract`** keywords altogether, but it may not be wise—in order for an object in our game to be useful, it must be of some specific type, not just a generic nameless “object.”

Notice also that the **`move`** method has no body. It consists of nothing more than a function declaration with a semicolon. We are saying that there exists another class, or classes, capable of performing the function **`move`**. We may want to refer to instances of these other classes as **`GameObject`** objects, but they must specify what the **`move`** method does or they will themselves be **`abstract`**.

Let us now define some classes that inherit from our **`GameObject`** class. When the NPCs move, we must execute different code when the player moves. NPCs are controlled by the computer,

and they typically employ some form of AI in order to talk to the player or wander around a town. The player, on the other hand, is not controlled by the computer. The player requires input from the user. So, we could create two derived classes from our **GameObject** class called **NPC** and **Player**, as per Code Listings 3.1 and 3.2.

Code Listing 3.1: NPC Class

```
public class NPC extends GameObject {
	public void move() {
		System.out.println(
			"The shopkeeper wanders around aimlessly...");
	}
}
```

Code Listing 3.2: Player Class

```
public class Player extends GameObject {
	public void move() {
		System.out.println("It is the player's move...");
		// Poll the keyboard or read the mouse movements, etc.
	}
}
```

In Code Listings 3.1 and 3.2, we have defined child classes that inherit from the **GameObject** class by using the **extends** keyword. The **extends** keyword means that all the member variables and the member methods in the **GameObject** class also belong to this class. What we have created is called a class hierarchy, which is something like a family tree. The parent class sits at the top and defines all the elements that the lower classes inherit (see Figure 9). The lower classes describe more specific items than does the parent. In our case, the parent describes a generic move method and the child classes define what it means.

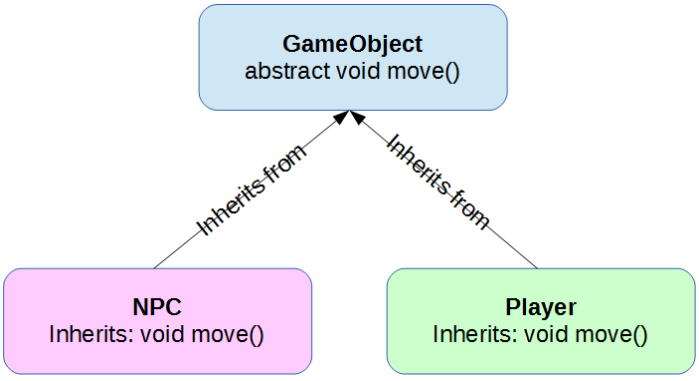


Figure 9: GameObject Class Hierarchy

We can now store all the objects in our game in a single collection, and we can iterate through the collection calling the **move** methods—both are advantages derived from the class hierarchy.

We can create instances of **NPC** and **Player** and store them in a collection of **GameObjects**. All **NPC**s are **GameObjects**, and all instances of the **Player** class are also **GameObjects**. Code Listing 3.3 shows an example **main** method that employs a collection of **GameObjects** but that uses polymorphism to call the two different versions of **move**.

Code Listing 3.3: Polymorphism

```
public class MainClass {
	public static void main(String[] args) {
		// Create 5 objects in our game.
		GameObject[] gameObjects = new GameObject[5];

		// First object is the player.
		gameObjects[0] = new Player();

		// Other objects are NPCs.
		for(int i = 1; i < 5; i++) {
			gameObjects[i] = new NPC();
		}

		// Call move for all objects in the game.
		for(int i = 0; i < 5; i++) {
			gameObjects[i].move();
		}
	}
}
```

The line highlighted in yellow in Code Listing 3.3 is an example of polymorphism. The first time the loop iterates, the local variable **i** will be set to **0**, and this line will cause the method **Player.move()** to be called because the first element of the **gameObject** array is of the **Player** type. But the other objects in the **gameObjects** array are all **NPC**s, which means the next iterations of this loop will call **NPC.move()**. The same line of code (i.e. "**gameObjects[i].move();**") is being used to call two different methods. We should understand that we did not create any instances from the **GameObject** class directly. We cannot do this because the **GameObject** class is **abstract**. We created instances of the **NPC** and **Player** classes, but then we used the generic term **GameObject** to store them and call their methods.

Upon running the application from Code Listing 3.3, the output is as follows:

```
It is the player's move...
The shopkeeper wanders around aimlessly...
The shopkeeper wanders around aimlessly...
The shopkeeper wanders around aimlessly...
The shopkeeper wanders around aimlessly...
```

Note: In Java, child classes can have only one parent class each. Some languages allow multiple parent classes, called multiple inheritance, but Java allows only one. However, Java does allow multiple interfaces to be implemented by a child class (see the next section).

Overriding methods

An abstract parent class can contain member methods and variables. In the previous examples, the **GameObject** class might contain **x** and **y** variables that specify where the object resides. We can define a method called print that outputs some information about the object (see Code Listing 3.4).

Code Listing 3.4: Nonabstract Parent Methods

```
// Abstract parent class:
public abstract class GameObject {
	// Member variables:
	int x, y;

	// Nonabstract method:
	public void print() {
		System.out.println("Position: " + x + ", " + y);
	}

	// Abstract method:
	public abstract void move();
}
```

Any child classes that inherit from the **GameObject** class in Code Listing 3.4 will automatically have the **x** and **y** variables of their parent class. They will also inherit the **print** method, which is not **abstract**. If we add a loop to our main method to call **print** with each of our five objects (Code Listing 3.4), they will each use the only version of the print method so far defined—the parent's **print** method.

Code Listing 3.5: Calling the Parent's Print Method

```
public class MainClass {
	public static void main(String[] args) {
		// Same code as before


		// Call print for all objects in the game.
		for(int i = 0; i < 5; i++) {
```

```
                gameObjects[i].print();
            }
        }
}
```

The output from Code Listing 3.5 is as follows:

```
Position: 0, 0
Position: 0, 0
Position: 0, 0
Position: 0, 0
Position: 0, 0
```

However, if we define a print method with the same signature as the parent's method in one of the child classes, we will see that the child classes can override the parent's method. Code Listing 3.6 shows the same code as the original **`Player`** class, except that this time I have overridden the parent's print method.

Code Listing 3.6: Overriding a Parent's Method

```
public class Player extends GameObject {
      public void move() {
            System.out.println("It is the player's move...");
            // Poll the keyboard or read the mouse movements, etc.
      }

      @Override
      public void print() {
            System.out.println("Player position: " + x + ", " + y);
      }
}
```

First, notice that the **`@Override`** annotation is optional. The print method in the **`Player`** class of Code Listing 3.6 has exactly the same name as the parent's print method and exactly the same arguments and return type. Now, when we run our **`main`** method, we will see that the **`Player`** object calls its own specific version of **`print`**, while the **`NPC`** objects (which do not define a specific version of the **`print`** function) call the parent's **`print`**. We say that the **`Player`** class has overridden the **`print`** method.

Constructors

Constructors are methods that are used to create new instances of objects. An **`abstract`** class can supply a constructor, even though we are not allowed to create instances of it. In Code Listing 3.7, the **`GameObject`** class has a constructor defined that sets the **`x`** and **`y`** values to -1.

Code Listing 3.7: Constructor in an Abstract Parent

```
// Abstract parent class:
public abstract class GameObject {
	// Member variables:
	int x, y;

	// Constructor
	public GameObject() {
		// Set the x and y:
		x = y = -1;
	}

	// Nonabstract method:
	public void print() {
		System.out.println("Position: " + x + ", " + y);
	}

	// Abstract method:
	public abstract void move();
}
```

If we change nothing else and run the program, we will see that the parent's constructor is called automatically for each of the child objects:

```
Player position: -1, -1
Position: -1, -1
Position: -1, -1
Position: -1, -1Position: -1, -1
```

However, we can also specify constructors for the child classes. Code Listing 3.8 shows the **Player** class with its own constructor.

Code Listing 3.8: Constructor for the Player Class

```
public class Player extends GameObject {

	// Constructor
	public Player() {
		x = y = 100;	// Start the player at 100x100.
	}

	public void move() {
```

```
            System.out.println("It is the player's move...");
            // Poll the keyboard or read the mouse movements, etc.
      }

      @Override
      public void print() {
            System.out.println("Player position: " + x + ", " + y);
      }
}
```

Running the application will show that the **Player** constructor is called to instantiate the **Player** object, and the **NPC**s all call the parent constructor because they do not define their own constructor.

Super keyword

When we need to refer to the parent class from within the child classes, we use the **super** keyword. As an example, Code Listing 3.9 shows how to call the **GameObject** constructor from within the **Player** constructor.

Code Listing 3.9: Child Constructor Calling Super Constructor

```
      // Constructor
      public Player() {
            // Call the parent's constructor.
            super();

            x = y = 100;      // Start the player at 100x100.
      }
```

When we call the parent's constructor using **super()**, it must be the first statement in the child's constructor. This is only true when calling the parent's constructor. If you want to call the parent's version of some other method, you can do so at any point in the child's overridden method.

instanceof keyword

Before we move on to interfaces, the **instanceof** keyword can be used to test the type of an object. The **main** method in Code Listing 3.10 uses the same class hierarchy as before.

Code Listing 3.10: Testing with instanceof

```
public class MainClass {
	public static void main(String[] args) {
		// Define some object:
		GameObject someObject = new Player();

		// Test if the first object is a GameObject.
		if(someObject instanceof GameObject)
			System.out.println("Object is a GameObject!");
		else
			System.out.println("Not a GameObject...");

		// Test if it is a Player.
		if(someObject instanceof Player)
			System.out.println("Object is a Player!");
		else
			System.out.println("Not a Player...");


		// Test if it is an NPC.
		if(someObject instanceof NPC)
			System.out.println("Object is a NPC!");
		else
			System.out.println("Not an NPC...");
	}
}
```

In Code Listing 3.10, we create a **Player** object called **someObject**. Then we use **instanceof** to test if the type is **GameObject**, **Player**, and **NPC**. Note that the data type of an object can be more than one thing. The output from the preceding **main** method shows that **someObject** is both a **Player** object and a **GameObject**. However, it is not an NPC:

```
Object is a GameObject!
Object is a Player!
Not an NPC...
```

Interfaces

Abstract methods are something like a contract. We say that any class that derives from a parent is capable of performing the abstract methods it inherits. Interfaces take abstract methods to the extreme.

An interface is similar to an abstract parent class, except that it will contain nothing but abstract methods (i.e. there are no methods specified at all in an interface, only method names). Interfaces do not specify member variables (though they can specify static members or class variables). When we derive a class from an interface, we are saying that the derived class must perform the set of methods specified in the interface (or the derived class must itself be an interface or abstract class).

Interfaces often describe some very general aspect of a class hierarchy. Often, interfaces are introduced as being some ultra-abstract version of a class. But there is a subtle difference between the way that an abstract class is typically used and the way an interface is typically used. Interfaces often describe that some particular activity can be performed using the instances of a class rather than describing that the instances can be used to perform some task. For example, an interface might describe objects that are sortable. Many types of objects are sortable—names can be sorted alphabetically, playing cards, and numbers, for example. But while these objects are sortable, the exact mechanism for comparing each is different. We could implement an interface called Comparable, meaning that any two objects that derive from the interface can be compared and thus a list of them might be sorted. Comparing and sorting objects is very common, and Java includes an interface called Comparable already.

Note: In Java, it is not possible to inherit from multiple parent classes. However, it is perfectly legal to inherit from multiple interfaces.

Begin a new project called Interfaces and add the **`Point`** class in Code Listing 3.11.

Code Listing 3.11: The Point Class

```
public class Point implements Comparable {
	// Member variables:
	public double x, y;

	// Constructor
	public Point(double x, double y) {
		this.x = x;
		this.y = y;
	}

	// Print out some info about the point:
	public void print() {
		System.out.println("X: " + x + " Y: " + y + " Mag: " +
				Math.sqrt(x*x+y*y));
	}

	public int compareTo(Object o) {
		// Firstly, if the second object is not a point:
		if(!(o instanceof Point))
		 return 0;
```

```
            // Cast the other point:
            Point otherPoint = (Point) o;

            // Compute the absolute magnitude of each point from the
origin:
            Double thisAbsMag = Math.sqrt(x * x + y * y);
            Double otherPointAbsMag =Math.sqrt(otherPoint.x * otherPoint.x
+
                        otherPoint.y * otherPoint.y);

            return thisAbsMag.compareTo(otherPointAbsMag);


            /*
            // Note: Double.compareTo does something like the following:

                  // If this object has a greater magnitude:
                  if(thisAbsMag > otherPointAbsMag) return 1;

                  // If this object a smaller magnitude:
                  if(thisAbsMag < otherPointAbsMag) return -1;

                  // If the object's magnitudes are equal:
                  return 0;
            */
      }
}
```

Notice the keyword **`implements`** on the first line of Code Listing 3.11, which is followed by the interface **`Comparable`**. This is how we inherit the methods from an interface. We do not use the term **`extends`** as we did with classes. The **`Comparable`** interface defines a single method that has the signature "**`public int compareTo(Object o)`**". Therefore, in order to implement the **`Comparable`** interface, we must supply this method in our class.

When we supply the **`compareTo`** method in our class, we have to understand the meaning of the parameters and the output. The method takes a single parameter that is presumably the same data type as the class we are defining (it does not make sense to compare points to playing cards, etc.; we are only interested in sorting points here). I have first supplied a test inside the **`compareTo`** method in order to make sure that the object we are comparing is actually a **`Point`**. If the object is not a **`Point`**, we could throw an exception, but I have returned **`0`** in Code Listing 3.11, which means the two objects are equal.

Next, we need to specify exactly what it means to compare our objects. If **`this`** object is less than **`o`**, we return **`-1`**. If **`this`** object is greater than **`o`**, we return **`1`**, and if the objects are the

same, we return **0**. For each class that implements the **Comparable** interface, we need to define what it means for the instances to be greater and less than each other. I have selected the meaning to be based on the absolute magnitude of the points (i.e. the distance from 0 on a 2-D plane, which is the square root of (x*x+y*y)). You will notice that I used the boxed version, **Double**, because the native **double** type does not implement the **Comparable** interface, while the boxed version, **Double**, does. After we have computed the distance between **this** and **o**, we call **Double.compareTo** and return the result. I also included a comment at the end of the code that shows roughly how the **Double.compareTo** method will behave.

Now that we have a class that implements the **Comparable** interface, we can create a collection of instances from our **Point** class and sort them using the standard Java sorting. Next, let's create a new class called **MainClass**. The code for this class is presented in Code Listing 3.12.

Code Listing 3.12: Sorting a List of Comparable Objects

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

public class MainClass {
	public static void main(String[] args) {
		// The total number of points in the demo:
		int numberOfPoints = 5;

		// Create a list of random points:
		ArrayList<Point> points = new ArrayList<Point>();
		for(int i = 0; i < numberOfPoints; i++)
			points.add(new Point(Math.random() * 100,
Math.random() * 100));

		// Print the unsorted points:
		System.out.println("Unsorted: ");
		for(int i = 0; i < numberOfPoints; i++)
			points.get(i).print();


		// Sorting a collection of Comparable objects:
		Collections.sort(points);

		// Print the sorted points:
		System.out.println("Sorted: ");
		for(int i = 0; i < numberOfPoints; i++)
			points.get(i).print();
```

```
            // Sort the items in reverse order (from largest to smallest):
            points.sort(Collections.reverseOrder());

            // Print the points sorted in reverse:
            System.out.println("Sorted in Reverse: ");
            for(int i = 0; i < numberOfPoints; i++)
                  points.get(i).print();
      }
}
```

In Code Listing 3.12, we use **`Collections.sort`** and **`points.sort(Collections.reverseOrder)`** in order to sort the points and also sort them in reverse order. These sorting methods are designed for use with any objects that implement the **`Comparable`** interface. This means we do not have to write a QuickSort (or some other algorithm) and mess around with swapping elements in arrays and comparing them efficiently in an **`ArrayList`**. Instead, all we need to do is to implement the **`Comparable`** interface and any list of our objects can be sorted!

Chapter 4 Anonymous Classes

Anonymous functions and classes appear in the code itself. They are not declared external to the body of the calling function, but instead are placed inline in the code, and they have no name (hence the term anonymous). They are often used to define a functionality or a class that is only required once at a point in the code. In order to use an anonymous class, we must implement an interface or extend an existing class. An anonymous class is a child class; it is an unnamed derived class that implements or extends the functionality of a parent class. We will see extensive use of anonymous classes in the event handling of the calculator application in Chapter 7.

Code Listing 4.0: Anonymous Inner Class vs. Class Instance

```
public class MainClass {
	// Parent class
	static class OutputLyrics {
		public void output() {
			System.out.println("No lyrics supplied...");
		}
	}

	public static void main(String[] args) {

		// Create a normal instance from the OutputLyrics class.
		OutputLyrics regularInstance = new OutputLyrics();

		// Anonymous Inner Class
		OutputLyrics anonymousClass = new OutputLyrics() {
			public void output() {
				System.out.println(
					"Desmond has a barrow in the market place.");
			}
		};

		// Call the output methods:
		regularInstance.output();

		// And using the anonymous class:
		anonymousClass.output();
	}
}
```

Code Listing 4.0 shows a basic example of an anonymous inner class. First, we define a parent class, which is called **OutputLyrics**. The class contains a single method that prints a string of text to the screen called **output**. Inside the **main** method, we create an instance of the **OutputLyrics** class. Note that the section marked with the comment "**// Anonymous Inner Class**" in the next line is important—we define and declare a new class that derives from the **OutputLyrics** class and that defines its own **output** method. Notice that we are creating an instance from a class that derives from **OutputLyrics**. The instance is called **anonymousClass**, but the class itself has no name. The syntax for an anonymous class is the same as the syntax for creating an instance from an existing parent class, except that the definition is followed immediately by a code block that defines the child class. Code Listing 4.1 shows the important lines from Code Listing 4.0.

Code Listing 4.1: Anonymous Class

```
// Anonymous Inner Class
OutputLyrics anonymousClass = new OutputLyrics() {
      public void output() {
            System.out.println(
                  "Desmond has a barrow in the market place.");
      }
};
```

Notice the first line in Code Listing 4.1 does not end with a semicolon, as a typical object definition would. Instead, we open a code block and override the output method. Declaring an anonymous inner class in this manner is a statement, and the semicolon actually comes at the end, after the code block is closed.

When we call the output method of our **regularInstance**, the program will print "**No lyrics supplied…**" to the output. This is the normal behavior of an **OutputLyrics** object. However, when we call the output method of our anonymous class, it will output the lyrics "**Desmond has a barrow in the market place.**"

Using an anonymous class as a parameter

The example in Code Listing 4.0 was trivial—it showed the basic syntax for using an anonymous class, but it is not a good example of why we would use this mechanism. Anonymous classes are often used when we need to pass functionality as a parameter to a method. For instance, if we know that we want to perform some operation between two integers and return some result, we could use an anonymous class that derives from the operation class, as per Code Listing 4.2.

Code Listing 4.2: Using Anonymous Inner Classes as Parameters

```
public class MainClass {
```

```
	// Parent class
	static class MathOperation {
		public int operation(int a, int b) {
			return 0;
		}
	}

	// Method that takes an object of MathOperation as a parameter.
	static int performOperation(int a, int b, MathOperation op) {
		return op.operation(a, b);
	}

	public static void main(String[] args) {
		// Some variables
		int x = 100;
		int y = 97;

		// Call the PerformOperation function with addition:
		int resultOfAddition = performOperation(x, y,
				// Anonymous inner class used as a parameter.
				new MathOperation() {
			public int operation(int a, int b) {
				return a + b;
			}
		});

		// Call the PerformOperation function with subtraction:
		int resultOfSubtraction = performOperation(x, y,
				// Anonymous inner class used as a parameter.
				new MathOperation() {
			public int operation(int a, int b) {
				return a - b;
			}
		});

		// Output Addition: 197
		System.out.println("Addition: " + resultOfAddition);

		// Output Subtraction: 3
		System.out.println("Subtraction: " + resultOfSubtraction);
	}
}
```

In Code Listing 4.2, we create a parent class called **MathOperation**. The class has a single method that takes two **int** parameters and returns some result. We also define a **static** method called **performOperation** that takes two **int** parameters and an instance of the **MathOperation**. The fact that the **performOperation** method takes a **MathOperation** as a parameter is the main concept in this illustration.

In the **main** method, we create two variables—**resultOfAddition** and **resultOfSubtraction**. The variables are defined as being the result from a call to **performOperation**, and two integer parameters, **x** and **y**, are passed. However, the crucial part is the third parameter to these calls to **performOperation** (highlighted in yellow in Code Listing 4.3).

Code Listing 4.3: MathOperation Anonymous Class

```
int resultOfAddition = performOperation(x, y,
                  // Anonymous inner class used as a parameter.
                  new MathOperation() {
                  public int operation(int a, int b) {
                        return a + b;
                  }
            });
```

The third parameter is an anonymous inner class. Instead of passing an instance of the **MathOperation** class, we derive and define an instance of an anonymous class. We are passing an instance of the anonymous child class to the **performOperation** method as a parameter. This child class has no name, and the instance of it has no name, either. We can still pass it as a parameter to the **performOperation** function. Inside the **performOperation** method, the instance is called **op**, but the caller does not need to create or name the instance—it is created and passed as a parameter when and where it is needed.

Anonymous classes and interfaces

The previous examples used a class as the parent for our anonymous inner classes. However, the parent class is often abstract or an interface. In Code Listing 4.3, when we use the child classes to call a single method, called operation, what are really doing is passing functionality to the **performOperation** class. In our example, we defined the **MathOperation** class as a normal class, but it might be more useful to define it as an interface (or perhaps an abstract class). The class has only a single method, and it makes little sense to perform an operation when we do not know what the operation is. Code Listing 4.4 shows the same example, except that the **MathOperation** class has been declared as an interface rather than a class.

Code Listing 4.4: Using an Interface as the Parent Class

```
public class MainClass {

      // Parent interface
```

```
	interface MathOperation {
		public int operation(int a, int b);
	}

	// Method that takes an object of MathOperation as a parameter.
	static int performOperation(int a, int b, MathOperation op) {
		return op.operation(a, b);
	}

	public static void main(String[] args) {
		// Some variables
		int x = 100;
		int y = 97;

		// Call the PerformOperation function with addition:
		int resultOfAddition = performOperation(x, y,
				// Anonymous inner class used as a parameter.
				new MathOperation() {
			public int operation(int a, int b) {
				return a + b;
			}
		});

		// Call the PerformOperation function with subtraction:
		int resultOfSubtraction = performOperation(x, y,
				// Anonymous inner class used as a parameter.
				new MathOperation() {
			public int operation(int a, int b) {
				return a - b;
			}
		});

		// Output Addition: 197
		System.out.println("Addition: " + resultOfAddition);

		// Output Subtraction: 3
		System.out.println("Subtraction: " + resultOfSubtraction);
	}
}
```

In Code Listing 4.4, the relevant changes are highlighted in yellow. Interfaces consist solely of abstract methods, so there is no longer a body for the operation method defined in the **MathOperation** interface.

The examples we have seen so far show the basic syntax of an anonymous inner class—they do not illustrate the most common usage of this mechanism. Inner classes are most commonly used to provide functionality for callbacks in event-driven GUI programming. We will see extensive use of inner classes in the section on GUI events in Chapter 7. The basic objective is to save code. We do not need to declare a class and create an instance to state what should occur when the user clicks a mouse or presses a key at the keyboard. Instead, we declare and define an anonymous inner class that specifies the action to be undertaken when these events occur. This makes our code easier to read, and shorter, because we define the functionality at the point where we need it (i.e. where the event is being created) instead of defining the functionality in some external class.

Chapter 5 Multithreading

Multithreading is a technique for programming more than one execution unit at the same time. Computers traditionally run with a single CPU executing the code. The CPU runs through the instructions one after another, jumping to various methods. CPUs execute code very quickly, but there is a limit to the speed any CPU can execute. A modern CPU can execute billions of instructions every second, but it is very costly to increase this execution speed—the hardware begins to require extreme measures to prevent the CPU from melting or catching on fire (for instance, water and even liquid nitrogen have been used to cool very fast CPUs).

Thankfully, we can greatly improve the performance of our CPUs without increasing the clock speed of the units. Instead, manufacturers add multiple execution units to a single die (die is simply a word for the physical object inside the computer upon which the CPU is etched). The units are called cores, and each can be thought of as being a complete CPU of its own.

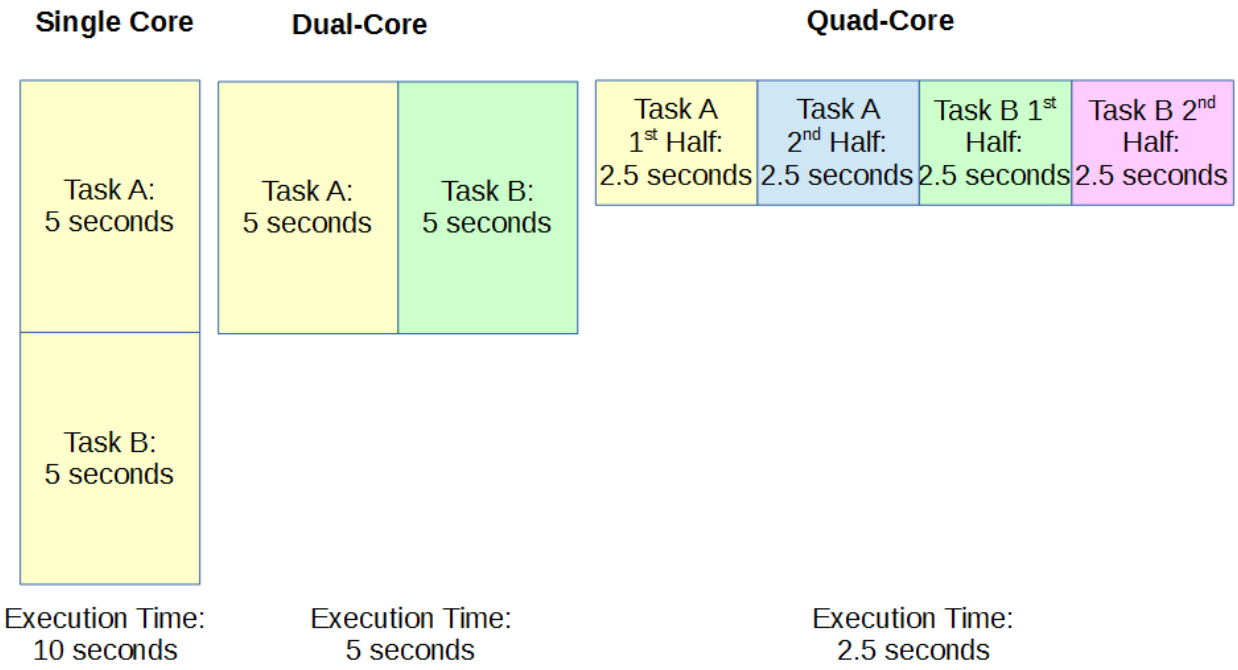


Figure 10: Multithreading

Figure 10 shows a hypothetical, best-case scenario for multicore CPUs. On the left, we see a single core performing two tasks. A single core executes Task A first, and when Task A is finished, it executes Task B. Both tasks take the CPU five seconds to execute, and thus the total execution time is 10 seconds.

In the middle of Figure 10, the dual-core CPU can execute Tasks A and B at the same time by allowing each to be executed by one of its two cores. Each task takes five seconds to compute, but the tasks are executed at the same time, therefore they will finish at the same time, taking five seconds (plus a small amount of time for overhead).

Finally, the right side of Figure 10 shows a quad-core CPU. It is sometimes possible to split tasks into several sections, and in the diagram, we have assigned one of the four cores of our quad-core CPU to execute a half of Tasks A or B. Execution of half of a task takes 2.5 seconds, and all four cores will finish after approximately 2.5 seconds. This speed represents four times the performance of the single core CPU.

Figure 10 shows a hypothetical case. This is the best possible case, and in practice tasks do not often split in half so easily. However, you can see that as the number of cores increases, the ability of the cores to share the workload becomes very useful. It is often practical to improve the performance of our applications by 200% or even 400% by employing multithreading and by cleverly allocating our workloads to different cores. As we shall see, typically cores are required to communicate and coordinate their actions with each other, and often we do not get a straight 100% improvement in speed when we add another core.

Multithreading is an extremely vast and complex topic, and we will only scratch the very surface in this text, but you should practice your multithreading skills frequently because the future of computing is very heavily dependent on efficient multithreading.

Threads

A thread is an execution unit. For instance, when our application is run by the user, the JVM will create a single, main thread for us to begin executing the **`main`** method. It may also create several background threads for garbage collection (to delete unneeded objects behind the scenes), and other background tasks. The main thread begins by executing the code from the **`main`** method, as we have seen many times.

Call stack

When a thread executes code, it jumps to various points in the code while it calls the methods. These method calls can be nested (i.e. a method can call itself; it can call methods). Methods may require parameters to be passed, and they can specify their own local variables. In order to return from methods in the correct order, to pass parameters to and from methods, and to keep track of the local variables of methods, an area of RAM is allocated called the “call stack.” Each thread has its own call stack, and threads can potentially call any sequence of different methods.

Eclipse shows a simple version of the program’s call stack when it pauses at a breakpoint. Figure 11 shows a screenshot of the Debug window while a program runs. The information presented is the name of the running application class (**`MainClass`**). The thread is called **`[main]`**, and it suspended due to a breakpoint at line 18 in the **`Animal`** class source code file. The next lines are the call stack. The program has executed the **`Animal.print`** method, along with the method before that—the **`MainClass.main`** method (which called the **`Animal.print`** method).

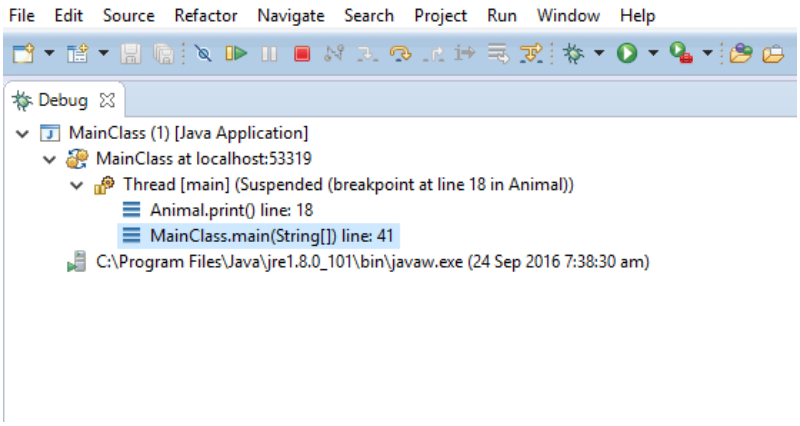


Figure 11: Debug Window

In Java, threads are resource-heavy. It takes time for the system to create and run a new thread, and creating a new thread results in the allocation of other system resources (such as RAM for the call stack). We should never attempt to create hundreds of threads, nor should we attempt to create and kill threads within a tight loop. Thread creation is slow, and the number of threads a system can concurrently execute is always limited by the physical hardware (i.e. the number of cores in the system, the amount of RAM, the speed of the CPU, etc.). There are several methods by which multiple threads can be created in Java, and we will look at two—implementing the **`Runnable`** interface and extending the **`Thread`** class.

Note: When we create a new thread, it will often be executed on a new core within the CPU. However, cores and threads are not always directly associated. Often, the operating system switches threads on and off the cores, giving each thread a small amount of time (called a time-slice) in order to execute some portion of code. Even a single core CPU can emulate multithreading by quickly switching between threads, allocating to each thread some time-slice on the physical core.

Implementing Runnable

In order to use multiple threads, we can create a class that implements the **`Runnable`** interface. This allows us to create a class with a private member thread. The **`Runnable`** interface defines an abstract method called **`run`** that we must implement in our derived classes. When we create a new thread, it will execute this method (probably using a different core in the CPU to the core that executes the **`main`** method). Note that we can create two, four or even eight threads, even if the hardware only has a dual-core CPU. But be careful—as mentioned, threads are resource-heavy, and if you try to create 100 or 1000 threads, your program will not run blisteringly fast, it will stop completely and possibly crash the program, if not the entire system (requiring a reboot).

Code Listing 5.0: MainClass

```
public class MainClass {
      public static void main(String[] args) {
            // Define Thready objects:
```

```
            Thready t1 = new Thready("Ned");
            Thready t2 = new Thready("Kelly");

            // Start the threads:
            t1.initThread();
            t2.initThread();
      }
}
```

Code Listing 5.1: Thready Class

```
public class Thready implements Runnable {

      // Private member variables:
      private Thread thread;
      private String name;

      // Constructor:
      public Thready(String name) {
            this.name = name;

            System.out.println("Created thread: " + name);
      }

      // Init and start thread method:
      public void initThread() {
            System.out.println("Initializing thread: " + name);

            thread = new Thread(this, name);

            thread.start();
      }

      // Overridden run method:
      public void run() {
            // Print initial message:
            System.out.println("Running thread: " + name);

            // Count to 10:
            for(int i = 0; i < 10; i++) {
                  System.out.println("Thread " + name + " counted " + i);
```

```
                try {
                        // Wait for 1 second:
                        Thread.sleep(1000);
                }
                catch (Exception e) {
                        System.out.println("Error: " + e.getMessage());
                }
        }
    }
}
```

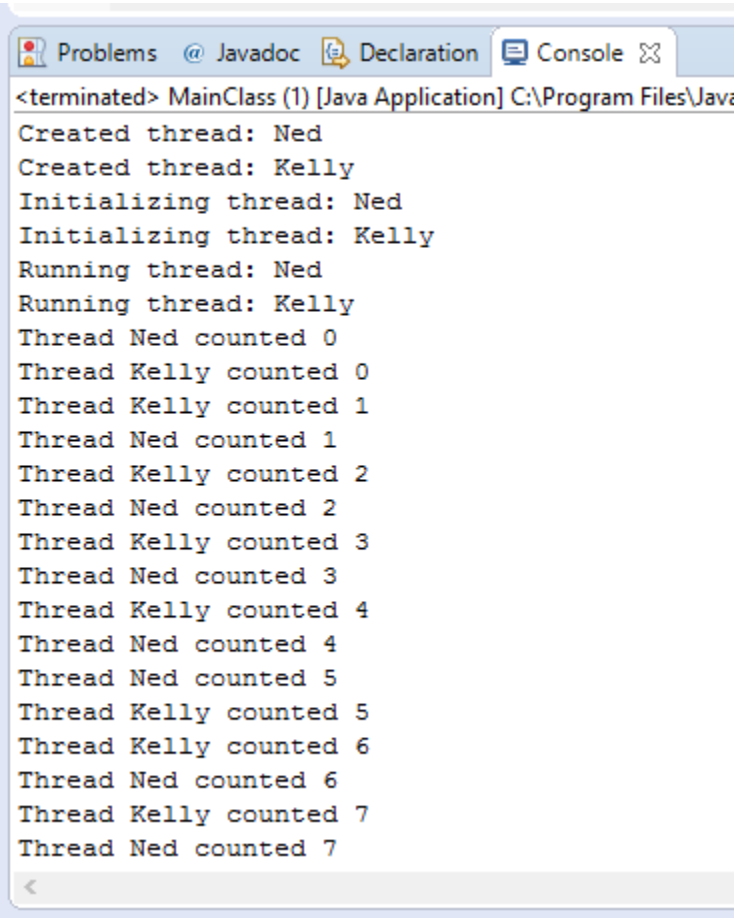


Figure 12: Output from Ned and Kelly

Figure 12 shows one possible output from the program in Code Listings 5.0 and 5.1. Code Listing 5.1 shows the **`Thready`** class that implements the **`Runnable`** interface, and it defines the **`run`** method, which is required to implement **`Runnable`**. The class defines a **`Thread`** object called **`thread`**, which we instantiate in the method called **`initThread`**. Then we call the **`Thread`** object's **`start`** method, which will in turn call the **`Runnable`** interface's **`run`** method. In the **`run`** method, we count from 0 to nine, pausing for one second between each number. Upon running

the application, you will see the two threads (created in the **`MainClass`** from Code Listing 5.0) counting slowly to nine.

There are several very important facts about the program from Code Listings 5.0 and 5.1:

- The threads count at the same time, and thus the program takes only about 10 seconds to execute.
- The exact timing and order of the counting threads is not known to us (**`Ned`** could inexplicably count slightly faster than **`Kelly`**, or vice versa).
- If you run this application on a multicore desktop PC, there is a high likelihood that **`Ned`** and **`Kelly`** (our threads) will run on different cores inside the machine.

Concurrency

The point above about not knowing the exact order of execution is important! When we look at the code from Code Listings 5.0 and 5.1, we cannot tell what will happen. **`Ned`** could count faster, or **`Kelly`** could count faster (the two will count at approximately one second per number, but on the nanosecond level, one thread will always beat the other).

The two threads could count completely randomly—they could swap leader every number, so that each time we execute the application, we might get a different output. **`Ned`** and **`Kelly`** are called "concurrent." If you cannot determine in which order the threads will execute simply by looking at the code, then the code is concurrent. It is never safe to assume an order of execution for concurrent threads (in fact, concurrency means we cannot assume the order!). We must be extremely careful when we coordinate concurrent threads. We cannot tell what will happen when we look at the code because the CPU's task is extremely complicated—it is executing the operating system and hundreds of background tasks. It executes time-slices of each thread and switches the background processes on and off the physical cores. Somewhere, in this mess of instructions, our humble little **`Ned`** and **`Kelly`** threads will be given some time to execute on a core, then they are put to sleep for some other program to execute on the core. We have no practical way of guessing in which order our threads will execute, thus our threads are concurrent. Generally, we hope that the CPU is not too busy executing background tasks, and when we create threads, we aim for them to be executed in their entirety, uninterrupted and simultaneously, but we cannot guarantee that this will happen.

Thread coordination

When the tasks that our threads perform are completely independent, the algorithm is called embarrassingly parallel. Embarrassingly parallel algorithms are the best-case scenario for multithreading because we can split the workload perfectly and threads do not need to communicate or synchronize in any way. This means each thread can perform its assigned task as quickly as possible with no interruptions and without worrying about what any other threads are doing. In real-world applications, many algorithms do not split so perfectly into two or more parts. The workload of each thread is typically not 100% independent from that of the other threads. Threads need to communicate.

In order for one thread to communicate with another, the threads need a shared resource. Imagine **`Ned`** and **`Kelly`** wish to perform two tasks—BoilWater and PourCoffee. The problem is

that we need the water boiled before the coffee is poured. So, if **`Ned`** is assigned the task of boiling the water, and **`Kelly`** is assigned the task of pouring the coffee, then **`Ned`** needs some way to indicate to **`Kelly`** that the water has been boiled. And **`Kelly`** must wait for some signal from **`Ned`** before she pours the coffee.

Low-level concurrency pitfalls

Let us take moment to examine some important concepts and pitfalls involved in concurrent programming. This section might seem particularly low level, but nothing in concurrent programming makes sense unless we know why we must watch our step.

In the current context, resources are variables. Shared resources are variables to which multiple threads can read or write. When we share variables between threads, we need to be careful not to allow race conditions. A race condition occurs when two threads might potentially alter a variable at the same time. Imagine two threads trying to increment a shared resource that is initially set to 0. The operation appears trivial—we want two threads to increment the variable, so the result should be two. The trouble is, the act of incrementing a variable is not atomic. If an operation is not atomic, it can be broken into several steps. These steps are called the Read/Modify/Write cycle. A thread first reads the current value of the variable from RAM, then it modifies it by performing the increment on a temporary copy of the variable, and finally it writes the result back to the actual variable in RAM.

Modern CPUs perform almost all operations on variables using a Read/Modify/Write cycle because they do not have the ability to perform arithmetic on the data in RAM. RAM does not work that way—it allows two operations: read a value at some address or write a value to some address. It does not allow a CPU to add two values together or subtract one from the other. Therefore, the CPU requests some variable from RAM, storing a copy in its internal registers (a register is a variable inside the CPU). The CPU then performs the arithmetic operation on this copy and finally sends the results back to RAM.

Figure 13 shows an example of a race condition. The example shows two threads trying to increment the shared variable from 0 to 2 at the same time. The two threads execute one step at a time, and the time is listed along the left side of the diagram. Both threads read the value of the shared variable as 0, increment this to 1 using their internal register, and write the 1 as the result. We can see that the final result at time-step 4 is 1 instead of 2. But this is not the only possibility—the CPU is making up the results as it schedules the threads for execution, and the result is out of the programmer’s control.

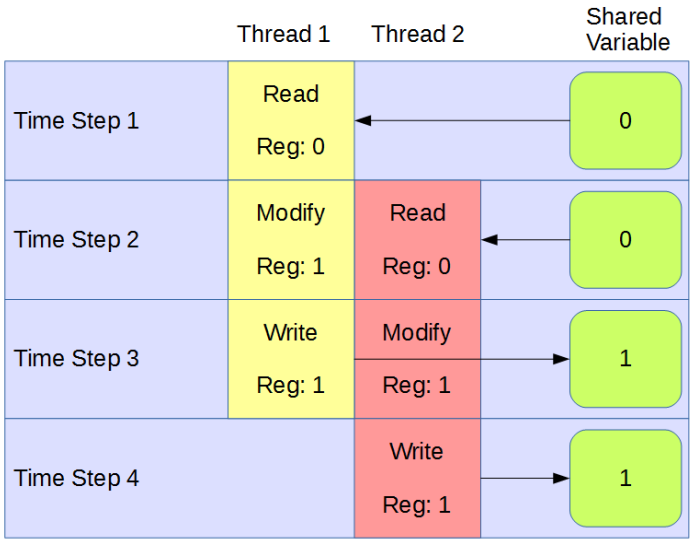


Figure 13: Race Condition

In order to use shared resources, we must be very careful to ensure that there are no race conditions. This often means that only one thread is allowed access to a shared resource at a time. In order to create shared resources in Java, we can use the **`synchronized`** keyword. Any method marked as **`synchronized`** will allow exactly one thread to execute the method at a time. This means that if we modify a variable from within a **`synchronized`** code block, we can be guaranteed that only one thread at a time is allowed access.

Mutex

Mutual Exclusion, or a mutex, is a parallel primitive. It is a mechanism used in parallel programming that allows only one thread at a time to access some section of code. A mutex is used to build critical sections in our code that only one thread at a time is allowed to execute. No mutex is provided in the standard Java libraries, so, as an exercise, we will create one.

The mutex has two methods associated with it—**`grabMutex`** and **`releaseMutex`**. The purpose of a mutex is to allow only one thread at a time to complete the call to **`grabMutex`**. Once a thread has the mutex (or, in other words, has successfully completed a call to **`grabMutex`**), any other threads that try to call **`grabMutex`** will block—they will stop execution and wait for the mutex to be released. Thus, any operations performed while a thread has the mutex are atomic. They cannot be interrupted by any other thread until the mutex is released.

In Java, we must synchronize on an object. That is, we must use some object as a lock in order to successfully design a mutually exclusive code block.

Code Listing 5.2: Main Method for Mutex

```
public class MainClass {
      public static void main(String[] args) {
```

```
            // Create three threads:
            Thready t1 = new Thready("Ned");
            Thready t2 = new Thready("Kelly");
            Thready t3 = new Thready("Pole");

            // init and run the threads.
            t1.initThread();
            t2.initThread();
            t3.initThread();

            // Wait for the threads to finish:
            while(t1.isRunning()) {
            }

            while(t2.isRunning()) {
            }

            while(t3.isRunning()) {
            }

            // Check what the counter is:
            System.out.println("All done!" +  Thready.getJ());
      }
}
```

Let's now look at three versions of the **Thready** class, each with a slightly different **run** method. I will only include the code for the complete class in the first example, as Code Listing 5.3 shows the complete code for the **Thready** class, although in this code I have purposely designed the class so that the threads are prone to race conditions.

Code Listing 5.3: Thready Class with Race Conditions

```
public class Thready implements Runnable {

      // A shared resource:
      public class Counter {
            private int j;

            public Counter() {
                  j = 0;
            }
```

```
        public int getJ() {
                return j;
        }
}

// Thready member variables
private Thread thread;
private String name;
private boolean running = true;

// Static shared resource:
private static Counter counter = null;

// Getters:
public static int getJ() {
        return counter.getJ();
}

public boolean isRunning() {
        return running;
}

// Constructor
public Thready(String name) {
        // Create the shared resource
        if(counter == null)
                counter = new Counter();

        // Assign name
        this.name = name;

        // Print message
        System.out.println("Created thread: " + name);
}

public void initThread() {
        // Print message
        System.out.println("Initializing thread: " + name);

        // Create thread
        thread = new Thread(this, name);
```

```
            // Call run
            thread.start();
      }

      public void run() {
            for(int q = 0; q < 10000; q++) {
                  counter.j++;      // RACE CONDITION!!!
            }

            running = false;
      }
}
```

Notice the line marked with comment "**RACE CONDITION!!!**" in Code Listing 5.3. The **main** method in Code Listing 5.2 creates and executes three threads, **Ned**, **Kelly**, and **Pole**. All three threads try to increment the shared **counter.j** variable in their **run** methods, but they do so at the same time with no coordination. Race conditions are disastrous, and to prove that they are much more than mere theory, run the program a few times and witness the final value that the **main** method reports. The **main** method will almost never count up to the intended value of 30,000 (i.e. the value we expect when three threads each increment a variable 10,000 times). It reports 12,672 and 13,722. In fact, it seems to report anything it wants, and we know why—the threads are incrementing their own copies of the shared resource and only occasionally writing a successful update! Let's take a moment to implement a mutex and see if we can fix the accesses to this shared resource.

Code Listing 5.4: Using a Mutex 1 (The Slow Way)

```
      public void run() {
            for(int q = 0; q < 10000; q++) {
                  synchronized (counter) {
                        counter.j++;
                  }
            }
      running = false;
      }
```

The second version of the **run** method is slightly better (Code Listing 5.4). We have employed the **synchronized** keyword and locked the code inside the loop using the shared resource as the key (this lock is our mutex). The **synchronized** keyword takes a resource to synchronize with, and it is followed by a code block. Only one of the three threads is able to execute inside the **synchronized** code block at a time, therefore the line of code "**counter.j++**" is a critical section. The threads will wait until the lock (or the mutex) is released, and they will take turns to enter the critical section, increment **counter.j**, then release the lock.

Grabbing and releasing the lock takes time. Ultimately, we want threads to work independently for as long as possible before they synchronize. If you run the application (and please do not) with the new version of the **run** method, you may have to wait a very long time for the threads to throw the lock about 30,000 times and increment **counter.j** all the way to 30,000. They will eventually finish, but it could take 10 minutes or it could take an hour. Code Listing 5.5 shows a far better way of doing this.

Note: You can also mark a method with the synchronized keyword. When we mark a method as synchronized, we are saying that the method can only be used by one thread at a time. However, we must be very careful because a synchronized method actually implicitly synchronizes with the "this" keyword as the object for the lock. That is—if we have a synchronized method in a class and we try to create multiple threads of that class, when we call the method, the threads will not synchronize because each of them is using itself as the lock.

Code Listing 5.5: Using a Mutex 2 (The Fast Way)

```
public void run() {
      synchronized (counter) {
            for(int q = 0; q < 10000; q++) {

                  counter.j++;
            }
      }
      running = false;
}
```

In Code Listing 5.5, we place the lock outside the for loop. This is the only difference, but it makes a huge difference in performance. Now, when a thread grabs the lock, it increments **counter.j** 10,000 times before it releases the lock. This means the lock is grabbed three times instead of 30,000, and the performance is far better.

Code Listing 5.5 might seem like the obvious choice from the examples I have presented, and in the present case, I would advise this. But the program has a big drawback—the increments are now happening sequentially. The threads are not performing their workloads at the same time, each one is either incrementing the counter or waiting for the lock. There is no point to incrementing a counter in this way because the main thread can perform the increment without allocating any new threads at all. Concurrent programming is a juggling act between coordinating threads to perform independent tasks simultaneously and allowing them to synchronize/communicate so that we avoid race conditions and the programmer remains in control of the outcome.

This example is problematic because incrementing a counter is not a suitable task for multithreading. Selecting which tasks in a program are suitable for designing concurrently is one of the most important aspects of multithreading.

Extending the Thread class

We've looked at implementing **`Runnable`** for multithreading. The second method for multithreading is to extend the **`Thread`** class. In Code Listings 5.6 and 5.7, a new thread is created by extending the **`Thread`** class and executing the **`Thread.start()`** method. The objective of our threads is to find factors of the shared number **`x`**. The threads do so by partitioning the values from **`2`** to **`sqrt(x)`** into two parts, and each thread checks for factors in half of this range. This program uses the brute force method for finding factors, and it is not intended to be a useful program for solving the factoring problem.

Code Listing 5.6: Main Method

```
// MainClass.java
public class MainClass {
      public static void main(String[] args) {
            // Define some threads:
            Thready thread0 = new Thready(0);
            Thready thread1 = new Thready(1);
            // Set the value to factor:
            Thready.x = 36847153;
            // Start the threads:
            thread0.start();
            thread1.start();

            // Wait for the threads to finish.
            while(thread0.isAlive()) { }
            while(thread1.isAlive()) { }
            // Print out the factors the threads found:
            System.out.println(
                        "Smallest Factor found by thread0: " +
thread0.smallestFactor);
            System.out.println(
                        "Smallest Factor found by thread1: " +
thread1.smallestFactor);
      }
}
```

Code Listing 5.7: Extending the Thread Class

```
// The class Extends the Thread class.
public class Thready extends Thread {
      // Shared resource
      public static int x;
      public int id;
```

```
	// Smallest factor found by this thread:
	public int smallestFactor = -1;

	// Constructor
	public Thready(int id) {
		this.id = id;
	}

	// Run method
	public void run() {
		// Figure out the root:
		int rootOfX = (int)Math.sqrt(x) + 1;

		// Figure out the start and finish points:
		int start = (rootOfX / 2) * id;
		int finish = start + (rootOfX / 2);

		// If the number is even:
		if(x % 2 == 0) {
			smallestFactor = 2;
			return;
		}
		// Don't check 0 and 1 as a factor:
		if(start == 0 || start == 1)
			start = 3;

		// Only check odd numbers:
		if(start % 2 == 0)
			start++;

		// Try to find a factor.
		for(int i = start; i < finish; i+=2) {
			if(x % i == 0) {
				smallestFactor = i;
				break;
			}
		}
	}
}
```

In general, using the Java **`Runnable`** mechanism is preferred to extending the **`Thread`** class for most scenarios. A detailed comparison of the two techniques is outside the scope of this e-book, but you can find many interesting discussions on the topic at Stack Overflow (http://stackoverflow.com/).

Chapter 6 Introduction to GUI Programming

In this chapter, we will look at some of the mechanisms provided by Java that allow us to design and implement Graphical User Interfaces (GUIs). We will create an application with a GUI in the next chapter using the windows builder, but in order for that code to make sense, we must examine how to manually build simple GUIs.

In Java, we can use one of two packages to offer users a GUI—Abstract Window Toolkit (AWT) or Swing. AWT has been largely superseded by Swing, and this chapter offers an introduction to GUI programming using Swing. The Swing toolkit is a wrapper around the AWT, which means we need to import both Swing and AWT when programming with the Swing toolkit.

A GUI is created by placing controls, such as buttons, picture boxes, combo boxes, etc., onto a window. The user interacts with the controls using the mouse or the pointer or the keyboard. Swing control names begin with the letter J, such as **`JPanel`** and **`JButton`**.

Begin by creating a new project. I have called mine **`GUITesting`**. Add a class that extends the **`JFrame`** class, as in Code Listing 6.0. A **`JFrame`** is a window—it has a control box in the upper corner that allows the user to close the window, and the window can be resized and moved around the user's desktop like a normal window.

Code Listing 6.0: MainWindow.java

```
import java.awt.*;
import javax.swing.*;

public class MainClass extends JFrame{
	// Main method
	public static void main(String[] args) {
		MainClass mainWindow = new MainClass();
	}

	// Constructor
	public MainClass() {
		// Set the size of the window.
		this.setSize(640,  480);

		// Set the app to close when the window closes.
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);

		// Set the window to visible.
		this.setVisible(true);
	}
```

```
}
```

Code Listing 6.0 shows a simple class called **MainClass** that creates and shows a blank window (see Figure 14). The class consists of the **main** method, which does nothing but create a new **MainClass** object called **mainWindow**. In the constructor, we set the size of the window to **640** by **480** pixels, and we set the visibility of the window to **true**. Note the class **extends** the **JFrame** class. The method **setDefaultCloseOperation** causes the application to close when we close the window. Without the call to **setDefaultCloseOperation**, the application continues to run in the background even after the user has shut down the window.

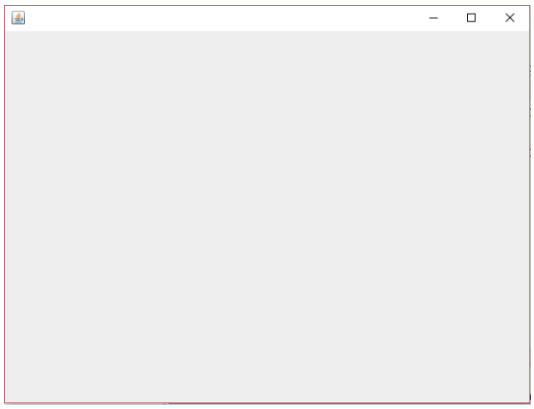


Figure 14: Blank JFrame

In order to add controls, we can create a new **JPanel** object. A **JPanel** is a control that holds other controls. You can also add controls directly to the **JFrame** itself by creating a Container object and calling the **this.getContentPane()** method, but we will use a **JPanel**. Code Listing 6.1 shows how to create and add a new **JPanel** and some controls to our project.

Code Listing 6.1: Adding Controls to a JPanel

```
	// Constructor
	public MainClass() {
		// Create a JPanel
		JPanel panel = new JPanel(new FlowLayout());

		// Add some controls.
		panel.add(new JLabel("Test Button: "));
		panel.add(new JButton("Click me!"));

		// Set the current content pane to the panel.
		this.setContentPane(panel);


		// Set the size of the window.
```

```
        this.setSize(640,  480);

        // Set the app to close when the window closes.
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the window to visible.
        this.setVisible(true);
    }
```

In order to add the label and button in Code Listing 6.1, we use the **`JPanel.add`** method. The method requires a control, and each control has a constructor that takes a **`String`**. The **`String`** is typically used as the caption for the control when it is displayed on the screen. Figure 15 shows the label and button from Code Listing 6.1.

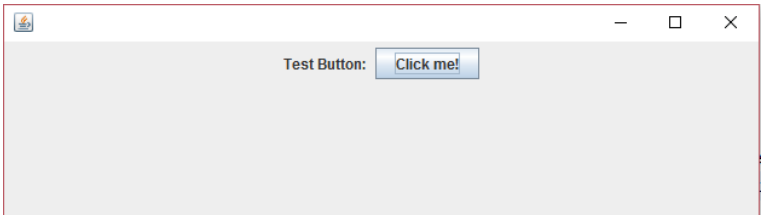


Figure 15: Adding Controls to a JPanel

Notice that in Code Listing 6.1 we create the **`JPanel`** and specify a layout in the constructor. When we add control to a **`JPanel`**, the **`JPanel`** organizes according to the layout. The **`FlowLayout`** is actually the default for **`JPanels`**. With a **`FlowLayout`**, all the controls are added in a single row. Table 2 contains all of the layouts and a brief description of each. For further information, visit https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html.

Table 2: Layouts

Layout	Description
BorderLayout	The panel is split into top, bottom, left, right, and center.
BoxLayout	Places controls in a single row or column.
CardLayout	Allows us to switch among several sets of controls.
FlowLayout	Lays out controls in a single row.
GridBagLayout	Lays out controls in a grid where controls can occupy multiple cells.
GridLayout	Lays out controls in a grid.
GroupLayout	Supplies horizontal and vertical layouts seperately; designed for GUI builders.
SpringLayout	Lays out controls with respect to relationships between their positions; designed for GUI builders.

Controls are the crux of GUI. Each control type has a special purpose. The classes often offer specific methods for controls, but many methods are available for all controls, as Code Listing 6.2 demonstrates.

Code Listing 6.2: Useful Control Methods

```
            JButton btn = new JButton("Initial Text");

            // Useful control methods:
            btn.setText("New text!");     // Set the text on the button.
            String text = btn.getText();  // Read the current text.
            btn.setVisible(false);  // Hide the control from view.
            btn.setVisible(true);   // Show the control.
            btn.setMargin(new Insets(100, 100, 100, 100)); // Set margins.
            Dimension dim = btn.getSize();// Read the size of the control.
            btn.setBackground(Color.BLUE);// Set the background color.
            btn.setForeground(Color.WHITE);// Set the foreground/text
color.
            btn.setEnabled(false);// Disable interactions with the control.
            btn.setEnabled(true); // Enable interactions with the control.

            // Depending on the layout manager, these may do nothing:
            btn.setSize(new Dimension(10, 10)); // Set size of the control.

            // Set size and position of the control:
            btn.setBounds(new Rectangle(20, 20, 200, 60));
```

Events and event listeners

In order to make our controls functional, we need to add **`ActionListeners`**.

`ActionListeners` are typically run on a separate thread from our main thread—they do nothing more than wait for the user to interact with our controls, then they call the appropriate method when the user does so. **`ActionListeners`** repeatedly monitor the state of the control and alert us to changes we are interested in, such as when the user clicks a button or changes the text in a text box. This is event-driven programming. We set up a collection of controls and assign them methods that we want to call when specific actions occur with the controls. We do not have to specifically check the state of our controls ourselves because Java does all of the back-end code for us. You can check Wikipedia's page on event-driven programming for more information on this fascinating topic at https://en.wikipedia.org/wiki/Event-driven_programming.

Code Listing 6.3 shows how to implement the **`ActionListener`** interface and respond to the user clicking a button.

Code Listing 6.3: Implementing an ActionListener in MainClass

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

import javax.swing.*;

public class MainClass extends JFrame implements ActionListener {
	// Main method
	public static void main(String[] args) {
		MainClass mainWindow = new MainClass();
	}

	// Constructor
	public MainClass() {
		// Create a JPanel.
		JPanel panel = new JPanel(new FlowLayout());

		// Add some controls:
		panel.add(new JLabel("Test Button: "));
		JButton btnClickMe = new JButton("Click me!");
		panel.add(btnClickMe);

		// Set the current content pane to the panel:
		this.setContentPane(panel);

		// Set this as the current action listener for the button
		btnClickMe.addActionListener(this);

		// Set the size of the window.
		this.setSize(640,  480);

		// Set the app to close when the window closes.
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);

		// Set the window to visible.
		this.setVisible(true);
	}

	// Method inherited from the ActionListener interface:
	public void actionPerformed(ActionEvent e) {
```

```
            // Show a message box:
            JOptionPane.showMessageDialog(null,
                  "You clicked on the button!");
      }
}
```

In Code Listing 6.3, I have changed the button to a local variable called **btnClickMe**. We implement the **ActionListener** interface and supply this as the **ActionListener** for the button. The **ActionListener** interface defines the function **actionPerformed**, and the code of this method will be executed when the user clicks the button. Upon running the program, you should be able to click the button and see a message box pop up.

Supplying the code that occurs for an event using an anonymous class is often more practical than implementing the **ActionListener** interface in some existing class. Code Listing 6.4 shows the same example as Code Listing 6.3, except that this time I have used an anonymous class to show the message box to the user when the button is clicked. Notice that in Code Listing 6.4 the **MainClass** no longer implements the **ActionListener** interface. This method for specifying events is what the windows builder uses, as we shall see in the next chapter.

Code Listing 6.4: ActionListener with Anonymous Class

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

import javax.swing.*;

public class MainClass extends JFrame {
      // Main method
      public static void main(String[] args) {
            MainClass mainWindow = new MainClass();
      }

      // Constructor
      public MainClass() {
            // Create a JPanel.
            JPanel panel = new JPanel(new FlowLayout());

            // Add some controls.
            panel.add(new JLabel("Test Button: "));
            JButton btnClickMe = new JButton("Click me!");
            panel.add(btnClickMe);

            // Set the current content pane to the panel.
```

```
            this.setContentPane(panel);

            // ActionListener as anonymous class.
            btnClickMe.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        // Show a message box.
                        JOptionPane.showMessageDialog(null,
                              "You clicked on the button!");
                  }
            });

            // Set the size of the window.
            this.setSize(640,  480);

            // Set the app to close when the window closes.
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);

            // Set the window to visible.
            this.setVisible(true);
      }
}
```

Example BorderLayout

As a final example of manual GUI building, in this section we will explore the **BorderLayout**, add more than one **ActionListener** to a single project, and respond to events by altering the state of a **TextArea**. This section is intended to provide an additional, slightly more complex example of manually coding GUI before we move on to using the windows builder. Practicing at least some manual GUI programming helps because it is often faster to manually fix problems that are created when using the windows builder, and it helps us to understand the code that the builder provides. A **BorderLayout** manager allows us to add controls to five regions of a panel—**PAGE_START**, **PAGE_END**, **LINE_START**, **LINE_END**, and **CENTER**. Figure 16 shows an example panel with these regions colored and labelled.

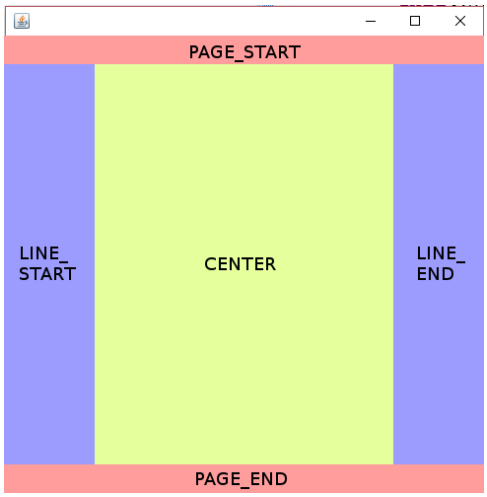


Figure 16: BorderLayout Regions

When we add a control to a **BorderLayout**, we specify the location of the control (**PAGE_START** or **CENTER**, etc.) and the layout manager takes care of resizing the controls so that they fill the entire region (with an optional gap surrounding each region).

Code Listing 6.5: Using the BorderLayout

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class MainClass extends JFrame implements ActionListener {
	public static void main(String[] args) {
		MainClass m = new MainClass();
		m.run();
	}

	// Declare txtInput.
	private JTextArea txtInput;

	private void run() {
		// Create a new border layout and main panel for controls.
		BorderLayout layoutManager = new BorderLayout();
		JPanel mainPanel = new JPanel(layoutManager);
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);
		this.setSize(500,  500);
		this.setContentPane(mainPanel);
		this.setVisible(true);
```

```
            // Set margins around control in layout.
            layoutManager.setHgap(25);
            layoutManager.setVgap(25);

            // Create buttons.
            JButton btnTop = new JButton("Page Start");
            JButton btnBottom = new JButton("Page End");
            JButton btnLeft = new JButton("Line Start");
            JButton btnRight = new JButton("Line End");

            // Add the buttons to panel.
            mainPanel.add(btnTop, BorderLayout.PAGE_START);
            mainPanel.add(btnBottom, BorderLayout.PAGE_END);
            mainPanel.add(btnLeft, BorderLayout.LINE_START);
            mainPanel.add(btnRight, BorderLayout.LINE_END);

            // Create and add a text area.
            txtInput = new JTextArea(5, 10);
            txtInput.setText("Click a button!");
            JScrollPane jsp = new JScrollPane(txtInput);
            txtInput.setEditable(false);
            mainPanel.add(jsp, BorderLayout.CENTER);

            // Add action listeners to respond to button clicks.
            btnTop.addActionListener(this);
            btnBottom.addActionListener(this);
            btnLeft.addActionListener(this);
            btnRight.addActionListener(this);

            // Redraw all controls to ensure all are visible.
            this.validate();
      }

      // Action performed prints the clicked button's text to the
      // txtOutput control.
      public void actionPerformed(ActionEvent arg0) {
            txtInput.append("You clicked " +
                        ((JButton)arg0.getSource()).getText() +
                        "\n");
      }
}
```

Using a **`BorderLayout`**, we can set the horizontal and vertical gap between controls using the **`setHGap`** and **`setVGap`** methods of the **`BorderLayout`** object. This effects the margins between controls. The manager resizes the controls in order to fill the entire region by default. After we add the controls in Code Listing 6.5, we use **`this.validate()`** to ensure that all controls are redrawn. Without this call, one or more of the controls may not be visible until the user resizes the panel (or performs some other action that causes the controls to be validated). Validating when you add or remove controls from a panel is always a good idea. Figure 17 shows an example of the program from Code Listing 6.5.

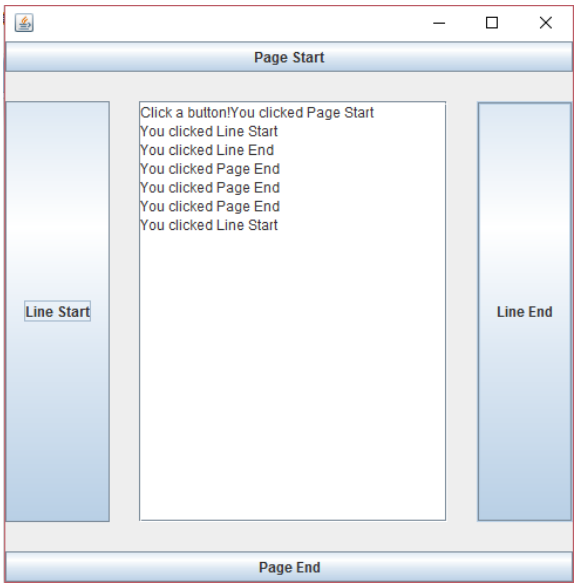
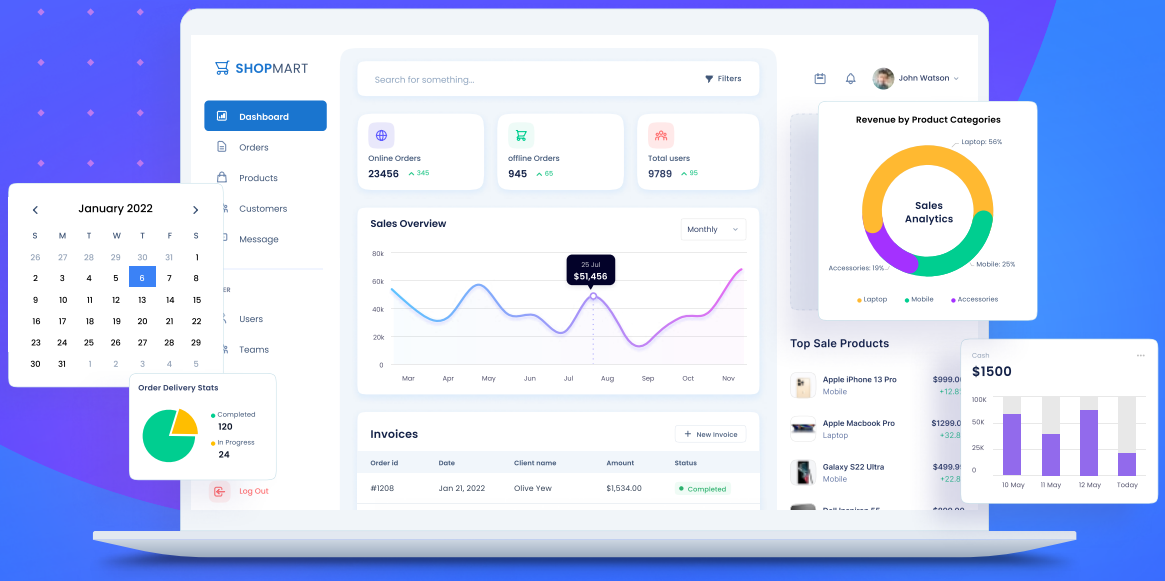


Figure 17: BorderLayout Example

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Chapter 7 GUI Windows Builder

In this section, we will use Eclipse's WindowBuilder to build a more complex GUI using a WYSIWYG (what you see is what you get) drag-and-drop system. This greatly increases the control and speed of GUI development. We will build a simple calculator application. I will only include basic arithmetic operations, but we will see that adding new functionality to our calculator is quick and easy.

Note: I have used Eclipse exclusively throughout this e-***book, but there are other IDEs available, and some have their own GUI building tools—IntelliJ has IDEA, for example. It is also possible to use GUI builders***, ***such as*** JFormDesigner***, that are designed for multiple IDEs***.

Adding a window

Next, let's create a new Java project as we have done previously. I have called mine **`Calculator`**. When your new project is created, add a main window by right-clicking the **src** folder in the Package Explorer and clicking **New** and **Other**, as in Figure 18.

The Eclipse IDE does not come with the WindowBuilder packages already installed, which means you must install them separately. The detailed instructions on how to do this are outside the scope of this e-book, but they can be found with an Internet search. Briefly, you choose the **Help | Install New Software** menu item from the Eclipse toolbar, then use a step-by-step wizard to install the four or so packages (some, such as Documentation, are optional) that are used to create a GUI application using WindowBuilder/Swing.

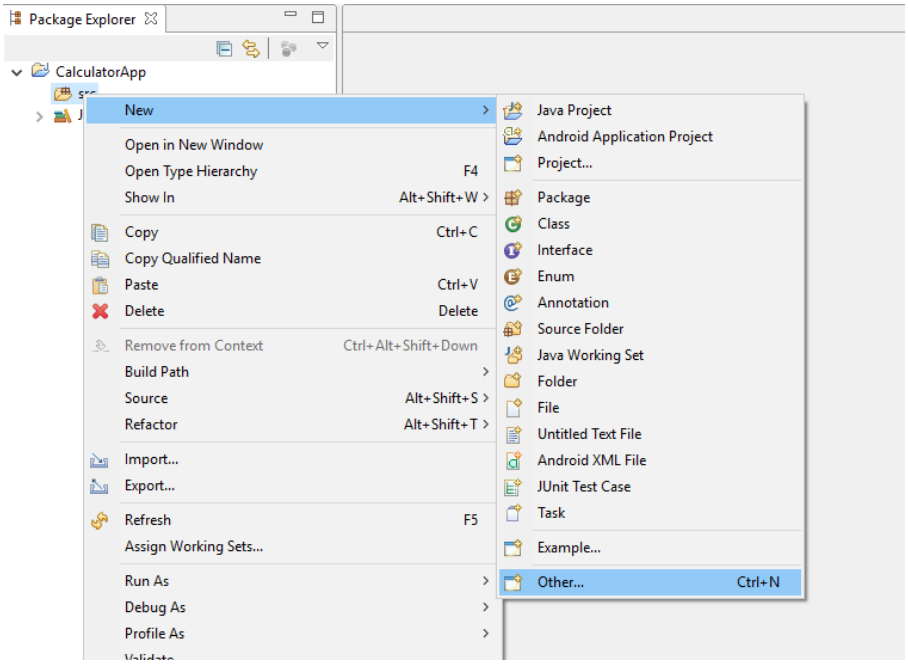


Figure 18: Adding a Window Step 1

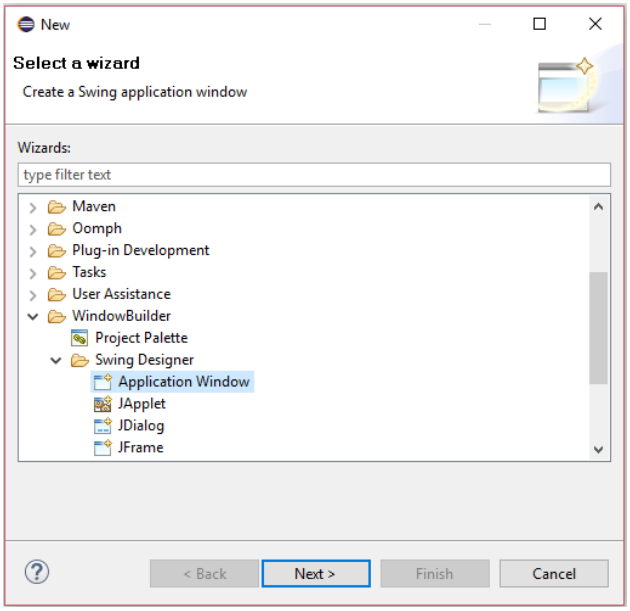


Figure 19: Adding a Window Step 2

After you select to add a new component, find the **WindowBuilder > Swing Designer** folder and the **Application Window** subfolder. Click the subfolder and click **Next**, as in Figure 19. This will open the New Swing Application window. You can also create a new project in this way from the file menu, instead of creating a Java application as we have done previously.

Note: Multiple window-building tools exist. If you are interested in IBM's SWT, you can also create windows using the SWT framework. This allows you to add multiple windows to your projects, and open and close them while the application runs.

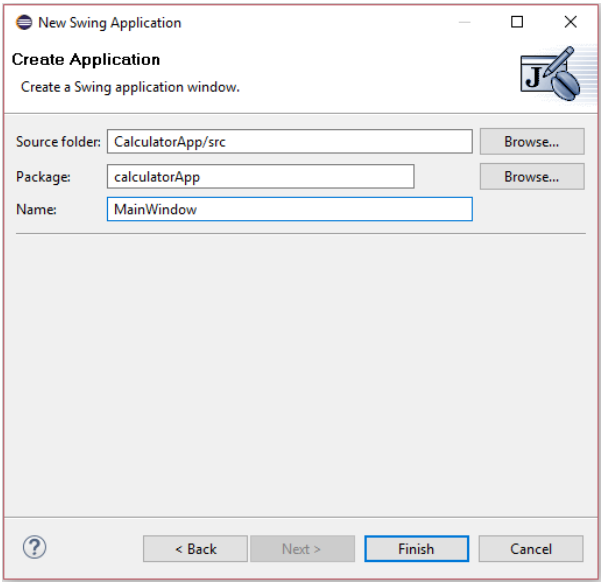


Figure 20: New Swing Application Window

In the New Swing Application Window, type a name for your **`main`** class and a package. I have used **`calculatorApp`** as my package name and **`MainWindow`** for the name of my **`main`** class (see Figure 20). Click **Finish** when you have given your package and project names.

Eclipse will use the WindowBuilder tool to create a new blank window. When we add a window using the Swing builder, it automatically writes a **`main`** method for us, which creates and shows the window using code similar to that used in the previous chapter. Code Listing 7.0 is generated by the Swing Window Builder. We can edit the code however we like, but be careful—this is automatically generated code, and in order for the WindowBuilder to continue to manipulate and add new code, this code must be kept somewhat close to the format that the builder prefers. In general, we do not alter the program-generated code, but rather we add to it and program around it.

Code Listing 7.0: Swing WindowBuilder Generated Code

```
package calculatorApp;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class MainWindow {

	private JFrame frame;

	/**
```

```
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					MainWindow window = new MainWindow();
					window.frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the application.
	 */
	public MainWindow() {
		initialize();
	}

	/**
	 * Initialize the contents of the frame.
	 */
	private void initialize() {
		frame = new JFrame();
		frame.setBounds(100, 100, 450, 300);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}

}
```

Designing a GUI in Design View

The code in Code Listing 7.0 does little more than supply a **`main`** method and create and show a blank window. In Eclipse, there are Source and Design tabs at the lower side of the main code window. These are two views of our new application, while the source view is the normal code window. Select the **Design View** tab and Eclipse will open the window for editing using the Swing WindowBuilder (see Figure 21).

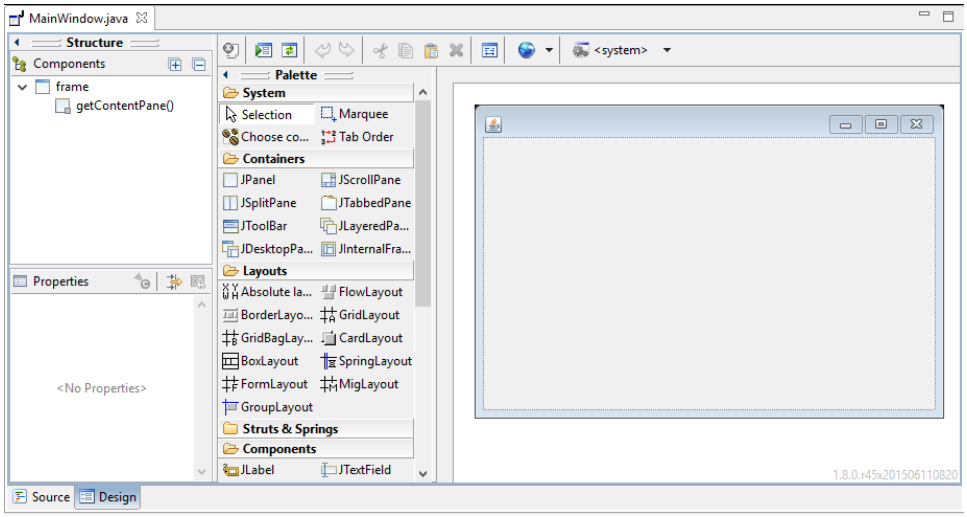


Figure 21: Switching to Design View

GUI design is a very large topic and a profession of its own. There are no strict rules to follow, but generally we should try to follow common conventions. Our users are familiar with thousands of layouts from their everyday lives—pocket calculators, food labels, office applications, etc. Users already understand these layouts, and if we want new users to understand our application with little effort, we should follow the principals of these established layouts. The gold standard for a new application is that a new user is able to use it without any instructions or manual at all. This is often not possible, but it is a very good standard to aim for, and something we should keep in mind whenever we design software.

Try drawing layouts using pen and paper or a drawing program and stylus. Even a project as simple as a calculator offers many options in terms of how we lay out our controls. Sketch out several possible layouts for the GUI, label them, and ask yourself why certain patterns in the layouts work while others do not. In Figure 22, I have taken inspiration from the layout of the CASIO fx series pocket calculator, which is very popular and well-designed—many people are familiar with its layout. We do not need to match the exact layout, and I have provided a space for “special function” buttons that will allow the functionality of our calculator to grow as we think of new features. I have also decided to include the arithmetic functions in a single column to save space, and I have increased the size of the Equals button because I believe this button is particularly important and will be used more often than any other button.

Note: We will not be programming expression parsing. Our calculator will compute the results of a single operation, and it will not include the use of parentheses. If you are interested in programming a calculator that is capable of parsing and computing the result of an expression, such as “4+2/(9*3)”, look up the Shunting Yard algorithm and Reverse Polish notation. The Shunting Yard algorithm converts an expression into Reverse Polish notation, and it is very easy to compute the result of a Reverse Polish notation expression. For the Shunting Yard algorithm, visit Wikipedia at https://en.wikipedia.org/wiki/Shunting-yard_algorithm. For more information on Reverse Polish, visit https://en.wikipedia.org/wiki/Reverse_Polish_notation.

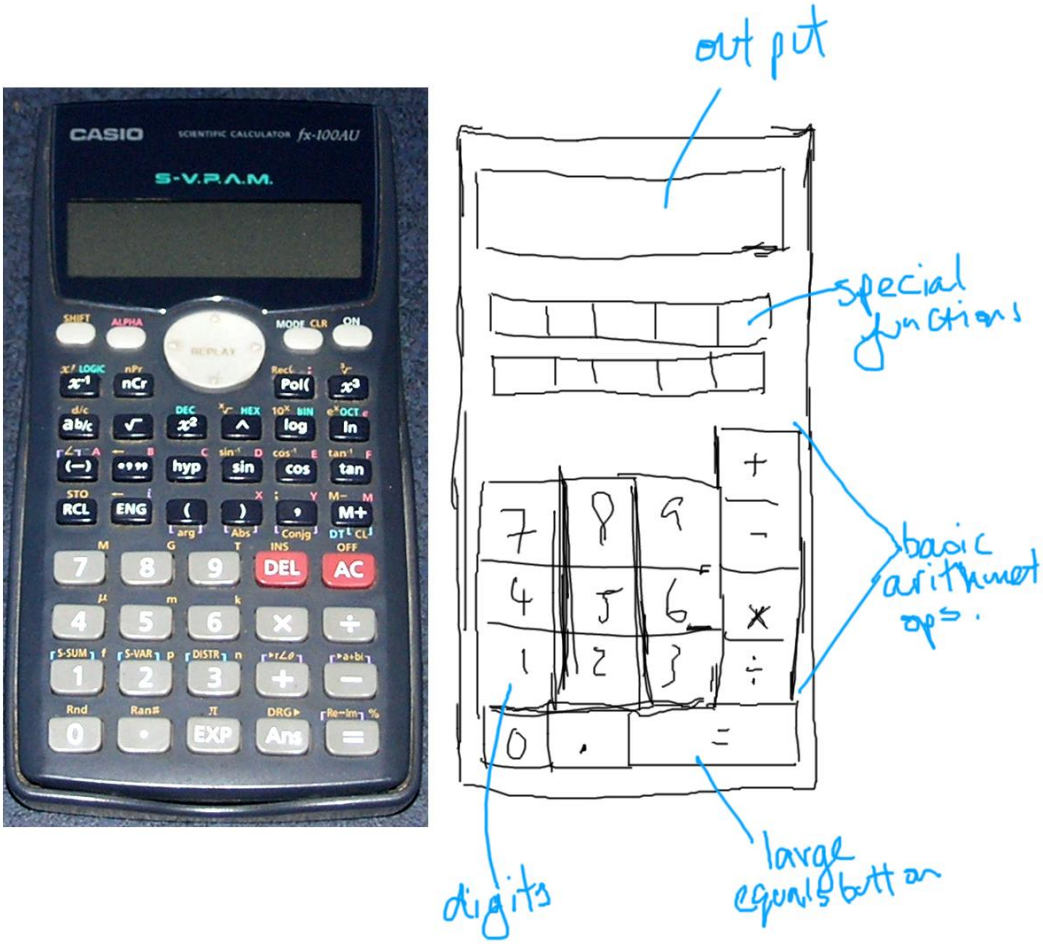


Figure 22: Designing a Calculator

Converting a design to Swing

The next step is to look at your GUI and decide how it will work in Java. Note that our form has a different aspect ratio—we want our calculator to be taller than it is wide. Select the form in the GUI designer along the edges and you will see small black square control points. Grab the lower-right control point and drag the form so that it roughly matches the aspect of your design (see Figure 23).

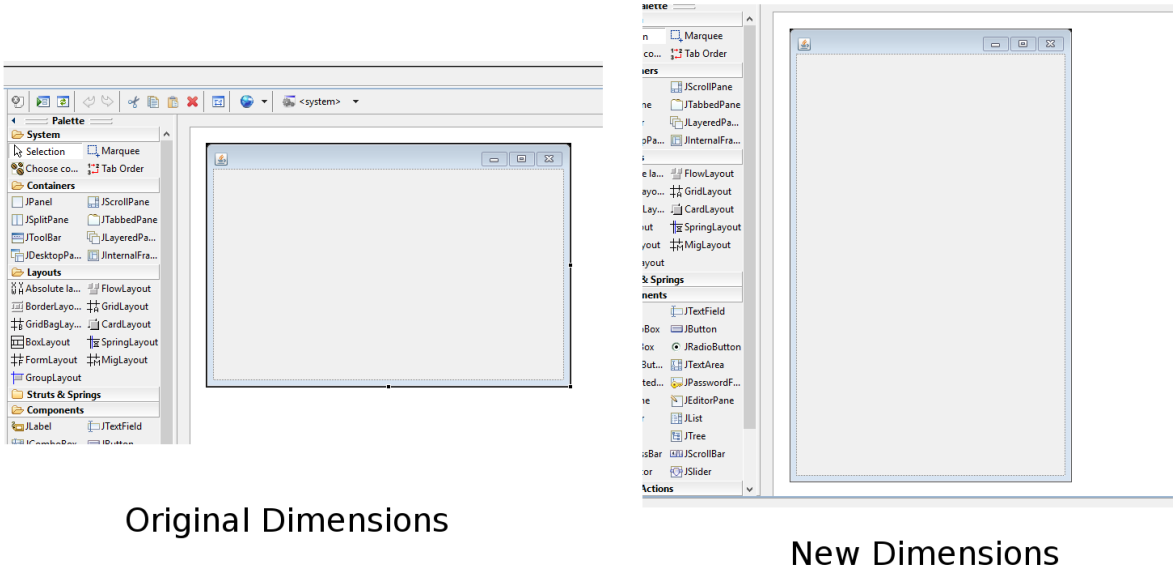


Figure 23: Resizing the Form

We will be nesting several layouts and using them together to provide some flexibility. The main structure of our calculator will be a border layout. Click **`BorderLayout`** in the layout's section of the palette, then click somewhere on the form (see Figure 24).

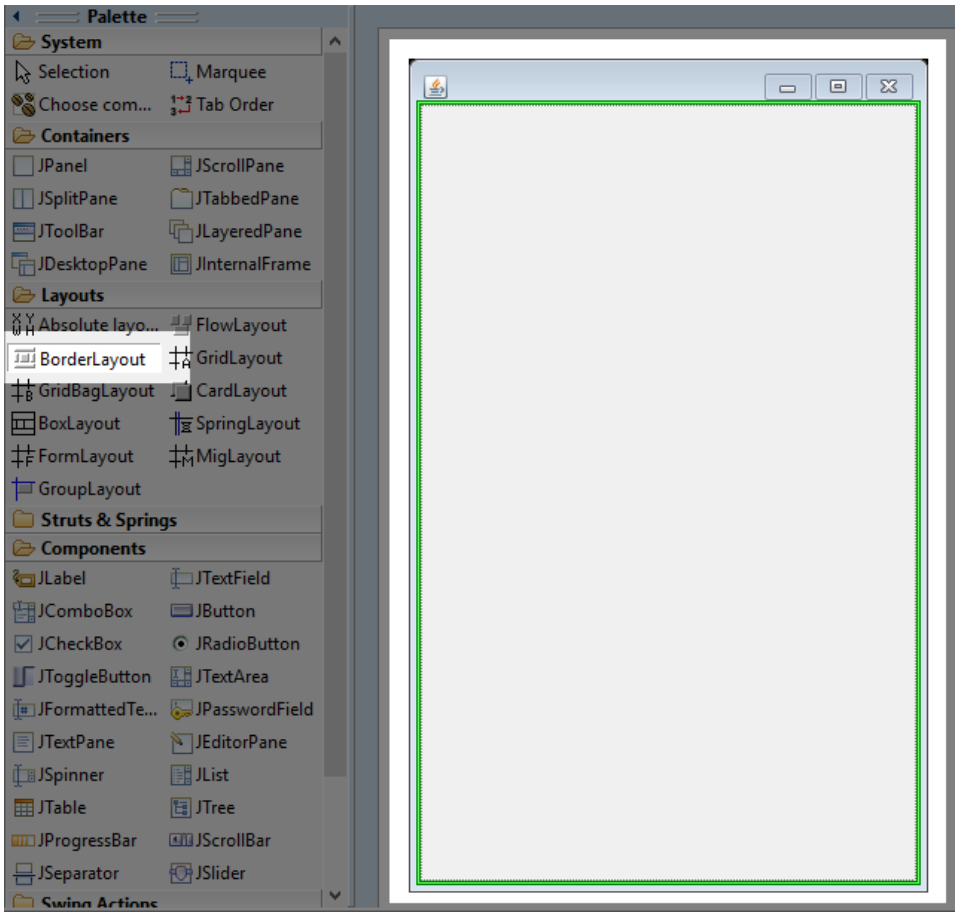


Figure 24: Selecting BorderLayout

Next, click the **`JTextField`** control in the Components section of the palette and click the North region of the form. This will add the **`JTextField`** to the top of the form. This will be the main output display of our calculator. The layout manager will automatically resize the **`JTextField`** to take up the width of the window (Figure 25).

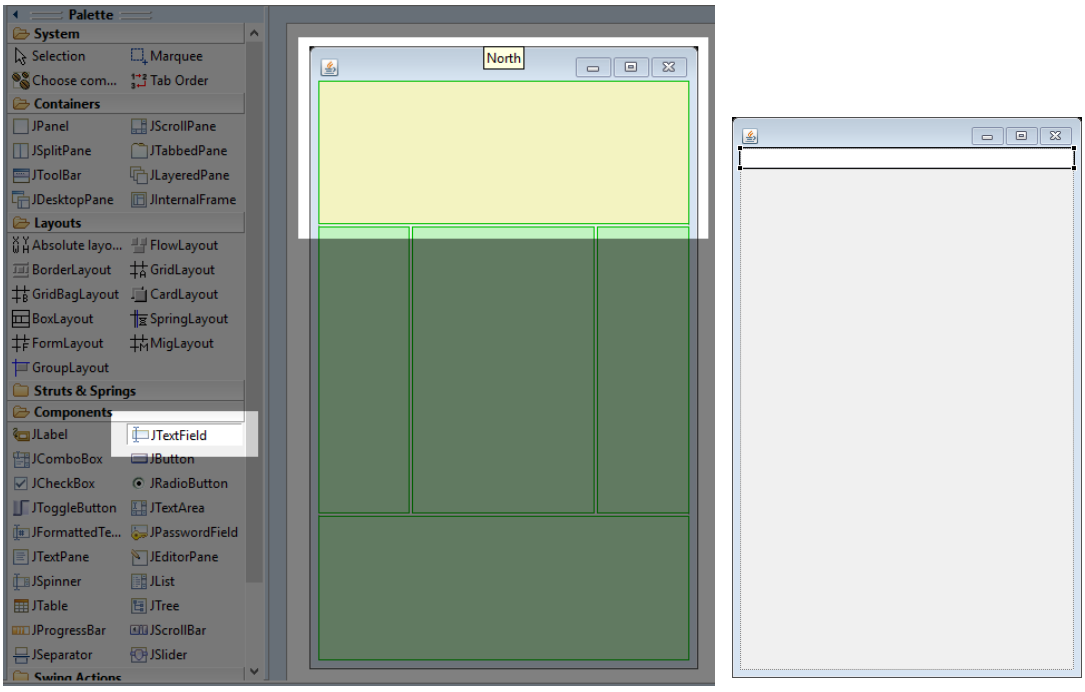


Figure 25: Adding a Text Field

When a control (or the form itself) is selected in the designer, the properties window shows the available settings for the control. Many aspects of the controls can be set through the properties box (see Figure 26). Make sure the **`txtOutput`** control is selected and change the Variable setting in the properties box to **`txtOutput`** (this is the name of the control). It is important to use logical control names, especially because projects often involve many controls.

Note: The "txt" prefix I use for my text field control is a reminder of the type of control. Adding a type prefix to the names of your controls helps speed up development and maintenance. If we type "txt" in the code, Eclipse's Content Assist will list all of the text fields—so long as we name all of them with the "txt" prefix. This means we do not need to remember the exact names of our controls, we only need to know their type. Content Assist will help us select the controls we need in our code. Likewise, we can name buttons with the "btn" prefix, check boxes with "chk" prefixes, etc. These prefixes are completely optional and represent a programming style choice.

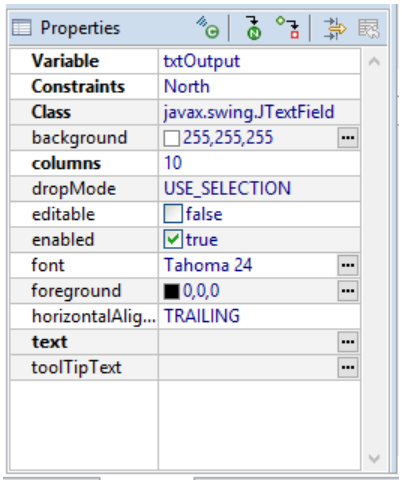


Figure 26: Properties Box

In the **`txtOutput`** control's properties, uncheck the Editable box. We will not allow our user to type expressions into our calculator. Select a font size of 24 and change the **`horizontalAlignment`** to "**`TRAILING`**"—this will make the numbers appear on the right side of the text field (which is how most calculators work). The final properties for the **`txtOutput`** control are in Figure 26.

Next, we will add a **`JPanel`** that will hold the buttons for our calculator. Select the **`JPanel`** container and add it to the center (as per the left side of Figure 27). Select a **`GridBagLayout`** and click the new **`Jpanel`** (as per the right side of Figure 27). This will create a **`GridBagLayout`** in the center portion of the **`BorderLayout`**. The **`BorderLayout`**'s components can, themselves, contain other layouts. This technique is called nesting—we have nested the **`GridBagLayout`** inside the **`BorderLayout`**. This allows us to combine layouts together in a complex and flexible way.

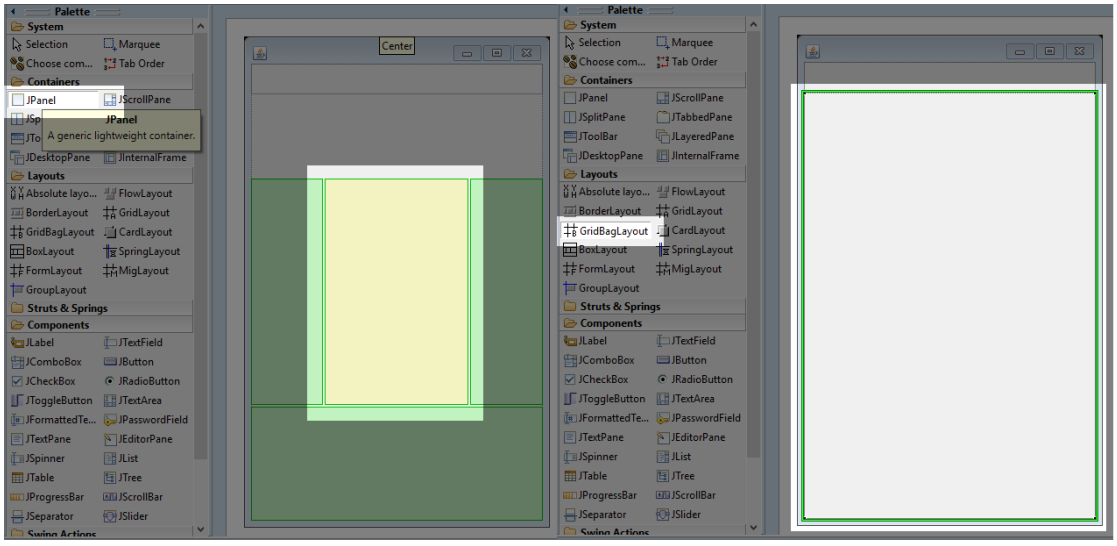


Figure 27: Adding a JPanel and GridBagLayout

Now that we have a **GridBagLayout**, click the **JButton** control and hover over the layout. You will see a guide showing the rows and columns of the **GridBagLayoutlayout**, as in Figure 27. Place your buttons to match Figure 28 (it does not matter if the exact placement of the controls is different for your calculator).

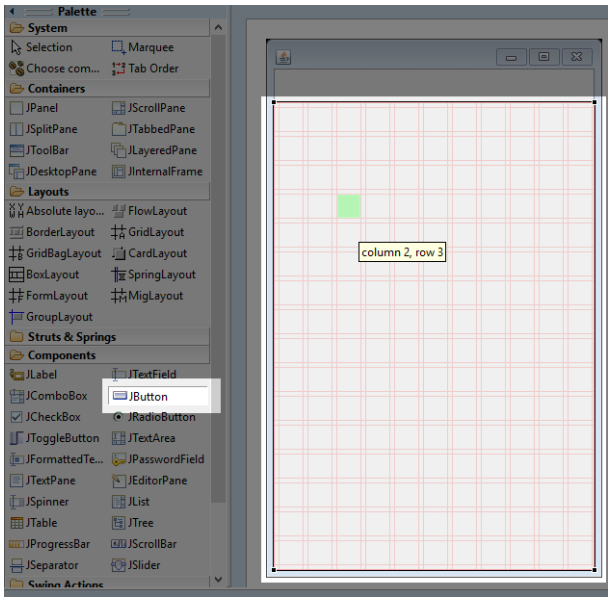


Figure 28: GridBagLayout Guide

Figure 29: Adding Buttons

Figure 29 shows the placement of all the buttons. You can change the text of a button when it is placed, or change it in the properties window. I have added twelve buttons labeled “f” in Figure

29 that will be special function buttons such as square root and trigonometry (the “f” is just a placeholder). Change all of the font sizes for these buttons to 24. You can select multiple controls by holding the shift key (the control key also works) while you click controls in the designer. Select all the buttons at once, then change the font size to 24 for all of them rather than changing the fonts one at a time. Click the **Equals** button and grab the control point on the right side. Move the control point two boxes to the right in the **`GridBagLayout`**. This will cause the button to consume three horizontal boxes of space, as in Figure 30.

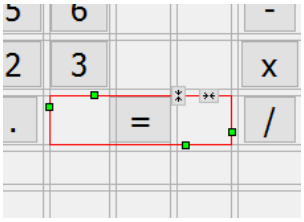


Figure 30: Resizing Controls

The Equals button takes up three boxes worth of space in the **`GridBagLayout`**, but presently the space is not filled. When a control is selected, we have several layout options in a toolbar at the top of Eclipse. Make sure your Equals button is selected and click **Fill**. This will cause Eclipse to resize the control to consume all the space of the three boxes in the **`GridBagLayout`** (see Figure 31).

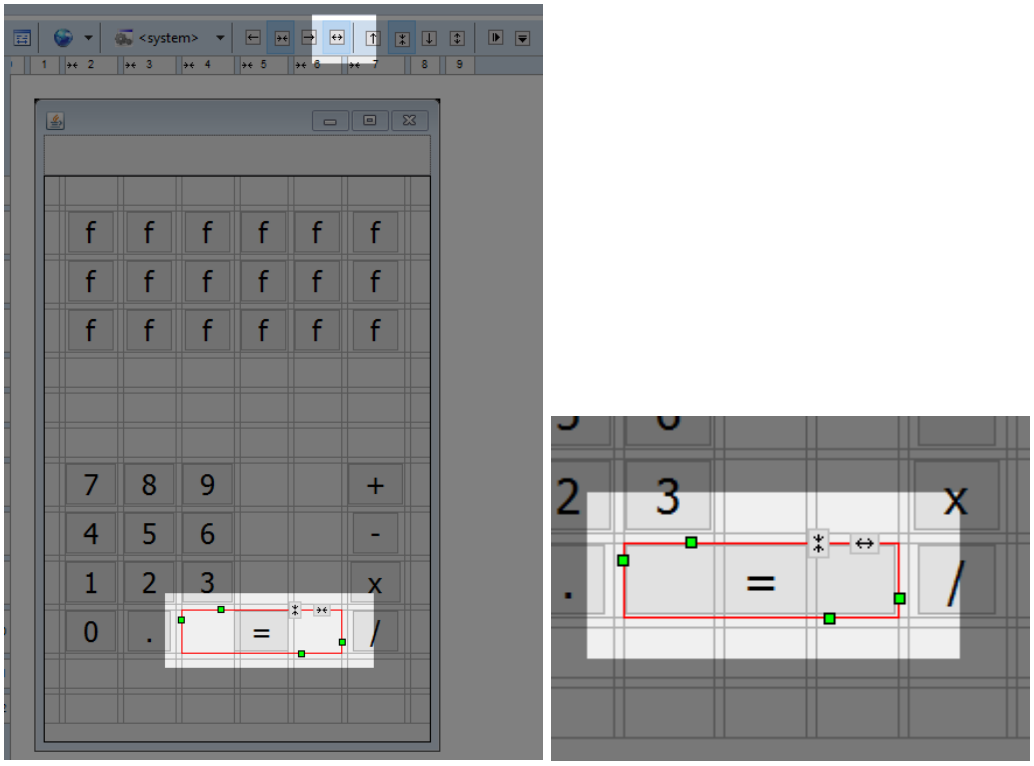


Figure 31: Filling a Control's Area

Quickly altering multiple controls

Our form has a lot of buttons. To quickly select them all, we can use the Components box in Eclipse. Click the first button's name in the Components box, hold down the shift key, then hold down the down arrow on your keyboard until all of the buttons are selected, as in Figure 32.

When all of your controls are selected, click **Fill**, exactly as we did with the Equals button. This will cause all of our controls to be resized to exactly the same size. Notice that before we click Fill, some of our controls are slightly different sizes, but default buttons are resized to surround their text (this is not true for all layouts), and the text in the buttons is not exactly the same width.

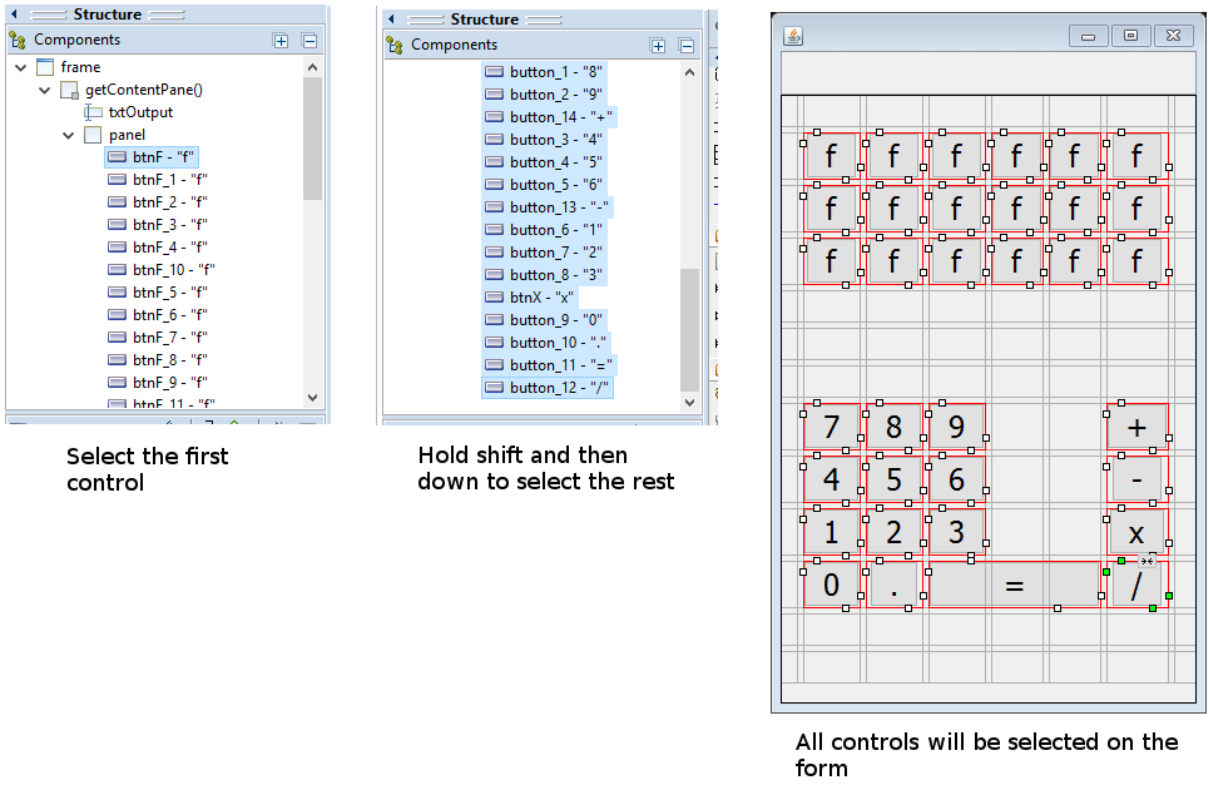


Figure 32: Selecting Multiple Controls in the Components Window

Spacing controls

When we run our program, we will see that the controls have very small gaps between them, despite what they look like in the designer (see Figure 33). We wish to place a small margin between the **`txtOutput`** control and our special functions, and we also want a small gap between the special function buttons and the digits and operator buttons.

In order to include gaps and margins in a **`GridBagLayout`**, we can set the sizes of the rows and columns. Click the **Selection** tool in the System palette, then click somewhere on your form that does not contain any buttons (i.e. select the **`GridBagLayout`**). You will see the grid has column and row headings numbered along the top and left side of the designer (see Figure 34).

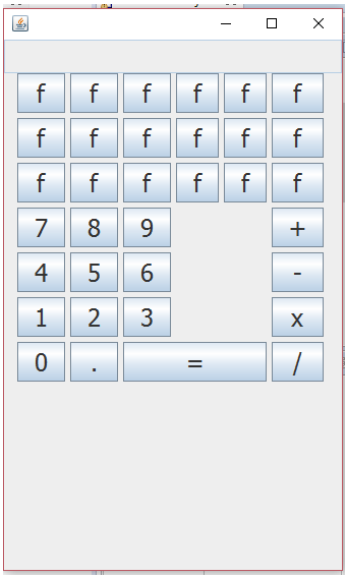


Figure 33: Our Calculator without Spacing

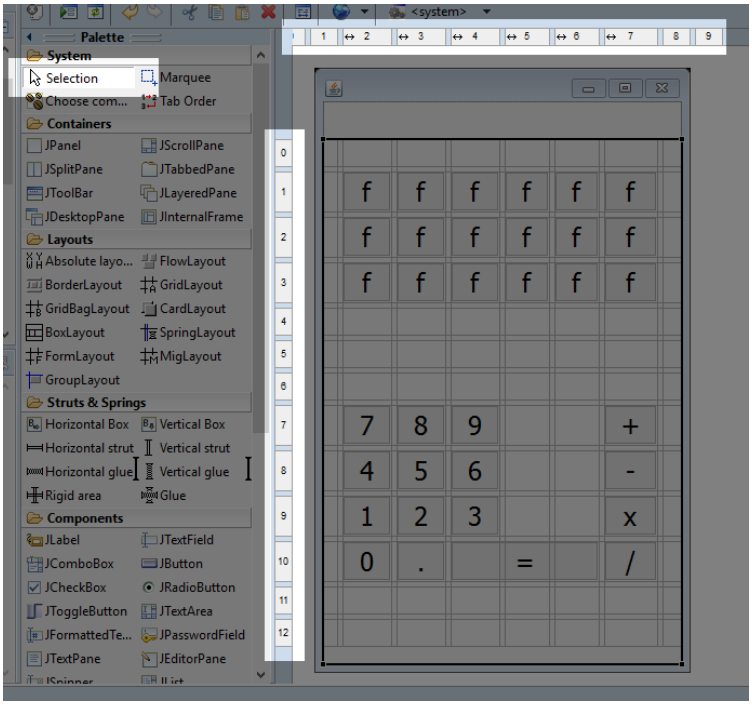


Figure 34: GridBagLayout Row and Column Headings

If you have a **`GridBagLayout`** selected, you can right-click on the appropriate row or column and perform some very useful functions. We can easily add new rows or columns, delete existing rows or columns, and change the attributes for any existing rows or columns. We want to change the minimum sizes of several rows so that our controls are better spaced. Right-click the button for row 0 and select properties from the Context menu (as per Figure 35).

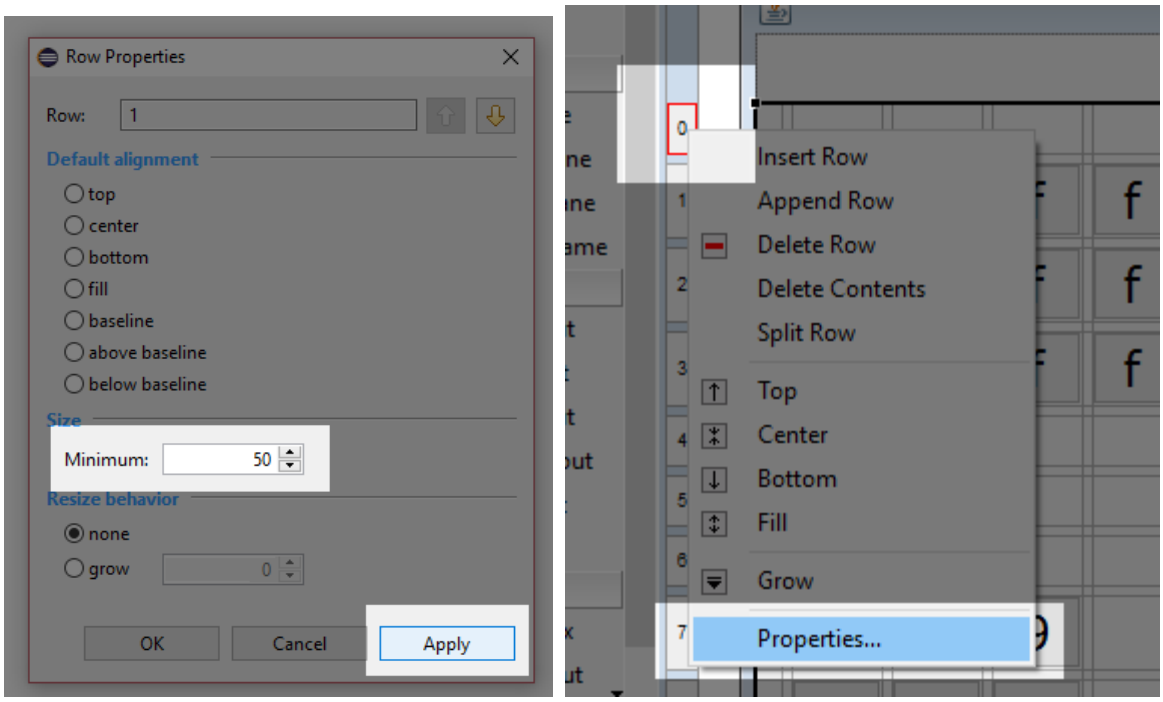


Figure 35: Row Properties

In the Row Properties box, change the minimum size to 50 and click **Apply**, as in Figure 35. This ensures a 50-pixel gap between the **`txtOutput`** control and our special function buttons. Next, change the minimum gap for row 5 in exactly the same way. This will add a gap of at least 50 pixels between our special function buttons and our Digit and Operator buttons. Notice that our calculator's spacing looks slightly better in Figure 36.

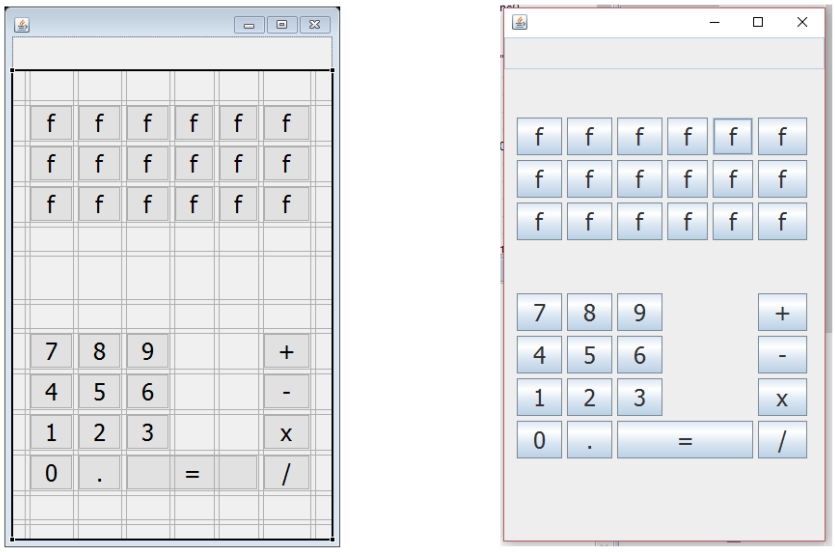


Figure 36: Spaced Controls

Now that we are happy with the general layout of our controls, we can proceed to add functionality to our calculator.

Adding functionality

Many arithmetic operators take two operands, such as 1+4. We will maintain a **`state`** variable that will record which number we are reading (i.e. the left or right operand for an arithmetic operator). We will also keep a variable that indicates which operation the user has initially selected, and we will instruct the calculator to read the digits of the first number (the left operand for the operator). We will build two types of operations—those that require two operands, such as Addition and Subtraction, and those that require a single operand, such as Natural Log and Square Root.

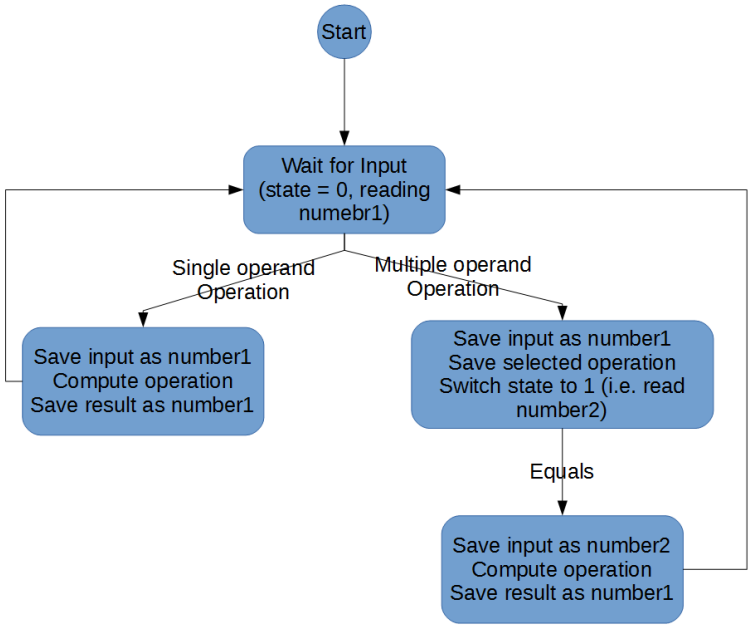


Figure 37: State Diagram

Figure 37 shows a basic state diagram, depicting the operational flow of our calculator. Initially, the **`state`** variable will be set to 0, which corresponds to reading the first operand (called **`number1`** in the following code). The user can input a number using the Digit buttons, then have two options—click a single operand operation (such as Square Root) or click a multiple operand operation (such as Addition).

When the user clicks a single operand operation, we can store the current digits that the user has clicked as **`number1`** and compute the result of the operation using this value. We can then store the result back to **`number1`**, output it to the screen, and start the process again.

When the user clicks a two-operand operation, actions become slightly more complicated. We store the current digits in the **`number1`** variable, we make a record of which operation the user has clicked (this is the **`operator`** variable in following code), and we set the **`state`** to 1, which means the program is reading the second value, **`number2`**. The user will input some new number and click Equals. When they click Equals, we store the current digits in **`number2`**, perform the selected operation between **`number1`** and **`number2`**, write the result to **`txtOutput`**, and start again.

Open the code view, scroll the code to the position where all the control member variables are specified (these should be the first member variables defined in the class). Add the four new **`state`** variables, as in Code Listing 7.1. I have not included the entire listing in Code Listing 7.1, but I have included a few lines to give context—the added code is highlighted in yellow. The code surrounding the highlighted code might be different, depending on the order that you added controls to your calculator, but this is fine. The **`state`** variables can be added in any place where member variables are specified. I have also added a small array of doubles called **`memory`**, which we will use to allow the user to store results later.

Code Listing 7.1: State Member Variables

```
	private JButton btnF_15;
	private JButton btnF_16;
	private JButton btnF_17;

	// State variables
	private int state = 0;         // 0 to first number, 1 for second.
	private int operator = 0;      // 0 means unknown.
	private double number1 = 0.0; // Variable for parsing 1st operand.
	private double number2 = 0.0; // Variable for parsing 2nd operand.
	private double[] memory = new double[5]; // For memory functions.

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					MainWindow window = new MainWindow();
					window.frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
```

Next, return to the design view and double-click on the 7 Digit button. When we double-click on a button, Eclipse will automatically write an event handler to handle the most common event for the control. In this case, it will write a handler for when the button is clicked and take us to the position in the code where we can specify what happens when the user clicks this button. The event handling code should look very familiar—Eclipse uses an anonymous class that implements the **`ActionListener`**. Again, the button in your code might not actually be called "**`button`**" in code. I added the 7 Digit button first (it could be called "**`button_7`**" or any other number in your code), so do not change the button name!

Code Listing 7.2: Clicking the 7 Digit

```
button = new JButton("7");
button.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent arg0) {
		txtOutput.setText(txtOutput.getText() + "7");
	}
});
```

When the user clicks a Digit button, we add the digit to the **txtOutput** box, as in Code Listing 7.2. The other nine digits are the same—except for the digits they add to the **txtOutput** string. In order to save space, I will not include a listing for all 10 digits, but you can implement all 10 digits in the same way as digit 7. Also, the Decimal Point button, ".", is the same as the Digit buttons, which means this can be implemented in the same way, too. If we were making a more complete calculator, we would need a check to ensure that we have input a valid number, and we would need to ensure that, at most, one decimal point is used to avoid numbers such as 7.6.4.

The two-operand operator buttons need to update the currently selected **operator** variable and reset **txtOutput** so that the user can input another number. In the design view, double-click the addition operator and add the code in Code listing 7.3 that is highlighted in yellow.

Code Listing 7.3: Code for Addition

```
button_14 = new JButton("+");
button_14.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		operator = 0;
		number1 = Double.parseDouble(txtOutput.getText());
		txtOutput.setText("");
	}
});
```

The three other arithmetic operators can be added in the same way, except that the operator variable should be set to different values. I will use 0 for addition, 1 for subtraction, 2 for multiplication, and 3 for division. I will not include the code for subtraction, multiplication, or division, but you can go add these operations in exactly the same way as addition, except that you must assign the integers **1**, **2**, and **3** for the **operator** variable. You can also specify an enumeration of operators. This would make the code clearer.

When the user hits the Equals button, we need to read the digits for **number2**, check the current **operator**, and perform the operation using the two numbers we have read. Then we store the resulting number as the current string in **txtOutput**. Code Listing 7.4 shows the code for the Equals button.

Code Listing 7.4: Equals Button

```
button_11 = new JButton("=");
button_11.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		// Read number 2:
		number2 = Double.parseDouble(txtOutput.getText());

		// Result variable
		double result = 0.0;

		// Compute the result based on the operator:
		switch (operator) {
		case 0: result = number1 + number2; break;
		case 1: result = number1 - number2; break;
		case 2: result = number1 * number2; break;
		case 3: result = number1 / number2; break;

		default: result = 0.0; break;
		}

		// Save the result to the output.
		txtOutput.setText("" + result);
	}
});
```

Tip: Note the use of the "" in the final call to txtOutput.setText in Code Listing 7.4. This causes the double to be converted to a string. If we try to setText or println and pass a double, such as println(2.5), our code will not compile. However, the addition operator is defined between strings and doubles, and it automatically converts the double to a string, so that instead of println(2.5) to print a double, we use println("" + 2.5).

Special functions

Tool tips and the clear button

We have included many special function buttons on our calculator, but I will specify only a few and let the reader design the functionality of the others. First, we need the ability to clear the output. I have made the lower-right special function button into a clear button by using the text “C.” The letter C is ambiguous, so I will also use a tool tip for this button. Tool tips are messages shown when users hover their cursor over the control. Change the text of the button to C and add the tool tip, as shown in Figure 38. Double-click the clear button to have Eclipse write the code for raising the event. Code Listing 7.5 shows the code.

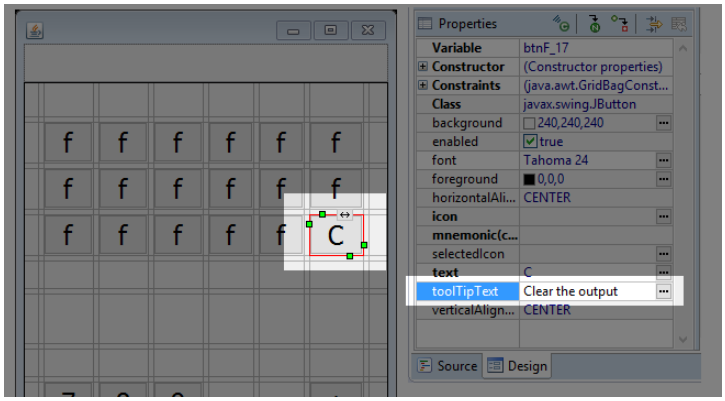


Figure 38: Clear Button Design

Code Listing 7.5: Clear Button

```
btnF_17 = new JButton("C");
btnF_17.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent arg0) {
	txtOutput.setText("");
	operator = -1;
	}
});
```

Trigonometry

Trigonometry functions are very useful for a calculator. I will add sine, cosine, and tangent function buttons as the three top-left buttons. In the designer, I have added tool tips and changed the font size of the buttons so that the words SIN, COS, and TAN fit onto our small buttons (see Figure 39). I have also clicked the Fill Vertical button above the designer so that the buttons are resized to fill the vertical area.

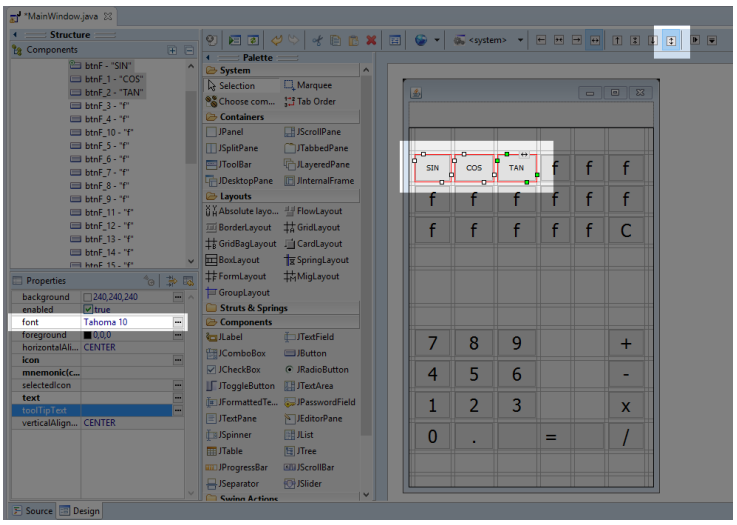


Figure 39: Sin, Cos, and Tan Designer

Code Listing 7.6: Sine Function

```
btnF = new JButton("SIN");
btnF.setToolTipText("Compute the Sine of an angle");
btnF.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
	number1 = Double.parseDouble(txtOutput.getText());
	number1 = Math.sin(number1);
	txtOutput.setText("" + number1);
	}
});
```

Code Listing 7.6 shows the code for computing the sine of an angle. The highlighted code shows the actual computation. The other trig functions are similar, except for the cosine we call **`Math.cos`** and for the tangent we call **`Math.tan`**. Note that computing these trigonometry functions requires only a single number, which means we read **`txtOutput`**'s text and compute the functions without waiting for the user to input a second number.

Raising a number to a power

We can easily add new two-operand functions to our calculator in the same way we added our original arithmetic operators. Here is an example that shows how to add a power function that takes two parameters and raises the first parameter to the power of the second. I used the text "^" for my power button in the designer (see Code Listing 7.7) and called the function **`Math.pow`** when the user clicked Equals (see Code Listing 7.8). I have called the power operation **`operator 4`**. Each time we add a new operator, the line "**`operator = xxx`**" must be unique, so the next two-operand operator would be called **`operator 5`**, and after that **`6`**, etc.

Code Listing 7.7: Power Button

```
btnF = new JButton("SIN");
btnF_3 = new JButton("^");
btnF_3.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent arg0) {
		operator = 4;	// 4 is power
		number1 = Double.parseDouble(txtOutput.getText());
		txtOutput.setText("");
	}
});
```

Code Listing 7.8: Equals Button with Power Operator

```
button_11 = new JButton("=");
button_11.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		// Read number 2:
		number2 = Double.parseDouble(txtOutput.getText());
```

```
                    // Result variable
                    double result = 0.0;

                    // Compute the result based on the operator:
                    switch (operator) {
                    case 0: result = number1 + number2; break;
                    case 1: result = number1 - number2; break;
                    case 2: result = number1 * number2; break;
                    case 3: result = number1 / number2; break;

                    default: result = 0.0; break;
                    }

                // Special Functions
                if(operator == 4)result = Math.pow(number1, number2);

                    // Save the result to the output
                    txtOutput.setText("" + result);
                }
            });
```

Memory buttons

Memory buttons are very useful for a calculator. I will add five memory buttons called M1, M2, M3, M4, and M5. They will be used to store and recall numbers in the **memory** array that we defined earlier.

First, when we click a memory button, we need to know if the user wants to store a value or recall one. By default, we will assume they are recalling a number. We will add a store button with the text STR so that the user can click store followed by a memory button, and, instead of recalling the value, we will store the current **txtOutput**. Add a button with the text **STR** for store and five buttons with the text **M1**, **M2**, **M3**, **M4**, and **M5** (see Figure 40).

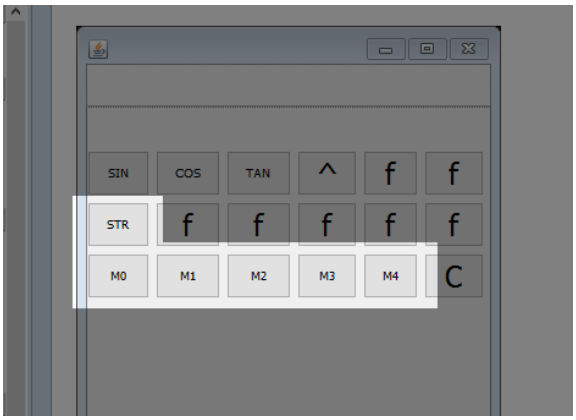


Figure 40: Memory Buttons

When the user clicks the **STR** function, we set the current **operator** to some new operator code. Our last **operator** was power, which we saved as **operator** 4, so I have used **operator** 5 as the Store operation in Code Listing 7.9. The code for each of our memory buttons is almost identical, except that each button accesses a different element from our **memory[]** array. I have included the code for **M0** in Code Listing 7.10.

Code Listing 7.9: Store Button Code

```
btnF_5 = new JButton("STR");
btnF_5.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		operator = 5;
		number1 = Double.parseDouble(txtOutput.getText());
	}
});
```

Code Listing 7.10: M0 Button Code

```
btnF_12 = new JButton("M0");
btnF_12.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		if(operator == 5)
			memory[0] = number1;
		else
			txtOutput.setText("" + memory[0]);
	}
});
```

At this point, I will stop and allow you to expand the functionality as you see fit. In this chapter, we have looked at how to implement a simple interface using the window designer tools in Eclipse. Our humble calculator is fairly straightforward, but the power and flexibility of these GUI designer tools is virtually limitless. You can add buttons for arc-sine, logarithms, and many other useful operations. We have looked primarily at buttons and text, but there are many other interesting controls available, and if you are new to GUI design or the windows builder tools, I encourage you to explore the tool palette more deeply in order to find the controls that best suit your applications.

Note: Many techniques have been developed for writing and maintaining large-scale projects. One of particular interest is MVVM, which stands for Model-View-View-Model. Using MVVM as a design principle, we intentionally split all of the functionality for our program from the code for generating the GUI. This helps maintain a clear distinction between the functionality and the GUI, and it can help scalability and maintainability of large-scale projects. Our calculator is small enough that MVVM design principles would be of little consequence, but the topic is worth reading up on. If you are interested in developing larger-scale GUI projects, visit https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel.

For the remainder of this e-book, we will turn our attention to a completely different type of programming—2-D graphics and game programming.

Chapter 8 2-D Game Programming

Java is frequently used in 2-D game programming. In this section, we will look at the basics for creating 2-D games using Java. Game programming is an extremely broad and complex topic, so we will address only some of its key concepts here. Practice is the key—you should take this final chapter and run with it, create a platformer or an endless runner, and explore 3-D game development and physics engines.

In this chapter, I will use larger listings of code, but I will supply only a comparatively small amount of explanation. One of the finest skills any programmer can develop is the ability to read other programmers' code and to see where they are wrong (or could be improved upon). I encourage all folks who are new to programming to scour the Internet for useful techniques. When you find a useful technique, or snippet of code, make sure the original author has allowed you to use it for your own projects, and always make a record of where you got the code—that way you can credit the original author if you ever use the code in production programming. Feel free to use any of the code in this e-book for whatever purpose you like!

MainClass

There are many ways to set up a foundation for 2-D game programming. We want our code to be maintainable, easy to understand, and quick to implement new features. The difficulty in programming is not learning the syntax—an experienced programmer can learn the syntax to a new language relatively easily. The difficulty is employing structures in such a way that our projects remain stable as the project increases in size. We will create a basic **`MainClass`** that does little more than run an instance of another class—the game's engine, which will be called **`Engine2D`**. The engine will run with a simple render/update loop, and we will use Java's timing facilities and event handlers to create the illusion of real-time and to respond to the keyboard.

You should next create a new project. I have called my project **`Graphics2D`**. Add a **`MainClass`**, exactly as we have done previously. The **`MainClass`** will do little more than run an instance of the engine class. The code for the **`MainClass`** is presented in Code Listing 8.0 (please note this code will not run at this point because we have not built the **`Engine2D`** class yet!).

Code Listing 8.0: MainClass

```
import java.awt.EventQueue;
import javax.swing.JFrame;

public class MainClass extends JFrame {
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			@Override
			public void run() {
				MainClass m = new MainClass(640, 480);
			}
```

```
            });
      }

      private MainClass(int windowWidth, int windowHeight) {
            setSize(windowWidth, windowHeight); // Set window size
            setLocationRelativeTo(null); // Default location
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Exit on
close
            setVisible(true);
            // Create and add the engine JPanel.
            final Engine2D engine = new Engine2D(windowWidth, windowHeight,
30);
            add(engine);
      }
}
```

Our **MainClass** extends the **JFrame** and will act as an application window. The **MainClass** contains a **main** method that creates a new instance of **MainClass** called **m** and that executes the instance using a new thread using the **EventQueue.invokeLater** method. This means our game will have its own thread and event queue. The constructor for the **MainClass** takes **windowWidth** and **windowHeight** arguments that will be the size of our window. After setting up the window, the **MainClass** constructor creates an **Engine2D** instance that is essentially a customized **JPanel** object, and it adds the panel to the frame's controls using the **add** method.

2-D game engine skeleton

Next, we will implement a new class called **Engine2D**. This class represents the main backbone of our games. It will handle the updating and rendering in our application. In order to render 2-D graphics, we need a control to render to, so the **Engine2D** class extends the **JPanel**. Code Listing 8.1 shows the blank skeleton of the new **Engine2D** class.

Code Listing 8.1: Engine2D Skeleton

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JPanel;
import javax.swing.Timer;

public class Engine2D extends JPanel implements ActionListener {
      // Width and height of the window.
      private int width, height;

      // Constructor
      public Engine2D(int windowWidth, int windowHeight, int fps) {
```

```
            width = windowWidth;
            height = windowHeight;
            Timer timer = new Timer(1000/fps, this);
            timer.start();
      }

      // This event is called when the timer fires at the specified fps.
      public void actionPerformed(ActionEvent e) {
            update(0.0, 0.0);
            repaint();
      }

      //
      public void paintComponent(Graphics g) {
            super.paintComponent(g);
            render((Graphics2D) g);
      }

      private void update(double timeTotal, double timeDelta) {
      }

      private void render(Graphics2D g) {
            // Clear the screen to blue.
            g.setBackground(Color.DARK_GRAY);
            g.clearRect(0, 0, width, height);
      }
}
```

One of the simplest and most common ways to implement a game engine is through the use of an update/render real-time game loop. We use a **`timer`** to repeatedly call two methods—**`update`** and **`render`**. We call the methods once for every frame of the game. The method calls can be seen in the code of Code Listing 8.1 in the **`actionPerformed`** method. First, we call the **`update`** method, in which we will compute the positions and logic of all of the objects in our game's world. Next, we call the **`render`** method, in which we render a pictorial version of our objects so that users have something to look at while they play (see Figure 41).

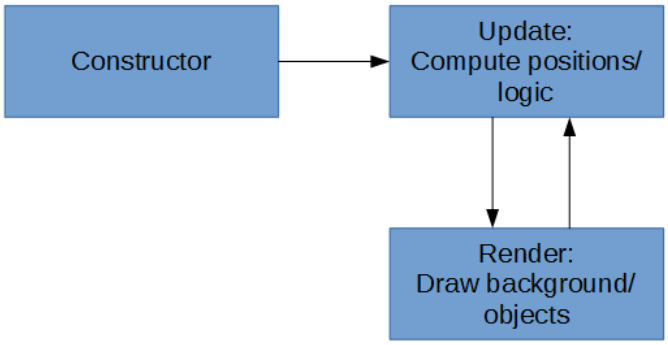


Figure 41: Real-Time Game Loop

In order to use a real-time game loop, we want our **update** and **render** methods to be called repeatedly at some specific interval. The interval is called the frames per second (FPS). A higher frame rate (60 or 100 FPS) will look smoother but will consume more power. If the frame rate is too high, the animation may become jerky as the processor falls behind and skips frames. A lower frame rate (12 or 16 FPS) does not look so smooth, but it consumes less power. There is a good chance that a lower frame rate can be rendered by the processor without skipping frames. One of the parameters to the constructor of our **Engine2D** is the frame rate. I have used 30 FPS for this value, which should look relatively smooth and should run without consuming too much power on portable devices (and thus conserve some battery for the players of our game).

We have employed a **timer** to call the **actionPerformed** method once per frame, and we have implemented the **ActionListener** class. If we perform too much computation in our **update** method, or if we attempt to render too many sprites in our **render** method, we might not achieve the desired frame rate. Each time the **timer** ticks, the **actionPerformed** method will be called, which calls **update**, followed by **paintComponent** (which calls **super.paintcomponent** to refresh the window), and **render**.

Note: Frame rates alone do not make animations look smooth. Even at a very high frame rate, an animated object will not appear smooth to human eyes. The true key to creating smooth animations is to employ a technique called motion blur. That topic is outside of the scope of this e-book, but you should visit the page http://www.testufo.com/ for some fantastic examples of how motion blur works and the effects of animation when implemented correctly.

At the moment, the update method has two parameters, **timeTotal** and **timeDelta**, that do nothing. And the **render** method simply clears the screen to **DARK_GRAY**. After adding the **Engine2D** class, you should be able to test your application. If you do not see a dark gray screen, as in Figure 42, something has gone wrong.

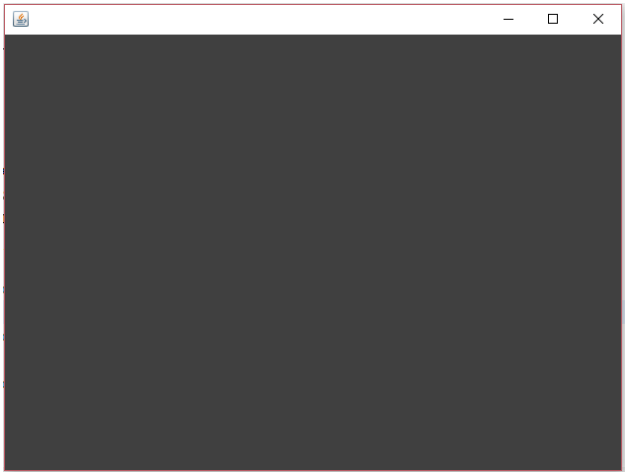


Figure 42: Clearing the Screen

Sprite sheet class

When we animate, we often draw images of our objects in quick succession that are slightly different from each other. For instance, an animation of the player walking might consist of eight frames, each slightly different from the last. We could store each image of our animation in a separate image file, but storing all the images of an animation in a single file is often more convenient. Such image files are called sprite sheets or sprite atlases.

There are many image formats—BMP, PNG, JPG, TIFF, etc. Each format is designed for specific purposes. PNG is the format of choice for sprites because it is compressed, is generally smaller than a bitmap, and it allows alpha transparency. And, unlike JPG (which is also compressed), the compression used for PNG images is lossless, which means we retain the exact values of every pixel we draw in our frames.

Code Listing 8.2: Sprite Sheet Class

```
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class SpriteSheet {

	private BufferedImage bmp; // The loaded image.
	private int spritesAcross = -1;// Number of sprites across the image.
	private int totalSprites; // Total sprites in the image.
	private int spriteWidth, spriteHeight;// width/height of each sprite.

	// Constructor
	public SpriteSheet(String filename,int spriteWidth,int spriteHeight)
{
		// Load the bitmap.
		try {
			bmp = ImageIO.read(new File(filename));
		}
		catch(IOException e) {
			// File not found.
			spritesAcross = -1;
			return;
		}
		// Save the sprite width and height.
		this.spriteWidth = spriteWidth;
		this.spriteHeight = spriteHeight;
		// spritesAcross is used to compute the
		// source rectangles when rendering.
		spritesAcross = bmp.getWidth() / this.spriteWidth;
		// totalSprites is used to ensure we're not
		// trying to render sprites that do not exist.
```

```
            totalSprites = spritesAcross * (bmp.getHeight() /
spriteHeight);
      }

      // This method can be used to test if the sprites loaded.
      public Boolean isValid() {
            return spritesAcross != -1;
      }

      public void render(Graphics2D g, int spriteIndex, int x, int y) {
            // Make sure the sprite is actually on our spriteSheet.
            if(spriteIndex >= totalSprites) return;
            // Compute the source x and y.
            int srcX = (spriteIndex % spritesAcross) * spriteWidth;
            int srcY = (spriteIndex / spritesAcross) * spriteHeight;
            // Draw the image.
            g.drawImage(bmp,
                        x, // Destination x1
                        y, // Destination y1
                        x + spriteWidth, // Destination x2
                        y + spriteHeight, // Destination y2
                        srcX, // Source x1
                        srcY, // Source y1
                        srcX + spriteWidth, // Source x2
                        srcY + spriteHeight,// Source y2
                        null); // Observer
      }
}
```

Code Listing 8.2 shows a simple sprite sheet class. The member variables are a buffered image (which is simply a method for storing an image loaded from the disk in RAM for quick access) and several simple records—**spriteWidth**/**spriteHeight**, **totalSprites**, and **spritesAcross**. The class takes a filename in the constructor and a width and height for the sprites. The file can be any standard 2-D image format, but we will use PNG.

GNU image manipulation program (Gimp)

I created the image for our test application for our sprite sheet by using Gimp (which is a very powerful drawing and photo manipulation program, available free from https://www.gimp.org/). The sprites in Figure 43 show a spaceship, a nasty-looking space critter, a bullet, a green wall, some stars, and a small explosion.

Note: I have used Gimp for creating my sprites in this text because it is popular, cross-platform, and powerful. Many other applications that readers might want to explore are also available. Piskel is an online sprite creation tool available from http://www.piskelapp.com/. Asesprite is an excellent, small-desktop sprite editor available from http://www.aseprite.org/. Spriter by BrashMonkey is an excellent sprite editor with free and paid versions available from https://brashmonkey.com/.

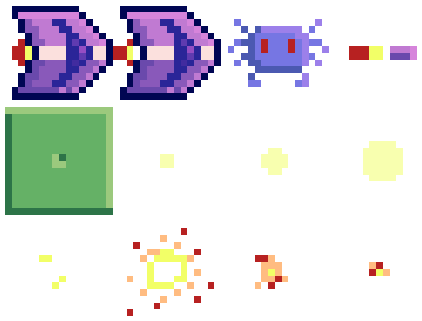


Figure 43: Sprites

Tip: When we draw an image in Gimp (or some other image editor), we do not want to lose any precision—we want to specify the exact color of every pixel. For this reason, I recommend that when you work on the image, save it in Gimp's specialized format (which includes extra information for layers, pixel colors, masks, paths, etc.). When we come to use the image in our game, we export it as a PNG image (the PNG does not have multiple layers, paths, etc.—it only contains pixel color data). This way, the layering and selection information available in Gimp will be maintained if you need to edit the image further, and you will have all the flexibility of PNG in the final exported image.

Right-click Figure 43 and click **Copy**, then paste the image as a new image into Gimp (depending on the PDF reader you use, the details of this operation might differ slightly). Open Gimp (or some other image editor of your choice) and select **New Image** under **Paste as**, as per Figure 44.

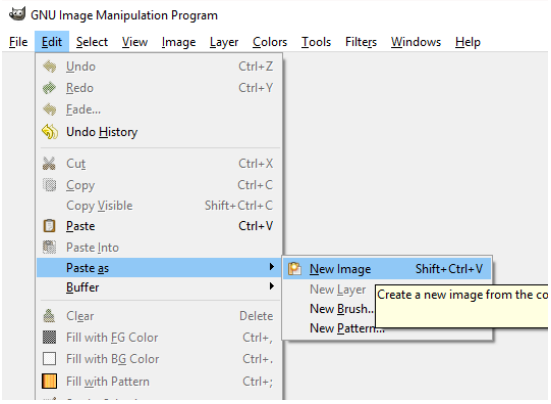


Figure 44: Paste as New Image

The copied and pasted image should look similar to the one pictured in Figure 43, except that the image will have black around the sprites where they are meant to be transparent (see Figure 45).

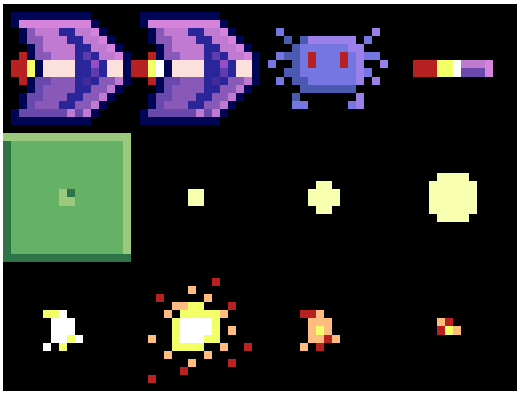


Figure 45: Sprites with Black Background

In order to change the background back to transparent, we need several windows open in Gimp. Choose the **Windows** item from the menu and open a **Toolbox**, **Tool Options**, and **Layers** window, as per Figure 46.

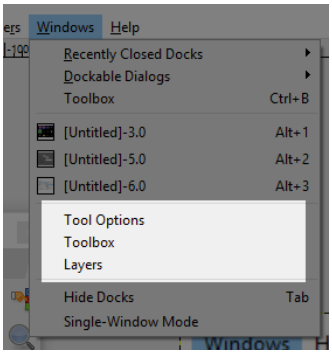


Figure 46: Gimp Windows

Next, we will add an Alpha Channel to the image. Right-click the layer in the Layers box and select **Add Alpha Channel** from the context menu (Figure 47). The Alpha Channel is used for transparency—we want the pixels outside of our animation frames to be transparent, so that there is not a white or black box around each sprite.

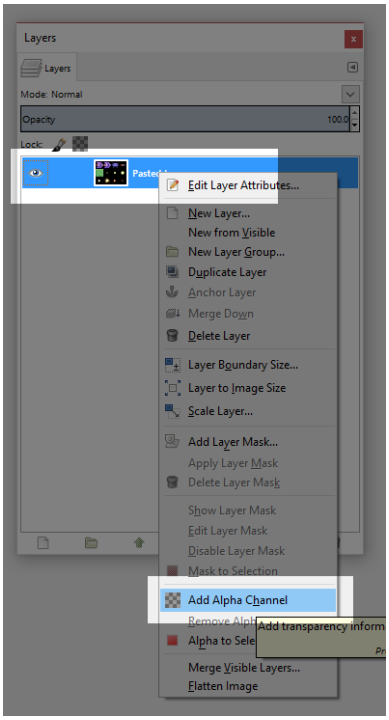


Figure 47: Add an Alpha Channel

Next, select the entire black region in order to remove the black boxes from our pasted image. There are several ways to do this, and I will use the Select by Color tool, as in Figure 48.

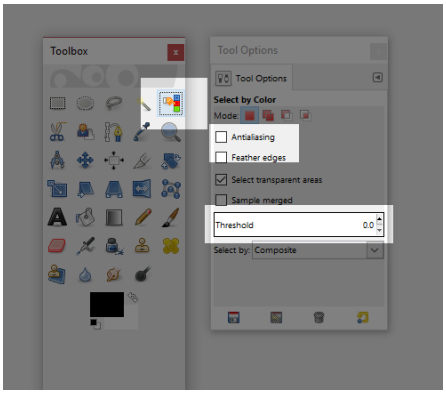


Figure 48: Select by Color

When you select by color, the Tool Options window is updated to include controls for manipulating how the selection should be performed. Turn off Anti-aliasing and turn off Feather Edges in the Tool Option box, as per Figure 48. Anti-aliasing and feather edges help selections appear smoother by adding slightly transparent edges to the selection—we do not want this. Make sure the Threshold is 0 so that we can select all the black regions without accidentally selecting nonblack but dark regions. After you have set up the Select by Color tool, click somewhere on the sprite's black region. The Select by Color tool will select all the matching black pixels. Hit the delete key on your keyboard in order to delete the black pixels—doing so will replace them with transparent pixels (pixels with an alpha value of 0), as per Figure 49.

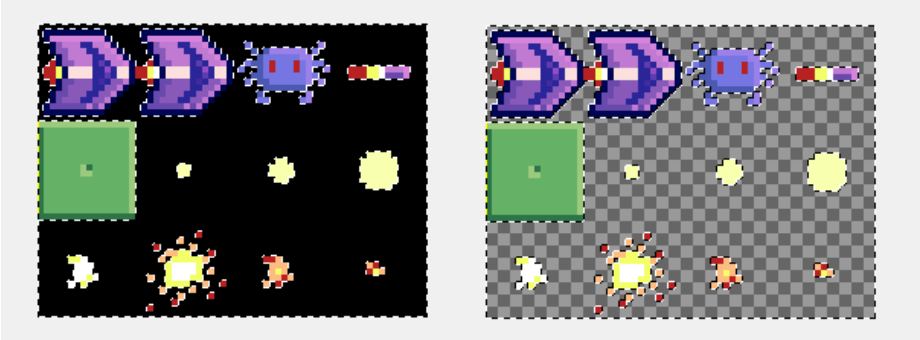


Figure 49: Deleting the Black Pixels

Next, export your image to the desktop (or some other place where it is easily accessible) by selecting **File** > **Export As…** and typing the name “spaceracer.png”.

Including an image in Java

We want to include this exported PNG sprite sheet in our Java application. Back in Eclipse, right-click your project in the Package Explorer and select **New** > **Folder**, as per Figure 50.

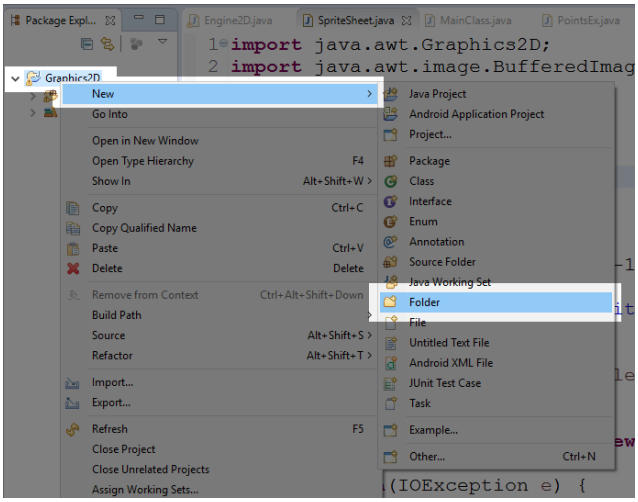


Figure 50: Adding a New Folder

Ensure the correct parent folder is selected (**`Graphics2D`** in my case), name the folder **`graphics`**, and click **Finish**, as per Figure 51.

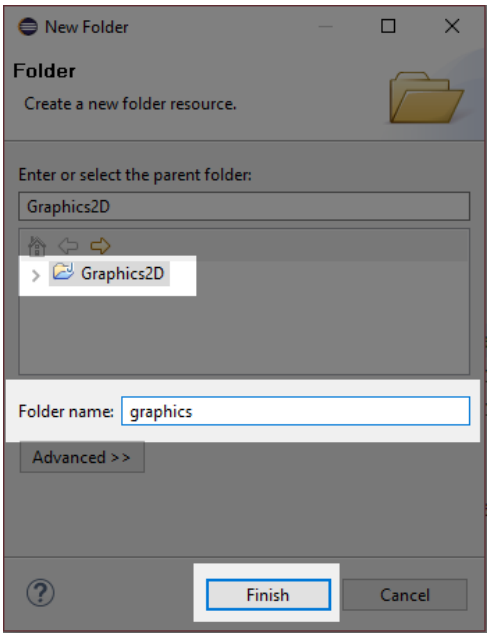


Figure 51: New Folder

Eclipse will create a new folder in your project. Copy the **`spacerace.png`** file that we exported earlier into this folder so that we can open it when our application runs. In order to copy a file to the project's folder, we need to open the folder in the System Explorer (this is simply the normal Windows file explorer). We could find the folder using the System Explorer, but Eclipse provides a fast method for opening the project's folders. As Figure 52 demonstrates, right-click the folder in the Package Explorer, and select **Show In** > **System Explorer**.

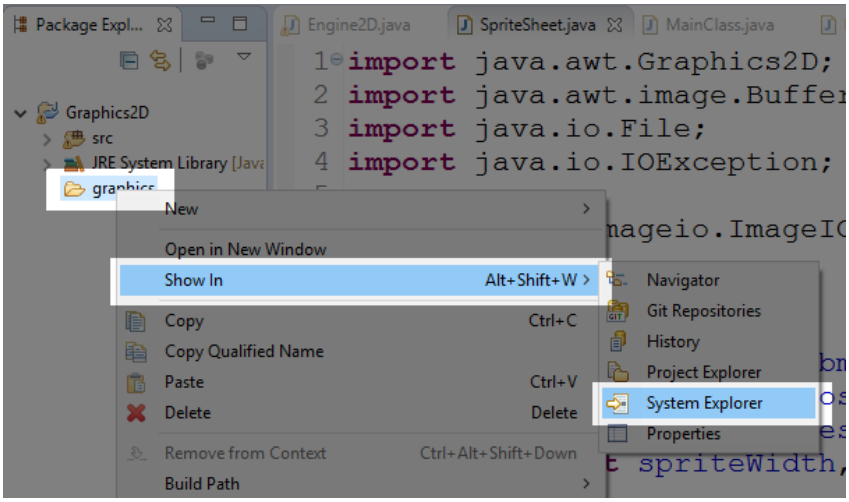


Figure 52: Show in System Explorer

Find the **`spaceracer.png`** file on your desktop (or wherever you exported it after adding transparency). Copy and paste this file into the **`graphics`** folder of the project in the System Explorer, as per Figure 53. Close the window in the Windows System Explorer and return to Eclipse.

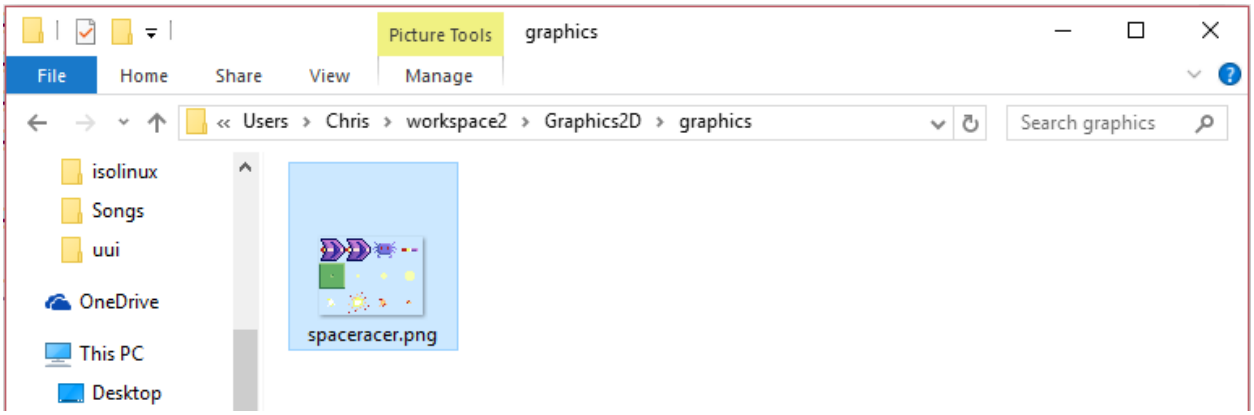


Figure 53: Graphics Folder in System Explorer

Our PNG image is now included in the **`graphics`** folder, but at present Eclipse is not aware of it. Right-click the **`graphics`** folder in the Package Explorer and select **Refresh**, as per Figure 54. This will cause Eclipse to include any files it finds in the folder in our application. Every time you update your sprites or add files to the folders in your application, you should refresh the folders in Eclipse.

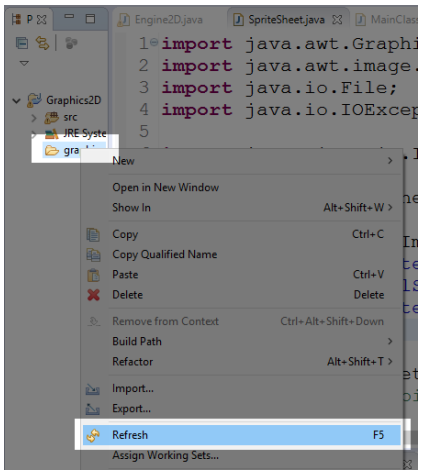


Figure 54: Refreshing the Graphics Folder

Loading and rendering sprites

Code Listing 8.3 shows the code that loads our sprites as a **`SpriteSheet`** instance in the constructor of our **`Engine2D`**. Be careful if you are copying and pasting this code—I have only included the important lines of code of **`Engine2D`**, so this is not the entire **`Engine2D`** class.

Code Listing 8.3: Loading spacerager.png

```
    // Width and height of the window.
    private int width, height;
```

```
	private SpriteSheet sprites;

	// Constructor
	public Engine2D(int windowWidth, int windowHeight, int fps) {
	width = windowWidth;
	height = windowHeight;

	// Load the sprites:
	sprites = new SpriteSheet("graphics/spaceracer.png", 16, 16);

	// Start the render/update loop.
	Timer timer = new Timer(1000/fps, this);
	timer.start();
	}
```

Notice that the parameters for the sprite width and height are 16, which occurs because the spaceracer.png image was drawn to have each separate frame fit inside a 16x16 pixel box. When we render our sprite sheet, we have the option of rendering only a portion of it (as we will see in a moment). If you have sprites of differing sizes on your sprite sheet, you need to know where each sprite begins and ends (in terms of the x and y coordinates) in order to correctly render the portions of the sprite sheet. Because all the sprites are the same size, we can perform a simple calculation to correctly render the desired portion of the image. This is a very fast way of including sprite sheets when all the sprites are made to be exactly the same size and when they are properly spaced on a grid in the PNG file. When we have loaded our **SpriteSheet** object, we can render a test sprite in our **Engine2D** render method in order to make sure everything is running smoothly, as we see Code Listing 8.4.

Code Listing 8.4: Rendering a Test Sprite

```
	// Temporary test, delete this line after making
	// sure the program animates:
	static int x = 0;

	private void render(Graphics2D g) {
		// Clear the screen to blue
		g.setBackground(Color.DARK_GRAY);
		g.clearRect(0, 0, width, height);

		// Temporary test:
		sprites.render(g, x % 2, x, 0);
		x++;
	}
```

In Code Listing 8.4, I have included an **x** member variable and incremented it each frame in the update method. In the render method, I have rendered sprite number "**x%2**" at position **x**. This will cause the first two frames of our sprite sheet to be drawn so that they slowly move across the screen from left to right, as per Figure 55.

Figure 55: Small Spaceship (Cropped)

Scaling sprites

Our spaceship is very small. As a stylistic choice, we might want our sprites to appear larger and pixelated, similar to games from the 1990s. The sprites will be easy for the player to see, and our game will have a retro aesthetic. We can scale our sprites by multiplying the coordinates of the destination in the **`SpriteSheet.render`** method. When you run the test application after implementing the changes in Code Listing 8.5, you should see a much larger spaceship.

Code Listing 8.5: Scaling the Sprite

```
public void render(Graphics2D g, int spriteIndex, int x, int y) {
      // Make sure the sprite is actually on our spriteSheet.
      if(spriteIndex >= totalSprites) return;
      // Compute the source x and y.
      int srcX = (spriteIndex % spritesAcross) * spriteWidth;
      int srcY = (spriteIndex / spritesAcross) * spriteHeight;
      // Draw the image
      g.drawImage(bmp,
      x*2, // Destination x1
      y*2, // Destination y1
      (x + spriteWidth)*2, // Destination x2
      (y + spriteHeight)*2, // Destination y2
      srcX, // Source x1
      srcY, // Source y1
      srcX + spriteWidth, // Source x2
      srcY + spriteHeight,// Source y2
      null); // Observer
      }
```

Timing and frame skipping

Many computing devices could potentially run our games and applications. Each device consists of different hardware, and each hardware has a specific performance—some devices are faster than others. In order to make our games run at a smooth, consistent rate, we need to move our sprites so that they appear to move at the same speed regardless of the hardware. We can do this by employing a technique called frame skipping.

A fast computer might be capable of rendering four frames in a short amount of time, and a slower computer might render only two frames in the same amount of time. However, our objects must move the same distance despite the number of frames. The faster computer might render frames more smoothly, but the game play must appear to run at the same speed. One way to achieve this effect is to scale the movement of our objects by the amount of time that has elapsed since the last call to update. We will include a new class called **HPTimer** (short for High-Precision Timer) that accurately records the amount of time passing so that we can use it in our call to the update method. Create the **HPTimer** class and add the code in Code Listing 8.6.

Code Listing 8.6: HPTimer Class

```
public class HPTimer {
	// Member variables
	long startTime, lastTime, currentTime;

	// Set the start, last and current times to now:
	public void reset() {
		startTime = System.currentTimeMillis();

		// You can also use nano time:
		//startTime = System.nanoTime();

		lastTime = startTime;
		currentTime = startTime;
	}

	// Reset the timer.
	public void start() {
		reset();
	}

	// Record the current time.
	public void update() {
		lastTime = currentTime;
		currentTime = System.currentTimeMillis();

		// If using nano time:
		//currentTime = System.nanoTime();
	}

	// Return the time since the last call to update.
```

```
	public double timeDelta() {
		double d = (double) currentTime - (double) lastTime;
		d /= 1000.0;

		// If using nano time:
		// d /= 1000000000.0;
		return d;
	}

	// Return the time since the last call to reset.
	public double timeTotal() {
		double d = (double) currentTime - (double) startTime;
		d /= 1000.0;

		// If using nano time:
		//d /= 1000000000.0;
		return d;
	}
}
```

Code Listing 8.6 shows the code for our timer class. As our application develops, if you find that the timer is not accurate enough, you might want to try uncommenting the “nano time” lines in order to use **`System.nanoTime`** instead of reading milliseconds. The class does nothing more than read the time in milliseconds each time the update method is called, and it offers **`timeTotal`** and **`timeDelta`** methods that return the total amount of time that has elapsed since the start of the timer, along with the elapsed time since the last call to update. Add an **`HPTimer`** instance to the **`Engine2D`** class. In Code Listing 8.7, I have called my instance **`hpTimer`**.

Code Listing 8.7: Create an HPTimer

```
	private SpriteSheet sprites;

	private HPTimer hpTimer;

	// Constructor
	public Engine2D(int windowWidth, int windowHeight, int fps) {
		width = windowWidth;
		height = windowHeight;
		// Load the sprites.
		sprites = new SpriteSheet("graphics/spaceracer.png", 16, 16);

		// Start the HPTimer.
		hpTimer = new HPTimer();
		hpTimer.start();

		// Start the render/update loop.
		Timer timer = new Timer(1000/fps, this);
		timer.start();
```

```
    }
```

In the **actionPerformed** method, we want to pass the **timeTotal** and **timeDelta** parameters to our **update** function by calling **hpTimer.update()** to ensure the **hpTimer** has read the most up-to-date time, then we pass the total and delta times to the **Engine2D.update** method function, as in Code Listing 8.8.

Code Listing 8.8: Updating the Timer and Passing the Times

```
    // This event is called when the timer fires at the specified fps.
    public void actionPerformed(ActionEvent e) {
        // Read the most up-to-date time:
        hpTimer.update();

        // Pass HPTimer's times to our update method:
        update(hpTimer.timeTotal(), hpTimer.timeDelta());

        repaint();
    }
```

We can now render our spaceship again, but this time we will scale the ship's movement by the **timeDelta**. Code Listing 8.9 shows the altered test code. This time when we run the application, the ship will move at the rate of one pixel every second. We should note our hardware's power doesn't matter here. If a very slow computer runs this application, it will take the ship exactly the same amount of time to reach the right edge of the screen as it would with a very powerful computer—even if the slow computer is only able to render three frames, the ship's speed will be exactly the same. In general, when creating animations, we always want to scale by **timeDelta** (which is the elapsed time since the last call to update).

Code Listing 8.9: Very Slow Spaceship

```
    // Temporary test, delete this line after making
    // sure the program animates:
    static double x = 0;

    private void render(Graphics2D g) {
        // Clear the screen to blue.
        g.setBackground(Color.DARK_GRAY);
        g.clearRect(0, 0, width, height);

        // Temporary test:
        sprites.render(g, (int)x % 2, (int)x, 0);
        x+=hpTimer.timeDelta();
    }
```

Animation class

Many of the objects in our game will be rendered with repeating animations. These animations will consist of consecutive frames from our sprite sheet, such as the first two frames, which represent the ship. The animations have a specific time for the frames and a start time. Code Listing 68 shows the code for the **`Animation`** class.

Code Listing 8.10: Animation Class

```
public class Animation {
	private double speed, startTime;
	private int firstFrame, frameCount;
	private int currentFrame;
	private boolean isComplete = false;
	private boolean looping;

	// Constructor for looping/multiframe animation.
	public Animation(double speed, double startTime, int firstFrame,
			int frameCount, boolean looping) {
		this.speed = speed;
		this.startTime = startTime;
		this.firstFrame = firstFrame;
		this.frameCount = frameCount;

		// Reset
		currentFrame = firstFrame;
		isComplete = false;
		this.looping = looping;
	}

	// Constructor for single-frame animation.
	public Animation(int frame) {
		speed = 1.0;
		startTime = 0.0;
		firstFrame = frame;
		frameCount = 1;
		// Reset
		currentFrame = firstFrame;
		isComplete = false;
		this.looping = true;
	}

	// Compute the current frame and the
	// isComplete boolean.
	public void update(double timeTotal) {
		double elapsedTime = timeTotal - startTime;
		currentFrame = (int)(elapsedTime / speed);

		if(currentFrame < 0) currentFrame = 0;
```

```
            // If the frame is past the end of the animation,
            // set it to the last frame.
            if(currentFrame >= frameCount) {
                  // If the animation does not loop, set it to the final
                  // frame indefinitely.
                  if(!looping)
                        currentFrame = firstFrame + frameCount - 1;
                  // If the animation is looping,
                  // set it back to the first frame.
                  else {
                        currentFrame = firstFrame;
                        startTime = timeTotal;
                  }

                  isComplete = true;
            }

            // Otherwise, the current frame is the first frame +
            // however many frames we've played so far:
            else
                  currentFrame += firstFrame;
      }

      // Returns the current frame.
      public int getCurrentFrame() {
            return currentFrame;
      }

      // Determines if the animation has played all frames.
      public boolean getIsComplete() {
            return isComplete;
      }
}
```

Game objects

Our game will consist of many objects—the player, scrolling stars, and alien baddies. These objects have common features, such as an x and y position, the ability to update/render, and an animation. We will create a parent class called **`GameObject`** from which we will inherit to create the specific object types in our game. Code Listing 8.11 shows the **`GameObject`** class.

Code Listing 8.11: GameObject Class

```
import java.awt.Graphics2D;
public abstract class GameObject {
```

```
	// Position
	public double x, y;

	// Is the object visible?
	private boolean visible = true;

	// The object's animation
	private Animation animation = null;

	// Update and Render
	public void update(double timeTotal, double timeDelta) {
		if(animation != null)
			animation.update(timeTotal);
	}

	// Render the animation with the current frame if it exists and
	// is visible.
	public void render(Graphics2D graphics, SpriteSheet sprites) {
		if(visible && animation != null)
		sprites.render(graphics, animation.getCurrentFrame(),
				(int)x, (int)y);
	}

	// Getters and setters
	public double getX() {
	return x;
	}

	public double getY() {
	return y;
	}

	public boolean getVisible() {
		return visible;
	}

	public void setVisible(boolean visible) {
		this.visible = visible;
	}

	public Animation getAnimation() {
		return animation;
	}

	public void setAnimation(Animation animation) {
		this.animation = animation;
	}

	// Location tests:
```

```
	// Test if the object is outside the screen to the left.
	public boolean isOffScreenLeft() {
	return x < -16;
	}
	// Test if the object is outside the screen to the right.
	public boolean isOffScreenRight() {
	return x >= 320;
	}
	// Test if the object is outside the screen at the top.
	public boolean isOffScreenTop() {
	return y < -16;
	}
	// Test is the object is outside the screen at the bottom.
	public boolean isOffScreenBottom() {
	return y >= 240;
	}

	// Compute the distance between the objects.
	public double getDistance(GameObject o) {
		// Faster, but less accurate detection:
		// return Math.abs(o.x - x) + Math.abs(o.y - y);

		// More accurate, but slow version:
		return Math.sqrt((o.x - x) * (o.x - x) +
			(o.y - y) * (o.y - y));
	}

}
```

Notice that the **GameObject** class is abstract. We will not create instances of **GameObject** directly, but we want to encapsulate all the elements that are the same for each object in the game so that we do not need to reprogram them for each object type. In addition to reducing our coding for each of the child classes, we will be able to store all of our game's objects in a single **ArrayList** and call all of the object's **update**/**render** methods very simply. This is an example of polymorphism in action.

Stars

In this section, we will create a scrolling background of stars. The stars will inherit from the **GameObject** class and call **super.update** in their update method. Add a new class to your application called **Star**. The code for this class is listed in Code Listing 8.12. One of the most important things to remember is that our **GameObject** parent class updates the current animation of the object, which means we should be sure to call **super.update** in the **update** method of all the child classes or else handle the animation updating in the child classes.

Code Listing 8.12: Scrolling Star Class

```
import java.awt.Graphics2D;

public class Star extends GameObject {
	double speed;

	public Star() {
		// Begin the stars in a random location:
		x = Math.random() * 320.0;
		y = Math.random() * 240.0;

		// Set the stars to a random speed:
		speed = Math.random() * 30.0 + 30;
	}

	public void update(double timeTotal, double timeDelta) {
		// Call the parent update.
		super.update(timeTotal, timeDelta);

		// Move the star left.
		x -= speed * timeDelta;

		// Reset the star on the right when it goes off screen.
		if(isOffScreenLeft()) {
			x = 320.0; // Just outside the right-hand edge
			y = Math.random() * 240.0 - 16; // Random Y location
		}
	}
}
```

Next, add an **`ArrayList`** for holding our **`GameObject`** objects to the **`Engine2D`** class and create 100 stars in the constructor (see Code Listing 3.13). You will also need to add an import for the **`ArrayList`** collection—"**`import java.util.ArrayList;`**"—at the start of the **`Engine2D`** class.

Code Listing 8.13: Creating gameObjects

```
	// The GameObjects array list:
	ArrayList<GameObject> gameObjects = new ArrayList<GameObject>();

	// Constructor
	public Engine2D(int windowWidth, int windowHeight, int fps) {
		width = windowWidth;
		height = windowHeight;

		// Load the sprites.
		sprites = new SpriteSheet("graphics/spaceracer.png", 32, 32);
```

```
		// Create 100 stars.
		for(int i = 0; i < 100; i++) {
			Star s = new Star();
			s.setAnimation(new Animation(Math.random() * 2 + 0.2,
Math.random(), 5, 3, true));
			gameObjects.add(s);
			}
```

Now that we have an array list for all the objects in our game, we want to call update and render for the elements of the array list in the **update** and **render** methods of the **Engine2D** class. Notice also that I have removed the test code from when we rendered our test spaceship (we will add the spaceship again in a moment, but it will be controlled by the keyboard).

Code Listing 8.14: Updating and Rendering GameObjects

```
	private void update(double timeTotal, double timeDelta) {
		// Update the game objects:
		for(GameObject o: gameObjects)
			o.update(timeTotal, timeDelta);
	}

	private void render(Graphics2D g) {
		// Clear the screen to blue.
		g.setBackground(Color.DARK_GRAY);
		g.clearRect(0, 0, width, height);

		// Render the game objects:
		for(GameObject o: gameObjects)
			o.render(g, sprites);
	}
```

You should be able to run the application and see a scrolling background of animated stars. The technique used here is a simple version of a technique called parallax scrolling. We create a series of background images (stars), rendering them on top of each other, and scroll those that are nearer to the camera faster than those that are farther away. Figure 56 shows a screenshot of our game so far.

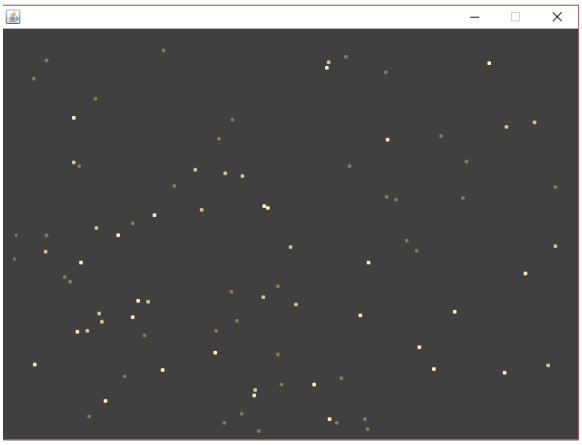


Figure 56: Stars

Walls

The stars are meant to be in the background. Our game will consist of a series of scrolling walls that the player must avoid and baddies that the player must either avoid or shoot. The walls are similar to the stars, except that we will generate them on the fly along the right edge of the screen and delete from the **`gameObjects`** array list as they reach the left side of the screen. Code Listing 8.15 shows the new **`Wall`** class. Note that this class also includes a method called **`collisionWithShip`** that we will use later to determine if the ship has collided with a wall.

Code Listing 8.15: Wall Class

```
import java.awt.Graphics2D;

public class Wall extends GameObject {
	public Wall(double x, double y) {
		this.x = x;
		this.y = y;

		this.setAnimation(new Animation(4));
	}

	// Move the wall to the left.
	public void update(double timeTotal, double timeDelta) {
		super.update(timeTotal, timeDelta);

		x -= 80 * timeDelta;
	}
}
```

In order to use this class, we will implement several new variables in our **`Engine2D`** class (see Code Listing 8.16). One interesting note—if you pass an argument to the **`Random()`** constructor

(i.e. **Random(1238)**), the walls will be generated in exactly the same pattern each time. You will also have to import "**java.util.Random**" at the top of the **Engine2D** class.

Code Listing 8.16: Wall Variables in Engine2D

```
	// The GameObjects array list:
	ArrayList<GameObject> gameObjects = new ArrayList<GameObject>();

	// Wall variables
	double nextWallGenerationTime = 1.0;
	Random wallRNG = new Random();		// Any argument will
	// cause walls to be generated with the same pattern
	// every time!

	// Constructor
	public Engine2D(int windowWidth, int windowHeight, int fps) {
```

Next, we generate the walls and remove them as they leave the left edge of the screen in the **Engine2D**'s update method (see Code Listing 8.17). Note that in order to delete walls from the **ArrayList**, it is no longer safe to employ the for each loop (we should never modify a collection by adding or removing items while iterating through it using a for each loop), and I have rewritten the **Engine2D** update with a **for** loop. Figure 57 shows a screenshot of the game with stars and walls.

Code Listing 8.17: Generating and Deleting Walls

```
	private void update(double timeTotal, double timeDelta) {
		// Generate new walls:
		if(timeTotal >= nextWallGenerationTime) {
// Add 0.5 seconds to the wall generation time.
			nextWallGenerationTime += 0.5;
			for(int i = 0; i < 14; i++) {
				if(wallRNG.nextInt(3) == 0) {
					gameObjects.add(new Wall(640, i * 32));
				}
			}
		}

		for(int i = 0; i < gameObjects.size(); i++) {
			GameObject o = gameObjects.get(i);
			o.update(timeTotal, timeDelta);

			// If the object is a wall:
			if(o instanceof Wall) {
				if(o.isOffScreenLeft()) {
				// Delete as they go off the screen to the left.
					gameObjects.remove(i);
					i--;
```

```
                        continue;
                    }
                }
            }
        }
```

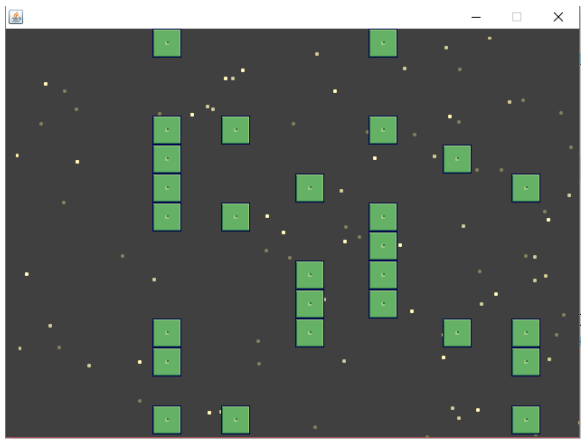


Figure 57: Stars and Walls

Baddies

Next, we will add some baddies. The baddies are essentially the same as the walls, except that we will enable to the player to shoot them and they will move with a clever sine-wave pattern, which will make them harder to avoid and shoot. The code for the new Baddie class is presented in Code Listing 8.18.

Code Listing 8.18: Baddie Class

```
import java.awt.Graphics2D;
import java.util.Random;

public class Baddie extends GameObject {
	private double startY;

	private double frequency;
	private double amplitude;

	// Constructor
	public Baddie(double x, double y) {
		this.x = x;
		this.y = y;

		startY = y;
```

```
            // Create random frequency and amplitude.
            Random r = new Random();
            frequency = r.nextDouble() * 2.0 + 2.0;
            amplitude = r.nextDouble() * 45 + 45;

            // Set the animation:
            this.setAnimation(new Animation(2));
      }

      // Move the baddie to the left.
      public void update(double timeTotal, double timeDelta) {
            super.update(timeTotal, timeDelta);

            x -= 60 * timeDelta;

            y = startY + (Math.sin(timeTotal * frequency) * amplitude);
      }
}
```

Adding the baddies to our game is similar to adding the walls. In Code Listing 8.19, I have added several variables to the **Engine2D** class below the wall variables that will be used to generate baddies. Once again, if you would like to generate baddies in exactly the same pattern every time, you can pass an argument to the Random constructor (e.g., **Random(678763)**). Code Listing 8.20 shows the new **Engine2D update** method for generating and updating the baddies.

Code Listing 8.19: Engine2D Baddie Variables

```
      // Wall variables
      double nextWallGenerationTime = 1.0;
      Random wallRNG = new Random();      // Any argument will
      // cause walls to be generated with the same pattern
      // every time!

      // Baddie variables
      double nextBaddieGenerationTime = 2.0;
      Random baddieRNG = new Random();

      // Constructor
      public Engine2D(int windowWidth, int windowHeight, int fps) {
```

Code Listing 8.20: Generating and Updating Baddies

```
      private void update(double timeTotal, double timeDelta) {
            // Generate new walls:
            if(timeTotal >= nextWallGenerationTime) {
```

```
// Add 0.5 seconds to the wall generation time.
      nextWallGenerationTime += 0.5;

                  for(int i = 0; i < 14; i++) {
                        if(wallRNG.nextInt(3) == 0) {
                              gameObjects.add(new Wall(320, i * 32));
                        }
                  }
            }

            // Generate new Baddies.
            if(timeTotal >= nextBaddieGenerationTime) {
      // Death wave:
      //nextBaddieGenerationTime += baddieRNG.nextDouble() * 0.2 + 0.1;
      // Normal wave:
      nextBaddieGenerationTime += baddieRNG.nextDouble() * 4.0 + 0.5;

      gameObjects.add(new Baddie(320,baddieRNG.nextInt(280)-40));
            }

            for(int i = 0; i < gameObjects.size(); i++) {
                  GameObject o = gameObjects.get(i);
                  o.update(timeTotal, timeDelta);

                  // If the object is a wall, or a baddie:
                  if(o instanceof Wall || o instanceof Baddie) {
                        if(o.isOffScreenLeft()) {
                        // Delete if they go off the screen to the left.
                              gameObjects.remove(i);
                              i--;
                              continue;
                        }
                  }
            }
      }
```

For a little fun, I have included a Death Wave mode that we can switch to for five seconds at a rate of once every 30 seconds or so. This will greatly increase the challenge of our game, and it will add a degree of progress while the player plays. The mode is commented out in Code Listing 8.20, but it consists of very fast generation of baddies. I will leave the in-game switching of this mode as an exercise for you to implement.

When you start the game, after a moment you should see baddies being generated on the right side of the screen—they bob up and down using a sine-wave pattern, and they exit on the left. Play around with the **`frequency`** and **`amplitude`** values in the **`Baddie`** constructor in order to explore the attributes of the sine wave. Setting the **`frequency`** to values higher than 2.0 will cause the baddies to bob up and down very rapidly, and setting the **`amplitude`** value to higher values will increase the vertical range of their pattern. Be careful not to set these values too

high—our collision detection will be most accurate if objects are not allowed to move more than 16 pixels per frame.

Figure 8.17 shows a screenshot of our game with stars, walls, and baddies. This is actually a screenshot of the Death Wave. Without the Death Wave option, the number of baddies generate at a speed that means there will be only one or two on screen at once.

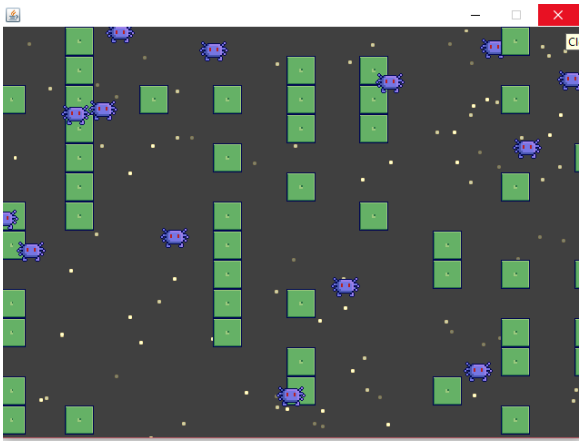


Figure 58: Stars, Walls, and Baddies

Reading the keyboard

Before we can add our hero's spaceship, we need to implement a method for controlling it. Let's now look at how to respond to events caused by the user pressing and holding keys on the keyboard. In order to allow our application to respond to input from the keyboard, add a new class called **`Keyboard`** that implements the **`KeyListener`** interface.

The **`Keyboard`** class in Code Listing 8.21 is an example of a singleton. This means we will design a single keyboard and never create multiple instances from the class. A singleton class can be designed in Java in many ways. I have marked the constructor as **`private`**, which prevents instances of the class from being created (instead, we will call **`Keyboard.Init`** to initialize the singleton—using this approach is sometimes called a Factory Pattern). Note also—the methods of this class and the **`keyStates`** array are all static. They belong to the class rather than an instance of it. This effectively means that the **`Keyboard`** class exists as a single, static object. We cannot create nor interact with instances, instead we interact with the class itself.

Code Listing 8.21: Keyboard Class

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class Keyboard implements KeyListener {
      // 256 key states: true means the key is down
      //                 false means the key is up.
      private boolean[] keyStates = null;
```

```
	// The only instance of this class is the following
	// private, static instance:
	private static Keyboard staticInstance = null;

	// Private Constructor:
	private Keyboard() {
		keyStates = new boolean[256];
	}

	// Public init method that creates the
	// static key states if they do not exist.
	public static void init() {
		staticInstance = new Keyboard();

		reset();
	}

	public static Keyboard getInstance() {
		return staticInstance;
	}

	// Set all key states to false.
	public static void reset() {
		for(int i = 0; i < 256; i++)
			staticInstance.keyStates[i] = false;
	}

	// Test if a key is down.
	public static boolean isKeyDown(int keyCode) {
		return staticInstance.keyStates[keyCode & 255];
	}

	// Set a key to down; true.
	public void keyPressed(KeyEvent e) {
		staticInstance.keyStates[e.getKeyCode() & 255] = true;
	}

	// Set a key to up; false.
	public void keyReleased(KeyEvent e) {
		staticInstance.keyStates[e.getKeyCode() & 255] = false;
	}

	// Extra, unused method from KeyListener interface.
	public void keyTyped(KeyEvent e) { }
}
```

The **Keyboard** class consists of a **static** array of **boolean** variables. We will use one element of this array for each of 256 possible keys. A value of **true** will mean that a particular key is held down, and a value of **false** will mean it is not. In reality, there are more than 256 possible

keys that could be down on a modern keyboard (taking into consideration languages other than English). In game programming, we are typically interested only in tracking keys such as the letters A to Z, digits 0 to 9, the arrow keys, space bar, etc. So, we will only store an array of 256 different keys at most, and we will read only the lowest byte of any keys that the user hits (rather than reading the entire Unicode short **int**—which would require an array of 2^16 different key states). After the array is created in the **init** method, we call **reset** to clear all key states to **false** and ensure that the initial state of the keyboard has no keys held down.

The **isKeyDown** method returns the current state of specified **keyCode**. So, if the key is down, this method returns **true**, and if it is up, the method returns **false**. In the **keyPressed** event, we read the **keyCode** of the key the user has just pressed, we limit the range of the code from 0 to 255 with a bitwise &, and we set the corresponding **keyState** to true, which means the key is now being held down. **KeyReleased** is similar to **keyPressed**, except that we clear the key's state to **false**, which means the key is no longer being held down**.**

When users press a key, the **keyPressed** event will occur. When users release a key, the **keyReleased** event will occur. And, when users type a key, the **keyTyped** event will occur. Note that the **keyTyped** responds repeatedly if the user holds down the key, but we do not need this event in our game. However, we must provide it because it is required by the **KeyListener** interface.

Next, we need to add a **keyListener** to our **MainClass**. The updated constructor for the **MainClass** is listed in Code Listing 8.22. A **keylistener** is any class that implements the **KeyListener** interface.

Code Listing 8.22: Initializing the Keyboard in the MainClass

```
      private MainClass(int windowWidth, int windowHeight) {
            setSize(windowWidth, windowHeight); // Set window size.
            setLocationRelativeTo(null); // Default location.
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Exit on
close.
            setVisible(true);

            // Init the static singleton keyboard:
            Keyboard.init();

            // Add a keylistener.
            addKeyListener(Keyboard.getInstance());


            // Create and add the engine JPanel:
            final Engine2D engine = new Engine2D(windowWidth, windowHeight,
30);
            add(engine);
      }
```

Keyboard controlled player

Now that we have a **Keyboard** class, we can add the player **GameObject**, which will be a player that can use the keyboard. Code Listing 8.23 shows the basic **Player** class.

Code Listing 8.23: Player Class

```
public class Player extends GameObject {
	double shipSpeed = 320.0;

	public Player(double x, double y) {
		// Set the x/y.
		this.x = x; this.y = y;

		// Set the animation.
		this.setAnimation(new Animation(0.1, 0.0, 0, 2, true));
	}

	public void update(double timeTotal, double timeDelta) {
		// Call parent's update:
		super.update(timeTotal, timeDelta);

		//Up/down
		if(Keyboard.isKeyDown(40)) y += shipSpeed * timeDelta;
		if(Keyboard.isKeyDown(38)) y -= shipSpeed * timeDelta;

		// Left/right
		if(Keyboard.isKeyDown(37)) x -= shipSpeed * timeDelta;
		if(Keyboard.isKeyDown(39)) x += shipSpeed * timeDelta;

		// Make sure the player is on the screen.
		if(x < 0) x = 0;
		if(y < 0) y = 0;
		if(x > 320 - 32.0) x = 320 - 32.0;
		if(y > 240 - 32.0) y = 240 - 32.0;
	}
}
```

Notice that in Code Listing 8.23, when we read the keys, we use numbers, such as 40, 38, etc. These are virtual keycodes, and there is a different code for each key on the keyboard. For a complete list of the codes for every key, see the following: http://docs.oracle.com/javase/6/docs/api/constant-values.html#java.awt.event.KeyEvent.VK_0. We are reading only the keydown and keyup events, so the user will be able to hold down two keys at once and the ship will move diagonally.

The ship can be added to the **gameObjects** array after we add the stars in the **Engine2D** constructor. I have added a separate copy of the player to the **Engine2D** class as a member variable, too. We do this because we need to test collisions between the **player** and the **walls**/**baddies**, so we need to know which object is in the array. As it happens, the **player** will

always be object number 100 in the array, so we can simply use item number 100 in the **gameObjects** list, too. Also note that this is not a different instance of the **Player** class but rather a different reference to the same instance. Code Listing 8.24 shows the new player variables that will be added to the **Engine2D** class and the changes for adding the new player object to the **gameObjects** array.

Code Listing 8.24: Adding the Layer to Engine2D

```
      Random baddieRNG = new Random();

      // Player variables:
      Player player;
      boolean playerExploded = false;

      // Constructor
      public Engine2D(int windowWidth, int windowHeight, int fps) {
            width = windowWidth;
            height = windowHeight;

            // Load the sprites.
            sprites = new SpriteSheet("graphics/spaceracer.png", 16, 16);

            // Create 100 stars.
            for(int i = 0; i < 100; i++) {
                  Star s = new Star();
                  s.setAnimation(new Animation(Math.random() * 2 + 0.2,
Math.random(), 5, 3, true));
                  gameObjects.add(s);
                  }

            // Create the player:
            player = new Player(16.0, 100.0);
            gameObjects.add(player);

            // Start the HPTimer.
```

At this point, you should be able to run the application and control the ship. You will be able to fly through walls and baddies because we have not yet implemented collision detection, but your ship should stop at the edges of the screen. I have also declared a **boolean** called **playerExploded**, which we will use in a moment.

Collision detection

Our walls do not do anything at the moment. We want to explode the spaceship when it hits a wall or a baddie, and we can do this by testing the distance between the **player** object and the center of the **walls**/**baddies**. If the player object is closer than, say, 14 pixels to the center of an obstacle, we will deem this too close for comfort and we will explode the ship, then reset the player back to the start by destroying all **walls** and **baddies**. These changes are all in the

Engine2D update method and highlighted in Code Listing 8.25. I have included the entire **Engine2D update** method.

Code Listing 8.25: Update Method with Exploding Ship

```
	private void update(double timeTotal, double timeDelta) {
		// Generate new walls.
		if(timeTotal >= nextWallGenerationTime) {
			nextWallGenerationTime += 0.5;// Add 0.5 second to the
wall generation time.

			for(int i = 0; i < 14; i++) {
				if(wallRNG.nextInt(3) == 0) {
					gameObjects.add(new Wall(320, i * 32));
				}
			}
		}

		// Generate new Baddies.
		if(timeTotal >= nextBaddieGenerationTime) {
	// Death wave:
	//nextBaddieGenerationTime += baddieRNG.nextDouble() * 0.2 + 0.1;
			// Normal wave:
			nextBaddieGenerationTime += baddieRNG.nextDouble() * 4.0
+ 0.5;

			gameObjects.add(new Baddie(320, baddieRNG.nextInt(280)-
40));
		}

		for(int i = 0; i < gameObjects.size(); i++) {
			GameObject o = gameObjects.get(i);
			o.update(timeTotal, timeDelta);

			// If the object is a wall or a baddie:
			if(o instanceof Wall || o instanceof Baddie) {

				// Test if the wall/baddie has hit the player.
				if(o.getDistance(player) < 14 && !playerExploded) {
					player.setAnimation(new Animation(0.5,
hpTimer.timeTotal(), 8, 4, false));
					playerExploded = true;
				}

				if(o.isOffScreenLeft()) {
					// Delete if they go off the screen to the
left.
					gameObjects.remove(i);
					i--;
```

```
                        continue;
                    }
                }
            }


            // When the explosion animation for the payer is finished,
destroy all walls and baddies
            // and reset the player.
            if(playerExploded && player.getAnimation().getIsComplete()) {
                player.x = 16;
                player.y = 100;
                playerExploded = false;
                player.setAnimation(new Animation(0.1, 0.0, 0, 2, true));
                for(int i = 0; i < gameObjects.size(); i++) {
                    if(gameObjects.get(i) instanceof Wall
||gameObjects.get(i) instanceof Baddie) {
                        gameObjects.remove(i);
                        i--;
                    }
                }
            }
        }
```

Player bullets

At the moment, our game does not seem particularly fair (or fun), so we will allow the player to shoot bullets. These will destroy the baddies but not the walls. Code Listing 8.26 shows the new **`Bullet`** class.

Code Listing 8.26: Bullet Class

```
import java.awt.Graphics2D;

public class Bullet extends GameObject {

    // Constructor
    public Bullet(double x, double y) {
        this.x = x;
        this.y = y;

        // Set the animation.
        this.setAnimation(new Animation(3));
    }

    // Move the bullet to the right.
    public void update(double timeTotal, double timeDelta) {
```

```
            super.update(timeTotal, timeDelta);

            x += 800 * timeDelta;
        }
}
```

When the player holds down the space bar, we want to create bullets and fire them to the right. We do not want the player to have too much firepower, so we will limit the speed that bullets are created by adding several variables to the **Engine2D** class. Code Listing 8.27 shows the new variables that will be added to the class.

Code Listing 8.27: Bullet Variables

```
        boolean playerExploded = false;

        // Bullet variables
        double lastBulletTime = 0.0;
        double bulletCreationSpeed = 0.25; // 4 bullets per second

        // Constructor
        public Engine2D(int windowWidth, int windowHeight, int fps) {
```

Code Listing 8.28 shows the code used to create the bullet once every 0.25 seconds that the space bar is held down. This code should be placed in the **Engine2D** update.

Code Listing 8.28: Creating Bullets

```
            // Create bullets
            if(Keyboard.isKeyDown(32) &&
                        hpTimer.timeTotal() - lastBulletTime >
                        bulletCreationSpeed && !playerExploded) {
                    gameObjects.add(new Bullet(player.x, player.y));
                    lastBulletTime = hpTimer.timeTotal();
            }

// When the explosion animation for the player is finished, destroy all
walls and baddies.
```

Finally, we should test collisions between all the objects that are instances of **bullet** and all objects that are instances **baddie**. If any **bullets** collide with any **baddies**, we will simply remove the **baddie** from the **gameObjects** list. It would make more sense to explode the **baddies**, but I will leave the implementation of such explosions to you. The routine in Code Listing 8.29 is very slow, and it highlights the difficulty in collision detection—if there are 1000 objects, there are a lot of possible collisions. For this reason, I have included a quick and dirty collision detection routine in the **GameObject** class that you might consider if your game begins to lag when there are too many collisions to detect. Alternately, we could organize our objects into search trees and greatly reduce the number of collisions we need to check.

Code Listing 8.29: Checking Baddie Collisions

```
            for(int i = 0; i < gameObjects.size(); i++) {
                  GameObject o = gameObjects.get(i);
                  o.update(timeTotal, timeDelta);

                  // If this object is a bullet:
                  if(o instanceof Bullet) {
                        // Delete the bullet if it goes off the screen
                        // to the right.
                        if(o.isOffScreenRight()) {
                              gameObjects.remove(i);
                              i--;
                              continue;
                        }

                        // Check all baddies for collisions.
                        for(int j = 100; j < gameObjects.size(); j++) {
                              GameObject g = gameObjects.get(j);
                              // If this is a baddie:
                              if(g instanceof Baddie) {
                                    // If the baddie has hit the bullet:
                                    if(o.getDistance(g) < 14) {
                                          // Remove the baddie.
                                          gameObjects.remove(j);
                                          j--;
                                          i--;
                                    }
                              }
                        }
                  }

                  // If the object is a wall, or a baddie:
```

You should be able to run the game and shoot baddies. At this point, a score system might be added to give the player a feeling of progress. The scoring should use an increasing difficulty curve and deadly waves of many baddies. All of these things can be implemented easily and quickly, and I will again leave those adventures up to you to implement.

Our game seems pretty crummy (I wouldn't play it for more than a few minutes before becoming bored and getting back to programming). However, the techniques in this chapter are virtually identical to those we might use to easily create Android games (the Android platform runs Java programs almost exclusively). If you are interested in developing Android applications, download and explore the Android Studio—you will find that this IDE is very similar in many ways to Eclipse (in fact, for a long time, Eclipse was the IDE of choice for Android developers). You will also find the GUI tools in Android Studio very similar to those offered by Eclipse.

Conclusion and Thank You

Java is a marvelous computer programming language full of rich, descriptive, and powerful mechanisms. It is currently the most-programmed computer language in the world (http://pypl.github.io/PYPL.html), and millions of Java applications are running at every moment all over the globe. I hope you have enjoyed reading the *Java Succinctly* e-books as much as I have enjoyed writing them. And I hope you learned something from the techniques presented. We have examined a multitude of syntaxes and techniques involved with Java programming, and yet we have still only scratched the surface. Java is a vast language, and its powers are limited only by our own imaginations and ingenuity.

When I sat down to write the *Java Succinctly* e-books, I was tempted to fill them with small, unrelated demonstrations of the various mechanisms of Java. This would've been an easy e-book to write, and it would be useful as a reference. But such e-books are not the kind of references I wish I'd read when I was learning. So, instead of manuals, I decided on something more ambitious. I decided to share some of the techniques involved with wrangling the mechanisms into coherent projects. With the Calculator app and the Space game, we explored a different level of programming—the most important level of all—putting a language's features together into a coherent project. This level of programming makes or breaks a student. It is not difficult to learn the syntax of any language, but the techniques for controlling the mechanisms together are what separates a student from a programmer. After you develop several applications on a similar scale to our Calculator and Space game, you should feel comfortable with using Java at this level. And, luckily, these techniques are readily transferable to any other language. After you become fluent and comfortable in Java, you will find it very easy to move to other languages.

If Java is your first language, I highly recommend that as you become comfortable with it, you explore the magic and power of the parent languages: C and C++ (which are native languages). C and C++ have syntax that is very similar to Java, but they lack the safety mechanisms and garbage collection. However, what they lack in safety, they make up for in power and speed (plus, using JNI (the Java Native Interface), you are able to combine these native languages with your Java code). If you are interested in programming web applications, I recommend you study JavaScript (which is not related to Java, despite the name). And, if you are interested in mobile development, you will be happy to know that Android mobile devices primarily run Java applications—it is a very small step from where we left off in our Space game to developing full Android applications!

Finally—thank you for reading. I hope you have a beautiful day, and I hope to see you again shortly in my next book: *Scala Succinctly*!