

JAVA SERIES

JAVA CRASH COURSE

The Complete Beginner's Course
to Learn Java Programming
in 20 Simple Lessons

ALPHY BOOKS

JAVA CRASH COURSE

The Complete Beginner's Course to
Learn Java Programming in 21 Clear-
Cut Lessons - Including Dozens of
Practical Examples & Exercises

By Alphy Books

Copyright © 2016

All rights reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in reviews and certain non-commercial uses permitted by copyright law.

Trademarked names appear throughout this book. Rather than use a trademark symbol with every occurrence of a trademark name, names are used in an editorial fashion, with no intention of infringement of the respective owner's trademark.

The information in this book is distributed on an "as is" basis, exclusively for educational purposes, without warranty. Neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this book.

Table of Contents

Greetings and Welcome

Chapter 1: Let's Start From The Beginning

Short History

What is Java?

Chapter 2: Java Environment

Installation of Java

Chapter 3: Java Language Structure

Chapter 4: Variables

What is a Variable?

Variable Types

Instance Variables

Class/Static Variables

Local Variables

Data Types

Byte

Short

Int

Long

Float

Double

Boolean

Char

[String](#)
[Declaring a Variable](#)
[Using a Variable](#)
[Assignment](#)

Chapter 5 : Operators

[The Arithmetic Operators](#)

[Addition](#)

[Subtraction](#)

[Multiplication](#)

[Division](#)

[Modules](#)

[Post-Incrementation](#)

[Pre-Incrementation](#)

[Post-Decrementing](#)

[Pre-Decrementing](#)

[Assignment Operators](#)

[Equal Sign: =](#)

[Add-Equal Sign: +=](#)

[Subtract-Equal Sign: -=](#)

[Multiplication-Equal Sign: *=](#)

[Division-Equal Sign: /=](#)

[Modules-Equal Sign: -=](#)

Chapter 6 : User Input: Getting Data in Runtime

Chapter 7 : The String Object

[String Length](#)

[Concatenating Strings](#)

[String Buffer](#)

[Constructor](#)

[StringBuffer Methods](#)

Chapter 8 : Boolean Logic

[Conditional Statements](#)

[If Statement](#)

[If Else Statement](#)

[Ladder If Statement](#)

[Nested If Statement](#)

[If Statements Examples](#)

[Relational Operators](#)

[The Logical Operators](#)

[Combining Operators](#)

[Assignment](#)

[Answer and Explanation](#)

Chapter 9 : Loops and Arrays

[Loops](#)

[While Loop](#)

[Do While Loop](#)

[For Loops](#)

[Arrays](#)

[Declaring Arrays](#)

[Multidimensional Arrays](#)

[Combining Loops and Arrays](#)

[Assignment](#)

[Answer and Explanation](#)

Chapter 10 : Methods

[Creating a Method](#)

[Types of Methods](#)

[Syntax of Static Method](#)

[Syntax of Non-Static Methods](#)

[Parameters](#)

[Method Overloading](#)

[Assignment](#)

[Chapter 11 : Inheritance and Polymorphism](#)

[Chapter 12 : Interfaces and Abstract Classes in Java](#)

[Chapter 13 : Packages](#)

[Types of Packages](#)

[Pre-Defined Packages](#)

[User Defined Packages](#)

[Chapter 14 : Debugging](#)

[Chapter 15 : Enums](#)

[Chapter 16 : Generic Types](#)

[Chapter 17 : Threading](#)

[Life Cycle of Thread](#)

[New Born State](#)

[Runnable State](#)

[Blocked State](#)

[Dead State](#)

[Creation of Threads](#)

[Common Methods](#)

[Thread Class Methods](#)

[Thread Priority](#)

[Chapter 18 : Graphical User Interface \(GUI\)](#)

[Applet](#)

[Structure of Applet Program](#)

[Applet Tag](#)

[Parameter Tag](#)

[Classes in Applet Programming](#)

[Graphics Class](#)

[Font Class](#)

[Color Class](#)

[Component Class](#)

[Menu](#)

[Dialog Box](#)

[File Dialog](#)

[JFC Swing](#)

[JFrame](#)

[JPanel](#)

[JLabel](#)

[JButton](#)

[Layout Managers](#)

[BorderLayout](#)

[BoxLayout](#)

[CardLayout](#)

[FlowLayout](#)

[GridBagLayout](#)

[GridLayout](#)

[GroupLayout](#)

[SpringLayout](#)

[Chapter 18 : Event Handling](#)

[Chapter 19 : JDBC Programming](#)

[Important Queries in SQL](#)

[Database Creation](#)

[Steps Involved in Connection Creation](#)

[Methods Involved in Database](#)

[Inserting the Data](#)

[Displaying Table Data](#)

[Chapter 20 : Exception Handling](#)

[Predefined Exception Classes](#)

[Chapter 21 : File Handling](#)

[Reading Files](#)

[Writing to Files](#)

[Chapter 22 : Example Programs](#)

[Problem N°1](#)

[Solution N°1](#)

[Problem N°2](#)

[Solution N°2](#)

[Problem N°3](#)

[Solution N°3](#)

[Problem N°4](#)

[Solution N°4](#)

[Problem N°5](#)

[Solution N°5](#)

[Bonus Chapter : Algorithms](#)

[Multiples of 3 and 5](#)

[Even Fibonacci Numbers](#)

[Largest Prime Factor](#)

[Final Words : Where to Go From Here](#)

***“I would love to change the world, but
they won't give me the source code”***

Unknown Author

Greetings and Welcome

Hello and welcome to your new programming language. This book is an introduction to programming using Java. We'll focus on breadth rather than depth. The goal is to be able to do something interesting as quickly as possible. We'll start out with fairly detailed explanations because it's important to have a thorough understanding of the basics, but as we go on, we'll have to leave a lot of stuff out. Some of the topics could have an equally long guide themselves. You don't need to understand all aspects of everything you read; what you should retain is knowledge that a particular capability exists and what it is called so that you can look it up when you need it. This is what most programmers do.

The first few sections of the book are fairly self-contained with simple examples. You should certainly not read them without typing them into the interpreter; you'll get even more out of the tutorial if you experiment with extending the examples yourself.

The remainder of the tutorial is structured around building a program that does something interesting. This will allow us to touch on many aspects of programming that are necessary to write real world programs: reading and writing to disk; error handling; code organization into classes, modules, and packages; regular expressions; and user input. We'll also touch on some general principles in programming, such as clarity and efficiency.

Let's get started!

Alphy Books

Chapter 1

Let's Start From The Beginning

Short History

We should start saying that Java is a programming language that was created by James Gosling from Sun Microsystems (Sun) in 1991 and first made publicly available in 1995, after Sun Microsystems was inherited by Oracle. The platform was originally designed for interactive television, but it surpassed the technology and design of the digital cable television industry at the time. Today, Java remains an open-source programming language that falls under the GPL (General Public License)

The language derives much of its syntax from C and C++, but lacks the power of those languages because it asks less of the user (less customization, more simplicity). For example, tasks such as garbage collection (the process of reducing memory being used by the program) are automated in Java.

Five principles were used in the creation of the Java programming language:

It must be “simple, object-oriented, and familiar”

It must be “robust and secure”

It must be “architecture-neutral and portable”

It must execute with “high performance”

It must be “interpreted, threaded, and dynamic”

An important design goal that was a key factor in Java’s sudden popularity is portability. In this context, “portability” means that code written in Java can be executed on any hardware, using any operating system.

Java was built as an exclusively object-oriented programming language—which doesn’t mean much right now, but will later in this guide. For now, suffice it to say that object-oriented programming allows for the creation of efficient, organized, and powerful code. Simply put, Java is a multithreaded, object-oriented, platform-independent programming language. This means that Java programs can perform multiple tasks using object-oriented concepts that can work across all platforms and operating systems. It is the most

important factor distinguishing Java from other languages.

Java helps us to develop normal desktop applications, mobile applications, and web applications through the use of separate packages such as the J2ME package for mobile application development and the J2EE package for web application development.

In this guide, we are going to learn the basics of object-oriented concepts as they apply to Java programming. We have two different types of application development concepts in Java: console-based application and GUI application development. Let's see how to develop these types of applications using Java.

What is Java?

Java is a programming language that is supported by all devices, whether it is an Android phone, a Windows computer, or an Apple product. Java's flexibility has made it one of the most popular programming languages around the globe. Java can be used to create web applications, games, Windows applications, database systems, Android apps, and much more.

Java's combined simplicity and power makes it different from other programming languages. Java is simple in that it doesn't expect too much from the user in terms of memory management or dealing with a vast and complex hive of intricate classes extending from each other. Although this doesn't make much sense right now, it will once we start learning about inheritance in Java.

A Java program is run through a Java Virtual Machine (JVM), which is essentially a software implementation of an operating system that is used to execute Java programs. The compiler (process of converting code into readable instructions for the computer) analyzes the Java code and converts it into byte code, which then allows the computer to understand the instructions issued by the programmer and execute them in the appropriate manner.

The distribution of the Java platform comes in two packages: the Java Runtime Environment (JRE) and the Java Development Kit (JDK). The JRE

is essentially the Java Virtual Machine (JVM) that runs Java programs. The JDK, on the other hand, is a fully featured software development kit that includes the JRE, compilers, tools, etc.

A casual user who only wants to run Java programs on their machine would only need to install the JRE, as it contains the JVM that allows Java programs to be executed. However, a Java *programmer* must download the JDK. We will explore these concepts in greater detail in the next part. As previously stated, Java programming creates an object-oriented and platform-independent program because the Java compiler creates a .class file instead of an .exe file. This .class file is an intermediate file that has byte code, and this is the reason why Java programs are platform independent. However, there are also disadvantages: Java programs take more time to complete their execution because the .class file must first load in the JVM before they are able to run in the OS.

We can develop all kinds of applications using Java, but we need to use separate packages for separate application developments. For example, if you want develop a desktop application, then you need to use JDK; if you want to develop an Android application, then you need to use Android SDK, because they have different sets of classes.

Chapter 2

Java Environment

Installation of Java

In order to install Java on your system, you need the following tools:

- IDE for Java Developers
- Java JDK

Downloading these two tools will put you on your way to becoming a Java programmer. An IDE (Integrated Development Environment) is a packaged application program that contains the necessary tools for processing and executing code. It contains a code editor, a compiler, a debugger, and a Graphical User Interface (GUI). There are many different types of IDE, but the most commonly used ones are:

- Netbeans
- Eclipse

I personally recommend using Eclipse because of its simplistic nature. It can be downloaded here:

<https://eclipse.org/downloads/>

Once you have reached this link, you will have to find this:



Then, select either the Windows 32 or 64 Bit OS, depending on the type of OS/processor you have installed in your system.

Once the IDE has been installed, we'll download and install the JDK, which will allow us to interact with the coding editor that we'll use to create and execute Java code. The JDK available at the link below contains all the packages and tools you'll need to develop Java Programs.

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

At that site, find and click this image to start the download:



After downloading and installing the JDK, you will be able to launch Eclipse.

The folder that was extracted from the Eclipse download will contain the file eclipse.exe. Once this has been launched, you will be met with the following window:

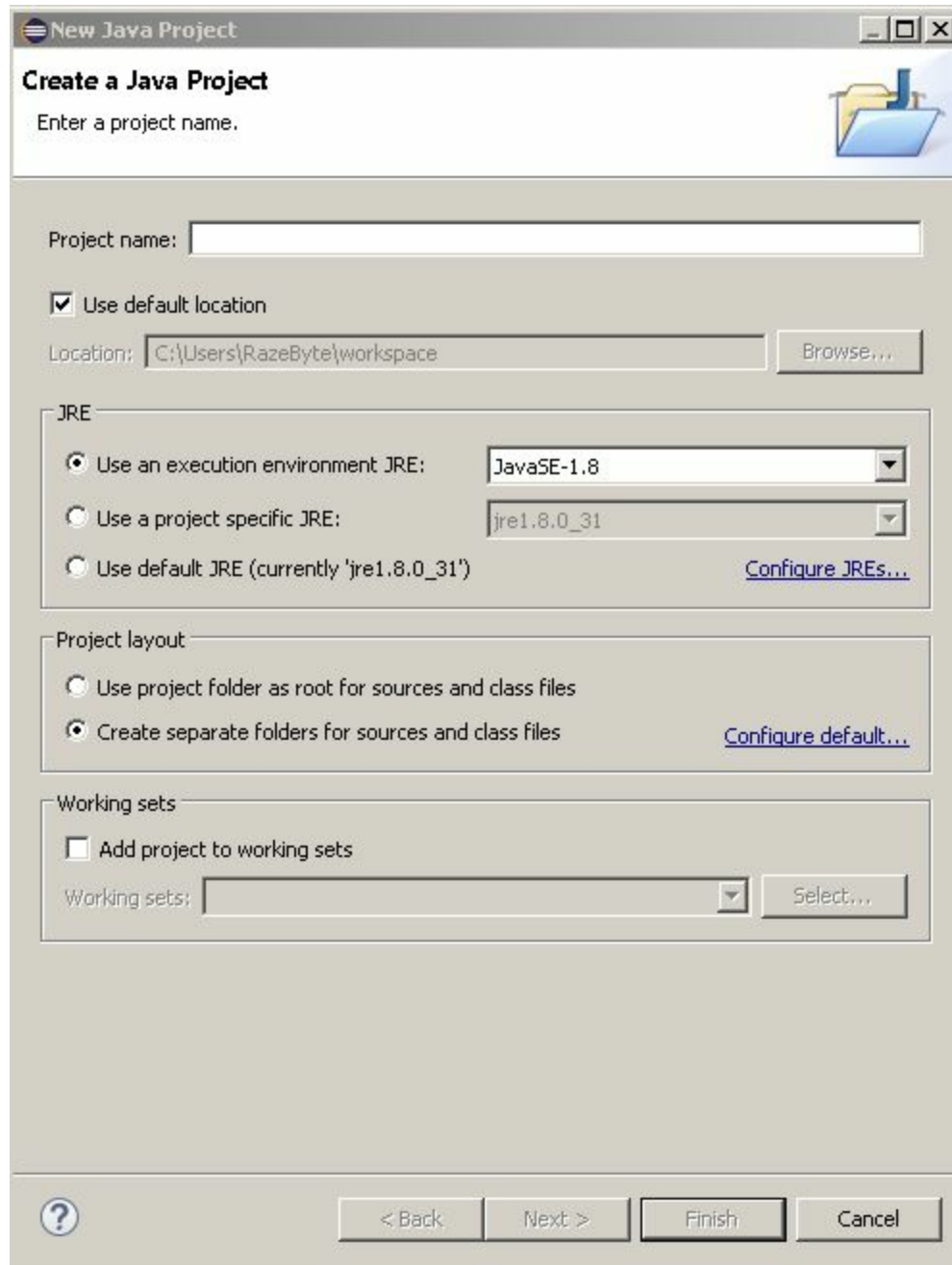


All this prompt asks for is where you want to set up the output directory to save all the code you are going to write. Once this has been selected, click “OK” to continue.

You should now be on the main screen.
In order to start working you should:

Click File

Click New → Java Project



Type a project name in the “Project name:” field

Click “Finish”

Right-click “src” → New → Class

New Java Class

Java Class
Create a new Java class.

Source folder: Beginner/src Browse...

Package: (default) Browse...

☐ Enclosing type: Browse...

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

- ☐ public static void main(String[] args)
- ☐ Constructors from superclass
- ☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

- ☐ Generate comments

Finish Cancel

Fill in the “Name” field using anything with letters—no special characters.
Example: “ThisIsAClass”

Click “Finish”

You will now be presented with a screen that says:

```
p ublic class ThisIsAClass {  
  
}
```

Congratulations, you have successfully installed Java!

Chapter 3

Java Language Structure

We will now use this sample code as an example to start the Java learning process. This code should make it easy to understand the basic structure of a Java program.

```
import java.util.Scanner;
public class ThisIsAClass {
    public static void main (String args[]) {
        int x = 5;
        System.out.println(x);
    }
}
```

The first line `import java.util.Scanner;` it is using the special keyword `import`, which allows the programmer to import tools in the Java library that aren't included by default when starting a new class. After the keyword `import`, the programmer specifies a specific directory within the Java library for the Scanner tool by typing out: `java.util.Scanner`, which first accesses the Java library in the Utilities directory before accessing the specific tool needed, which in this case is "Scanner."

By default, when starting a Java class, only the bare minimum tools and functions from the Java library that are needed for any basic program will be provided. For example, you don't need to import packages for a simple program. If the programmer wants to use more than just the basic functionalities, they must use the "Import" keyword to give themselves more tools to work with.

You will also start to notice that there is a semicolon ";" after each statement. This semicolon functions as a period does for an English sentence. When the compiler is going through the program to prepare it for its execution, it will check for semicolons, so it knows where a specific statement ends and a new one starts.

The next thing you'll notice in the sample code is: `public class ThisIsAClass`.

There are three elements to this line that are very important to understand.

`public`—Defines the scope of the class and whether or not other classes have access to the class. This may not make sense now, but you will gain a better understanding of what this means when we learn about “Inheritance and Polymorphism.”

`class`—A class can be thought of as a “section” of code. For example:

Section {everything here is the content of the section}

Again, you will gain a better understanding of how classes can be useful when we learn about Inheritance and Polymorphism.

`ThisIsAClass`—This third and final element of this important line is “`ThisIsAClass`,” which is simply a custom name that the user can define. You can call this anything, as all it does is give the “Section” or “Class” a name. You can think of it this way:

```
Section:      Name      {  
    Essay/Contents  
}
```

In code, it would be presented:

```
Class      ThisIsAClass      {  
Code  
}
```

Another thing you may be scratching your head over are the curly braces: “{” and “}.” All that these characters do is tell the compiler where a specific section starts and ends. In English, this could be thought of as starting a sentence with a capital letter or indenting a new paragraph.

One thing to note is that the spacing in code does not affect whether or not it works. However, it is conventional and efficient to write properly spaced code so that you and other programmers can read and understand it more

easily. You will learn how to space as you read more sample code. Eventually, you will start to notice trends in how conventional spacing works in programming.

```
public class ThisIsAClass {  
public static void main (String args[]) {  
int x = 5;  
System.out.println("The number is" + x);  
}  
}
```

As shown in the above sample code, two curly braces are set in bold to indicate that they communicate with each other. Whatever is written between them is the “Section” or “Class” of name “ThisIsAClass.” The sample applies to the bold and underlined curly braces to indicate that they are separate section dividers for the section within the parent section. The parent section is again considered a class, whereas the section within the parent section is considered a sub-section, or a subclass, or the formal term, a method.

A method is essentially a subclass that also has its own special elements, similar to a class but more specific. This contains four elements: a scope (public/private/protected), a return type, a name, and parameters. The scope, return type, and parameters are things you will understand better when we learn about Methods, along with Inheritance and Polymorphism.

public—Scope.

static—Conventional keyword for the main method.

void—Return type.

main—Name of the method (you can call this anything, but in this case, the main method must always exist in every Java program because it is the starting point of the Java program).

This method is a special method because it is named “main” and so it will be the first method that is called. The compiler always looks for the method named “main” at the process start. The main method must hold the following properties: “public static void” and have “String args[]” as the argument in the parameters. There can only be one main method in a Java project. The

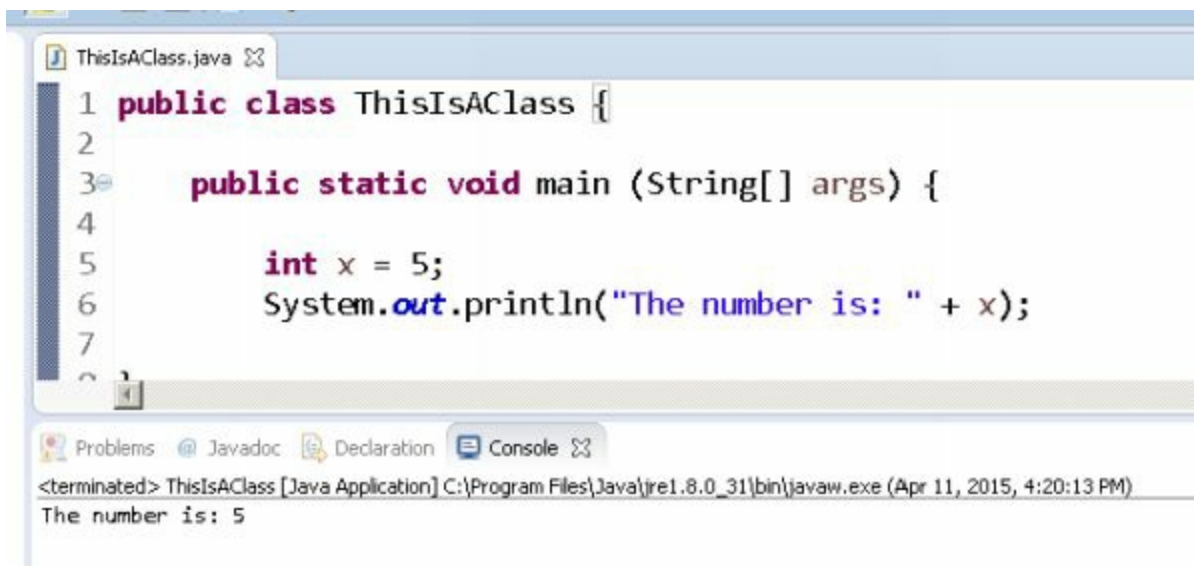
idea of the main method is that it will be the first method to be called, and so you can think of it as the home base for calling other methods and traveling to other classes (you will learn what this means later). For the remainder of this guide, your program must be written within this main method, as it is a special method that the compiler will always be looking for.

The contents of this method will also be introduced when learning about “Variables” later on in this guide. For now, this brief explanation should be enough for you to understand what is going on:

`int x = 5;`—A variable is being declared (a container) that is a type of integer (whole number), and it holds the value of five. The “=” is used to assign values to variables.

`System.out.println`—This is a classic line that every beginner learns; all it does is print a line of text to the console.

Example:

A screenshot of a Java IDE window titled 'ThisIsAClass.java'. The code editor shows the following code:

```
1 public class ThisIsAClass {  
2  
3     public static void main (String[] args) {  
4  
5         int x = 5;  
6         System.out.println("The number is: " + x);  
7  
8     }  
9 }
```

The bottom of the window features a 'Console' tab with the output: `<terminated> ThisIsAClass [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (Apr 11, 2015, 4:20:13 PM)` followed by `The number is: 5`.

The console is where the program’s output is placed in order for the programmer to test his or her program. The “`System.out.println`” line contains parenthesis and a string of text within those parenthesis. The string of text must be surrounded by quotation marks in order for the program to know that it has to print out text instead of an actual variable like “x.” The

reason the user can't just say: `System.out.println("The number is: x ");` is because the program won't know if `x` is a variable or not. The program reads anything within quotation marks as a piece of text instead of a container holding a value. Therefore, the `x` must be outside of the quotation marks.

Then again, you can't say `System.out.println("The number is: " x);` because the program needs a specific keyword to know that the piece of text and the variable are two entities that need to be joined or "concatenated" together. The `+` symbol is used to show that the string of text and the variable "`x`" are to be connected together; this will allow the program to output "The number is: 5" (without the quotation marks). That is why the line is: `System.out.println("The number is: " + x);`

Commenting:

Commenting is the final fundamental concept to understand in a programming language. Although it is not necessary, it can greatly help you and other programmers around you if you ever forget how the program works.

This concept is used by programmers as an opportunity to explain their code using common English. A compiler will always ignore comments, as they aren't actual code. They are simply explanations that were written to help people understand what was programmed. In order to write a comment, you must use `/**` or `/*` and `*/`.

The `/**` symbol is used to comment on one line. For example, if you were to explain what a certain variable was going to be used for, you would do the following:

```
int x = 5; /** this variable is going to be used to find the total money
```

The compiler will ignore the commented line, but will process `int x = 5`. However, you can't use the `/**` symbol with the following:

```
int x = 5; /** this variable is going to be  
used to find the total money
```

This is because the “//” symbol is only used for one line, and the comment is on two lines. You could do:

```
int x = 5;  
// this variable is going to be used to find the total money
```

As long as the comment is one line, it is fine. Anything on the line after that symbol is considered a comment.

The other technique does the same thing as “//” except it supports multi-line commenting. You must start the comment with “/*” and end it with “*/.”

Example:

```
int x = 5; /* this variable is going to be  
used to find the total money */
```

Chapter 4

Variables

What is a Variable?

A variable is essentially a storage unit that holds a certain type of data. It is named by the programmer and used to identify the data it stores. A variable can usually be accessed or changed at any time. You can write information to it, take information from it, and even copy the information to store in another variable.

Variable Types

In Java, a variable has a specific type that determines what size it is and the layout of its memory. The Java language defines three types of variables.

Instance Variables

Instance variables are declared within a class but accessed outside of any method, constructor, or block. Instance variables are created with the keyword “new” and are destroyed when the object is destroyed. An instance variable is visible to all methods, constructors, and blocks within the class where it is declared. By giving a variable the keyword “public,” it can be accessed by subclasses within the class. However, it is recommended to set a variable’s access modifier to “private” when possible. All instance variables have a default value, unlike local variables, which do not have default values.

Example:

```
public class Test
{
    public int num; // integer named num
    public void dog()
    {
        num = 3;
    }
}
```

```
}
```

In this example, we declare the integer “num” outside of any method, and can easily access it within any method that is inside of the class “Test.”

Class/Static Variables

Static variables are declared outside of any method, constructor, or block, and are declared with the keyword “static.” If a variable is static, only one instance of that variable will exist, regardless of how many instances of the object are called. Static variables are rarely used except for constant values, which are variables that never change.

Example:

```
public class Test
{
    public static final int num = 3; // integer named num
    public void dog()
    {
        System.out.println( num );
    }
}
```

In this example, we declare an int called “ *num* ” and it set to be public, static, and final. This means that it can be accessed from within subclasses, only one instance of it can ever exist, and the value can never be changed.

Local Variables

Local variables are only declared within methods, blocks, or constructors. They are only created when the method, block, or constructor is created, and then they are destroyed as soon as the method ends. You can only access local variables within the method, block, or constructor where it is called; they are not visible outside of where they are called. Local variables do not have a default value.

Example:

```
public void cat(){ // method named cat  
int x = 0; // int with value of 0  
}
```

In this example, a variable named “x” is called within the method “cat.” That variable only exists within the context of cat, and it cannot be directly called or accessed outside of that method.

Data Types

As said above, variables are used to store data. Java has multiple built-in data types that are used to store predefined types of data. These data types are called primitive data types, and are the most basic data types in the Java programming language. You can also create your own data types, which we will go over later. Java has eight primitive data types.

Byte

The byte data type is an 8-bit signed two’s complement integer. It has a default value of zero when it is declared, a maximum value of 127, and a minimum value of -128. A byte is useful when you want to save memory space, especially in large arrays.

Example:

```
byte b = 1; // has a value of one
```

Short

The short data type is a 16-bit signed two’s complement integer. Its maximum range is 32,767 and its minimum value is -32,768. A short is used to save memory or to clarify your code.

Example:

`short s = 1; // has a value of one`

Int

The int data type is a 32-bit signed two's complement integer. Its maximum value is 2,147,483,647 ($2^{31} - 1$) and its minimum value is -2,147,483,648 (-2^{31}). Int is the most commonly used data type for integral numbers, unless memory is a concern.

Example:

`int i = 1; // has a value of one`

Long

The long data type is a 64-bit signed two's complement integer. Its maximum value is 9,223,372,036,854,775,807 ($2^{63} - 1$) and its minimum value is -9,223,372,036,854,775,808 (-2^{63}). This data type is used when a larger number is required than would be possible with an int.

Example:

`long l = 1; // has a value of one`

Float

The floating point data type is a double-precision 64-bit IEEE 754 floating point. The min and max range is too large to discuss here. A float is never used when precision is necessary, such as when dealing with currency.

Example:

`float f = 1200.5f; //value of one thousand two hundred, and a half`

Double

The double point data type is a double-precision 64-bit IEEE 754 floating

point. It is often the data type of choice for decimal numbers. The min and max range is too large to discuss here. A float is never used when precision is necessary, such as when dealing with currency.

Example:

```
double d = 1200.5d; // value of one thousand two hundred, and a half
```

Boolean

A boolean data type represents one bit of data. It can contain two values: true or false. It is used for simple flags to track true or false conditions.

Example:

```
boolean b = true; // has a value of true
```

Char

The char data type is a single 16-bit Unicode character. Its minimum value is “/u0000 ” (or 0) and its maximum value is “/uffff” (or 65,535 inclusive)

Example:

```
char c = 'w'; // returns the letter "w"
```

String

Another commonly used data type is called String. String is not a primitive data type, but it is a very commonly used data type that stores collection of characters or text.

Example:

```
String cat = "meow"; // sets value of cat to "meow"
```

This example gets a String named “cat” and sets it to the string of characters

that spell out “meow.”

Declaring a Variable

The declaration of a variable has three parts: the data type, variable name, and the stored value. Note that there is a specific convention and set of rules that are used when naming variables. A variable name can be any length of Unicode letters and numbers, but it must start with a letter, the dollar sign “\$,” or an underscore “_,” or else it will return a syntax error. It is also common naming convention to start the name with a lowercase letter, followed by each subsequent word starting with a capital letter. For example, in the variable named “theName,” the first word “the” starts with a lowercase letter, and each following word, in this case “Name,” starts with a capital letter.

Example:

```
int theName = 123; // value of 123
```

In the example above, the data type is `int` and the name of the variable is `theName`. The value stored inside that variable is 123. You can also declare a variable without storing a value in it.

Example:

```
int name; // no value
```

You can do this if you choose to declare it later.

Using a Variable

After a variable is declared, then you can read its value or change it. After the variable has been initially declared, you can only reference it by its name; you only need to declare its data type when you are declaring the variable.

Example:

```
name = 2; // sets the int "name" to a value of 2
```

The example above sets the value of name to 2. Notice how I never restated the data type.

Example:

```
System.out.println(name); // prints the value of name to the console
```

This example reads the value of “name” and writes it to the console.

Variables can also be added together for example.

Example:

```
int a; // no value  
int b = 1; // value of one  
int c = 2; // value of two  
a = b + c; // sets a to the value of b + c
```

In the example above, we set the value of a to equal the value of b and c added together. The addition sign is known as an operator, which we are going to learn about in the following section. It is also possible to a certain extent to combine values of variables that are different data types.

Example:

```
int a; // no value  
float b = 1; // value of one  
int c = 2; // value of two  
a = (int) (b + c); // sets the int "name" to a value of b + c
```

This example is just like the one before, except we have changed the data type of b from int to float. The only difference when adding them together is that we had to include something called a “cast” in the equation. What a cast does is simply let the compiler know that the value of (b + c) should be of the data type int. Note that for this example, if the value of b + c were to equal a

decimal number (for example 3.2), the value of a would not be 3.2 but rather 3, because int does not support decimals.

Assignment

Using what we have learned about variables, we can now create a simple calculator to add numbers together for us. The first thing we will want to do is declare three variables: one to store the value, one to represent the first number we want to add, and one to represent the second number we want to add. We will declare these variables as double so that we can add decimal numbers:

```
double a = 0; // stores value of addition
double b = 3.55; // first number to add
double c = 52.6; // second number to add
```

Next, we will simply set the value of a to equal the value of b and c combined, and then print out the value of a.

```
a = b + c;
System.out.println(a);
```

If you run this program, it will print out 56.15. Now you have created a very simple calculator. I highly encourage you to play around with this and test things for yourself. Change the data type of the variables, add more numbers together, and experiment to understand how things work.

Chapter 5

Operators

In math class, you learned about addition, subtraction, multiplication, division, etc. These are all arithmetic operators that are used within programming to intake numbers, process them, and calculate them accordingly. Let's go over these operators in programming, as they are one of the most important things to understand and also one of the easiest to grasp.

The Arithmetic Operators

Addition

```
5+5 = 10
int x = 5;
int y = 5;
int sum = 0;
sum = x + y;
```

In the example above, the variable sum is taking the two variables x and y and adding them together to produce a value of 10.

Subtraction

```
10-5 = 5
int x = 10;
int y = 5;
int total = 0;
total = x - y;
```

In the example above, the variable total is taking the two variables x and y and subtracting them to produce a value of 5.

Multiplication

```
5*4 = 20
```

```
int x = 5;  
int y = 4;  
int total = 0;  
total = x * y;
```

In the example above, the variable total is taking the two variables x and y and multiplying them to produce a value of 20.

Division

```
20/5 = 4  
int x = 20;  
int y = 5;  
int total = 0;  
total = x / y;
```

In the example above, the variable total is taking the two variables x and y and dividing them to produce a value of 20.

Modules

```
7 % 2 = 1  
int x = 7;  
int y = 2;  
int total = 0;  
total = x % y;
```

In the example above, the variable total is taking the remainder of the two variables x and y and calculating how many times 2 multiplies in to 7 evenly before it can't. After that process, the remainder is the output. For example:

How many times does 2 go into 7 evenly?

3 times.

$2 * 3 = 6$

$7 - 6 = 1$

Therefore, 7 modules 2 is equal to 1 because that is the remainder.

Post-Incrementation

```
5 + 1 = 6  
int x = 5;  
x++;  
int y = 0;  
y = x++;
```

In the example above, the variable `x` is being incremented by a value of 1. Therefore, the value of `x` must now be 6 because $x + 1 = 6$ because $x = 5$. If you were to print out the value of `x`, the output would be 6. You can do `x ++` again and the value would then become 7. The value of `y`, on the other hand, is still 6 because it is post-incrementing, meaning that the value of `x` has changed by an increment of 1 (to 7); however, the value of `y` has yet to be changed until it is called again (`y = x ++`), which in this case would be `y = 7`.

Pre-Incrementation

```
5 + 1 = 6  
int x = 5;  
++x;  
int y = 0;  
y = ++x;
```

In the example above, the variable `x` is being incremented just as it was when it was post-incrementing as a variable by itself. The value of `y`, on the other hand, changes directly to 6 because it is pre-incrementing. What this means is that the value of `x` has changed by an increment of 1, and the value of `y` changes, meaning that `y = 7`.

Post-Decrementing

```
5 - 1 = 4  
int x = 5;  
x--;  
int y = 0;  
y = x--;
```

In the example above, the variable `x` is being decremented by a value of 1. Therefore, the value of `x` must now be 4 because $x - 1 = 4$, since `x` used to equal 5. If you were to print out the value of `x`, the output would be 4. You can do `x --` again and the value would then become 3. The value of `y`, on the other hand, is still 4 because it is post-decrementing, meaning that the value of `x` has changed by a decrement of 1. The value of `y` still has yet to be changed until it is called again (`y = x --`), which in this case would be `y = 3`.

Pre-Decrementing

```
5 - 1 = 4
int x = 5;
--x;
int y = 0;
y = --x;
```

In the example above, the variable `x` is being decremented by a value of 1. Therefore, the value of `x` must now be 4 because $x - 1 = 4$, since `x` was equal to 5. If you were to print out the value of `x`, the output would be 4. You can do `--x` again and the value would become 3. The value of `y`, on the other hand, is 3 because it is pre-decrementing. This means that the value of `x` has changed by a decrement of 1, and the value of `y` has changed.

Assignment Operators

The assignment operators are operators that are used when assigning values to variables, the most commonly used operator being `(=)`. Here are a list of examples:

Equal Sign: `=`

```
int x = 5;
```

In this example, if you print out `x`, the value would be 5 because you assigned the variable `x` as equal to 5.

Add-Equal Sign: +=

```
int x = 5;  
x += 5;
```

In this example, if you print out x, the value would be 10 because you are adding the value of 5 to the value of x. This statement is the same as saying $x = x + 5 \rightarrow x += 5$.

Subtract-Equal Sign: -=

```
int x = 5;  
x -= 5;
```

In this example, if you print out x, the value would be 0 because you are subtracting the value of 5 from the value of x. This statement is the same as saying $x = x - 5 \rightarrow x -= 5$.

Multiplication-Equal Sign: *=

```
int x = 5;  
x *= 5;
```

In this example, if you print out x, the value would be 25 because you are multiplying the value of 5 by the value of x. This statement is the same as saying $x = x * 5 \rightarrow x *= 5$.

Division-Equal Sign: /=

```
int x = 5;  
x /= 5;
```

In this example, if you print out x, the value would be 1 because you are dividing the value of 5 by the value of x. This statement is the same as saying $x = x / 5 \rightarrow x /= 5$.

Modules-Equal Sign: %=


```
int x = 5;  
x %= 5;
```

In this example, if you print out x, the value would be 0 because you are finding the remainder of (5÷5). This statement is the same as saying $x = x \% 5 \rightarrow x \% = 5$.

Chapter 6

User Input: Getting Data in Runtime

The Scanner object in Java will allow us to input information to the Console so the program will be able to process and use it. We have some other classes to read the data at runtime.

How it works:

Input → Process → Output

To explain how the Scanner object works, we'll look at a real life application: a calculator that takes in two values through the use of the console.

The first thing you want to do is import the object with the following line:

```
import java.util.Scanner;
```

The next line you have to code, assuming your class and main method are already set, will declare the Scanner object by doing the following:

```
Scanner scan = new Scanner(System.in);
```

You will learn what the “new” keyword means in the section on Inheritance and Polymorphism, but for now, just know that this is a conventional means of declaring a Scanner object. Now you must decide what you would like the user to input. A string? An integer? In our situation, we want the user to input an integer for this calculator to add two integers together.

The Scanner object has different properties and methods (you will learn what these are in the sections on methods and inheritance/polymorphism). The method we're looking for is `scan.nextInt()`, which is a method that waits for a value from the user. In order to create a variable that receives the value of whatever the user types in, you must type the following (variable names are optional):

```
int firstNumber = scan.nextInt();
```

```
int secondNumber = scan.nextInt();
int sum = firstNumber + secondNumber;
System.out.println(sum);
```

So what exactly does this program do?

It declares two integers that both wait for user input. First, it waits for the input of the first number, i.e., for the user to press enter after typing a number. Then it waits for the second number to be inputted. Finally, the sum variable adds the two variables that contain values given by the user and outputs it to the console. All of this takes place in the Java Eclipse/IDE console.

Instead of the Scanner object, we can also use the BufferedReader and DataInputStream class methods to read different types of data at runtime. In these classes we have some set of methods that provide the things needed to read different types of data, like Integer, character, float, double, String, etc.

The BufferedReader and DataInputStream classes are defined in the java.io package. If you want to use these classes, then you will have to import these packages to your program. After importing these packages, you can use these class methods wherever you want, but you will have to create an object before calling these methods.

BufferedReader and DataInputStream classes contain the following methods:

- readInt()
- readLong()
- readFloat()
- readDouble()
- readLine()

Creating an Object for DataInputStream and BufferedReader

```
BufferedReader br=new BufferedReader (new
InputStreamReader(System.in));
String name=br.readLine (); // This will read a string data at runtime.
```

```
int no=br.readLine (); /*This will throw an error because you can't store
string data in an integer variable. You will have to convert this data to integer
data by using wrapper classes. Wrapper classes are nothing but data
conversion classes that contain some methods to convert string data to some
other equal data.*/
```

Wrapper Classes:

- Integer
- Float
- Double
- Byte

Integer.parseInt(String data)

Will convert String data to Integer data.

Float.parseFloat (String data)

Will convert String data to float data.

Double.parseDouble(String data)

Will convert String data to double data.

Byte.parseByte(String data)

Will convert String data to Byte data.

```
DataInputStream obj=new DataInputStream (System.in);
String data=obj.readLine(); // Reads string data that will then be stored inside
a string variable
int no=Integer.parseInt(data); // Will convert string data to Integer data.
Sample Program for BufferedReader:
```

```

import java.io.*;
class students
{
private int sno;
private String sname;
private int mark1,mark2,total;
students() throws Exception
{
BufferedReader      br      =      new      BufferedReader(new
InputStreamReader(System.in));
System.out.print("Enter Student No : ");
sno = Integer.parseInt(br.readLine());
System.out.print("Enter Student Name : ");
sname = br.readLine();
System.out.print("Enter mark1 and mark2 : ");
mark1 = Integer.parseInt(br.readLine());
mark2 = Integer.parseInt(br.readLine());
}
students(int no,String name,int m1,int m2)
{
sno = no;
sname = name;
mark1 = m1;
mark2 = m2;
total = mark1 + mark2;
}
void putstud()
{
System.out.println("Sno : " + sno);
System.out.println("Sname : " + sname);
System.out.println("Mark1 : " + mark1);
System.out.println("Mark2 : " + mark2);
System.out.println("Total : " + total);
}
public static void main(String args[]) throws Exception
{
students s1 = new students();

```

```

students s2 = new students(100,"Kirthika",90,80);
s1.putstud();
s2.putstud();
}
}

```

This sample program demonstrates how `BufferedReader` is used in runtime data reading. Here we have created a new class called by the name `students` with two different constructors. The first constructor is an empty constructor that doesn't have any parameters in it. This is going to read a student's details in runtime, such as student name and number, with the help of the `readLine()` method. This method reads a string data when we call it. So, first it reads a data, then it will convert that to an integer data to store inside the integer variable called by the name `sno`, then another string data will be stored inside the `sname` variable. In the second constructor, we will send input data as a parameter that will be stored in the `sno` and `sname` variables. In our main method, we are creating an object for the first and second; it will execute these constructors respectively. Then the `putstud` method will be called, and finally `sname` and `sno` data will be printed.

Sample Program for `DataInputStream`

```

import java.io.*;
class Const1
{
private int sno;
private String sname;
Const1()
{
System.out.println("Constructor Called");
sno = 0;
sname = "";
}
void getdetails() throws IOException
{
DataInputStream br = new DataInputStream (System.in);
String line;

```

```

System.out.print("Enter Sno : ");
line = br.readLine();
sno = Integer.parseInt(line);
System.out.print("Enter Sname : ");
sname = br.readLine();
}
void putdetails()
{
System.out.println("Sno : " + sno);
System.out.println("Sname : " + sname);
}
public static void main(String args[]) throws IOException
{
Const1 c1 = new Const1();
c1.putdetails();
c1.getdetails();
c1.putdetails();
}
}

```

This example explains the use of DataInputStream. In this example, we are reading the data using the readLine() method, which is defined under the DataInputStream class. It does the same work as the previous example program.

Chapter 7

The String Object

Strings are quite commonly used in Java programming. A string is a sequence of characters or a line of text. A String in Java is not a primitive data type, but rather an object, which means that it has properties that can be called from it.

A String variable is declared by doing the following:

```
String thisIsText = "example";
```

Since a String is an object in Java, it is capitalized as "String," unlike a primitive data type, which is simply all lowercase to indicate that it is primitive. An example of this is an "int," which indicates an integer. Later on, you will learn about classes and objects in more detail. Considering that a String is an object, it can also be declared like all objects.

```
String thisIsText = new String("example");
```

Now let's go over the fundamental properties of a String object that can be used to gain extra information over the piece of text or String.

String Length

One way to receive information on the number of characters in a String is by doing the following:

```
String text = "test";  
int amount = text.length();  
System.out.println("The amount of characters are: " + amount);
```

The `.length()` function or property in a String object returns a value for the amount of characters within the String object and assigns the value to the integer amount. The output in this sample code would be 4, since "test" is four characters long.

Concatenating Strings

Concatenating is the joining together of two strings. There are different ways of joining two strings together, such as the join + operator or the concat method. Here are a few examples:

Example 1:

```
String text = "Hello ";  
String text2 = "World!";  
System.out.println(text + text2);
```

Output: Hello World

Example 2:

```
String text = "Hello ";  
String text2 = "World!";  
System.out.println(text.concat(text2));
```

Output: Hello World

Example 3:

```
String text = "Hello";  
String text2 = "World!";  
System.out.println(text + " fun " + text2 + " of programming!");
```

Output: Hello fun world of programming!

Example 4:

```
String text = "Hello";  
String text2 = "World";  
System.out.println(text.concat("    fun    ").concat(text2).concat("    of  
programming!"));
```

Output: Hello fun world of programming!

As shown in the examples above, you are able to use the `.concat()` method to join two strings together, or you can use the “+” operator.

Here are various examples of the String object in action:

Example 5: `charAt(int index)`

```
String text = "Hello World!";  
char letter = text.charAt(0);
```

The value of “letter” contains the letter “H” because the inputted index of the string is 0, which is the first letter of the string. The way the computer reads the string is from 0 to the length of the string, not 1.

Example 6: `Equals(String anyText)`

```
String text = "Hello";  
boolean doesItEqual = text.Equals("Hello");
```

The value of `doesItEqual` is going to equal true in this case because the `.Equals()` method returns a boolean value (true or false) and the text does indeed equal Hello. Later on, you will learn about “if” statements, which can allow you to check whether or not something equals something else. When comparing two strings, you must always use the `.Equals()` method, but when comparing most other data types you will use “==.” You will learn more about this later on.

Example 7: `substring(int beginIndex)`

```
String text = "Hello World";  
String justWorld = text.substring(6);
```

Since the position of the letter “W” is 6, if you say `substring(6)` it will essentially divide that text starting from that position. Therefore, the “justWorld” variable ends up containing the value “World.”

Example 8: trim()

```
String text = "  Hello World  ";  
String trimmedText = text.trim();
```

As seen, the text variable has spaces in the beginning and the front. All the trim() method does is simply delete any spaces on the left or right side of the string of text. Therefore, the trimmedText would hold the value of "Hello World" instead of " Hello World ."

There are many other String methods that can be referenced using the Java API to manipulate a String. An API is simply a directory of all the methods/functionalities and objects/classes of the Java platform that can be used to help the user program. It can be referenced here:

String Buffer

StringBuffer is a peer-class string that provides much common use functionality. Where String represents fixed-length-character sequences, StringBuffer represents varied length character sequences. StringBuffer may have characters and substrings inserted in the middle or appended at the end. The compiler automatically creates a stringbuffer to evaluate certain expressions, in particular when the overloaded operators + and += are used with the string objects.

Constructor

We have three different constructors to create the stringbuffer data, each listed here with its syntax.

- `StringBuffer sb = new StringBuffer ();`

This is an empty constructor. It will create the StringBuffer object, but it won't have anything in it. You can provide those data by simply assigning some to the string buffer object.

- `StringBuffer sb = new StringBuffer(16);`

This is a parameterized constructor that has only one parameter: the size of the newly created StringBuffer. For example, this one will create a new StringBuffer object with capacity of 16 characters.

- `String s;`

```
StringBuffer sb = new StringBuffer(s);
```

This is a parameterized constructor that has only one parameter: string data object. For example, this one will create a new StringBuffer object, then acquire the data of String s.

StringBuffer Methods

- **`length()`**

Creates a string from this StringBuffer or converts to string.

Example of `length()`

```
StringBuffer text = "test";  
int amount = text.length();  
System.out.println("The amount of characters are: " + amount);
```

The `.length()` function or property in a StringBuffer object returns a value for the amount of characters within the String object and assigns the value to the integer amount. The output in this sample code would be 4, since "test" is four characters long.

- **`capacity()`**

Returns the current number of spaces allocated.

Example of Capacity

```
StringBuffer s1=new StringBuffer(16);
```

```
int capacity=s1.capacity();  
System.out.println("Capacity of the StringBuffer"+capacity);
```

The `capacity()` method will return the capacity of the `StringBuffer` object. This sample code will create the new `StringBuffer` object `s1`, then once we call the `capacity` method it will return the capacity of the new `StringBuffer` object `s1`. The result will be stored in an integer variable `capacity`. Finally, we print the capacity of `s1` with the help of the `println` method.

`boolean ensureCapacity()` makes the `StringBuffer` hold at least the desired number of spaces.

- **`setLength()`**

Truncates or expands the previous character string, if expanding pads with nulls.

Example of `setLength`

```
StringBuffer s1=new StringBuffer();  
s1.setLength (25);
```

This `setLength` method will resize the `StringBuffer`. In this example, we are resizing the data by using the `setLength` method.

- **`charAt(int)`**

Returns the character to that location in the buffer.

Example of `charAt`

```
String text = "Hello World!";  
char letter = text.charAt(0);
```

The value of "letter" contains the letter "H" because the inputted index of the string is 0, which is the first letter of the string. The way the computer reads the string is from 0 to the length of the string, not 1.

- **setCharAt(int,char)**

Modifies the value of that location.

Example of setCharAt

```
StringBuffer sb = "Hello";  
sb.setCharAt(2,'w');      Hewlo;
```

In this example we are changing the second index data by “w.”

- **getChars()**

Copies chars into an external array. There is no `getBytes()` as there is in `String`.

- **append()**

Converts the argument to a string and appends it at the end of the current buffer.

Example of append

```
StringBuffer s1=new StringBuffer(25);  
s1="Hello";  
s1.append("AABBCC");
```

In this example, `append` will add the “AABBCC” text to the end of this `s1` `StringBuffer` object so the final result will be “HELLOAABBCC.”

- **insert()**

Will convert the second arg to a string and insert it into the current buffer beginning at the offset.

Example of insert

```
StringBuffer s1=new StringBuffer("Dwayne Johnson");  
s1.insert(7," The Rock ");
```

This sample code will insert the second parameter's text to the mentioned index of the StringBuffer object so the final result will be "Dwayne The Rock Johnson."

- **reverse ()**

Will reverse the order of the characters in the buffer.

Example of reverse

```
StringBuffer s1=new StringBuffer("Saravanan");  
System.out.println("Reverse string is:"+s1.reverse());
```

This example convert the string data in reverse order; the final result will be "nanavaraS."

Example:

```
class StringBufExample  
{  
public static void main(String args[])  
{  
StringBuffer sb = new StringBuffer("Hello World");  
System.out.println(sb + ".capacity() = " + sb.capacity());  
System.out.println(sb + ".length() = " + sb.length());  
System.out.println(sb + ".insert(5,aaa) = " + sb.insert(5,"aaa"));  
System.out.println(sb + ".capacity() = " + sb.capacity());  
System.out.println(sb + ".length() = " + sb.length());  
System.out.println(sb + ".append(bbbb) = " + sb.append("bbbb"));  
System.out.println(sb + ".reverse() = " + sb.reverse());  
}  
}
```

Chapter 8

Boolean Logic

Java provides us with a vast set of operators to manipulate variables. In the following section we are going to go over Relational Operators and Logical Operators, but first you have to know how and where to use these operators. There are many different statements that can be used in order to perform and test different types of logic. We will go over some more complex statements in later sections.

Conditional Statements

If Statement

In an If statement, we have a condition. If the given condition is true, then the program will execute the true part of the If statement. If the given condition is false, then the program will not show anything.

Syntax:

```
If (condition)
{
.....
True Part ....
.....
}
```

If Else Statement

In an If Else statement, we have a condition just like with the previous If statement. But here we have a false part, too: This is the major difference between an if and If Else statement. If the given condition is true, then the program will execute the true part of the If statement. If any given condition is false, then it will execute the false part.

Syntax:

```
If (condition)
{
.....
True Part ....
.....
}
Else
{
.....
False Part .....
.....
}
```

Ladder If Statement

In a Ladder If statement, we have multiple conditions. If the first condition is false, then the program will check the next one, and so on until it reaches the Else part of the statement. If any conditions become true, then that true part will be executed and immediately come outside of the block. If it is false, then control passes to the next statement. The Else part will only execute once all given conditions become false.

Syntax:

```
If (condition)
{
.....
True Part ....
.....
}
else if (condition)
{
.....
True Part .....
.....
}
```

```

}
else if (condition)
{
.....
True Part .....
.....
}
.....
.....
else
{
.....
false Part .....
.....
}

```

Nested If Statement

In a Nested If statement, we have multiple If conditions nested inside one another. If the first condition becomes true, then the program will check the second condition that is defined inside of the first condition. This process continues until the last condition becomes false: the Else part only executes once the first condition become false.

Syntax:

```

If (condition)
{
If (condition)
{
if (condition)
{
.....
True Part. . .
.....
.....
}
}
}

```

```

}
Else
{
.....
False Part .....
.....
}

```

If Statements Examples

For the following examples, I will use an if statement. An if statement, simply put, checks to see IF something is true or false. To create an if statement, you simply write the word if followed by two rounded brackets. This will contain the logic to check IF something is true or not. if statements can also include else if statements, and else statements. An else if statement will run if the if statement returns false. However, it also contains its own parameter that must return true. An else statement also is called if the if statement returns false. The difference between an else statement and an else if statement is that no parameters need to be true for an else statement to run.

Example:

```

int a = 10;
int b = 20;
if (a == b) {
System.out.println("a is equal to b ");
}
else if (a > b) {
System.out.println("a is greater than b ");
}
else {
System.out.println("a is not equal to b or greater than b ");
}

```

The operands used in this example (“==” and “>”) are explained below. In the following code, we first check to see if “a” is equal to “b.” If that is true, we print “a is equal to b” and ignore the rest of the code. If “a” does not equal

“b,” then the program calls the else if. The else is due to the first if returning false. The if part of else if checks its own parameters, which in this case is whether “a” is greater than “b.” If “a” is greater than “b,” then the program would have printed “a is greater than b.” However, since “a” is less than “b,” we go further down and simply call else, which runs if all the above parameters return false. It does not require its own parameter to be true. Note that each if statement can have an infinite number of else if statements following it, but only one else statement. Also note that you cannot have code between an if statement and an else statement, because the code would not be able to find the else. Also, an else statement requires there to be an if statement before it, since for an else statement to be called, an if statement HAS to equal false immediately before it. It is also important to know that it is possible to have an if statement INSIDE of another if statement. This is called a NESTED if statement

Example:

```
int a = 20;
int b = 30;
if (a == 20) {
    if (a < b) {
        System.out.println("a is less than b and it is equal to 20");
    }
}
```

In this example, we first check if the value of “a” is equal to the value of “b.” If it is, then we check to see if the value of “a” is less than the value of “b.” If this statement also returns true, we can print to the console “a is less than b and it is equal to 20.” This can work with as many if statements as you like. This can be useful if it is completely necessary to check the parameters separately, or if you want to add additional logic between each check. You will fully understand the way an if statement works by the end of this section.

Relational Operators

Java supports 6 different types of relational operators, which can be used to compare the value of variables. They are the following:

- ==

This operator checks to whether the value of two integers are equal to each other. If they are equal, the operator returns true. If they are not, it returns false.

Example:

```
int a = 10;
int b = 10;
if (a == b) {
    System.out.println("they are the same");
}
```

This code checks to see IF the value of “a” is equal to the value of “b.” Since the value of “a” is equal to the value of “b,” the value within the brackets will be true. This ends up causing the above code to print out the statement “they are the same.”

- !=

This operator checks whether the value of two things ARE NOT equal. If two things DO NOT equal the same thing, then it will return true. If they DO equal the same thing, it will return false.

Example:

```
int a = 10;
int b = 20;
if (a != b) {
    System.out.println("they are not the same");
}
```

This code will check if “a” does not equal “b.” Since “a” does not equal “b,” the program will print “they are not the same” in the console.

- >

This operator checks if something is greater than something else. If the value of the variable in front of it is greater than the value of the variable after it, it will return true. If the value of the variable in front of it is less than the value of the variable after it, it will return false.

Example:

```
int a = 10;
int b = 20;
if (a > b) {
    System.out.println("a is larger than b");
}
else {
    System.out.println("a is not larger than b");
}
```

This example checks to see if “a” is larger than “b.” Since “a” is not larger than “b,” this if statement will return false. Instead, it will print “a is not larger than b,” because the code passes the failed if statement and thus calls the else statement.

- <

This operator checks to see if something is less than something else. If the value of the variable before the operator is less than the value of the operator after the variable, then the code will return true; else, it will return false.

Example:

```
int a = 10;
int b = 20;

if (a < b) {
    System.out.println("a is larger than b");
}
```

```
else {  
System.out.println("a is not larger than b");  
}
```

This example is just like the one before it; however, the operator changed from greater than to less than. Therefore, the if statement will return true this time. Since the value of “a” is less than the value of “b,” the program will print to the console “a is larger than b.”

- **>=**

This operator checks whether the condition is greater than or equal to something else. If the value of the variable before it is greater than or equal to the variable after it, then it will return true. If the variable after it is greater but not equal to the variable before it, it will return false.

Example:

```
int a = 20;  
int b = 20;  
if (a >= b) {  
System.out.println("a is larger or equal to b");  
}  
else {  
System.out.println("a is not larger or equal to b");  
}
```

In this example, “a” and “b” both have a value of 20. Although “a” is not greater than “b,” it is EQUAL to “b,” therefore the statement returns true and the code prints “a is larger or equal to b.”

- **<=**

This operator checks to see if something is less than or equal to something else. If the value of the variable before it is less than or equal to the variable after it, then it will return true. If the variable after it is less but not equal to the variable before it, it will return false.

Example:

```
int a = 20;
int b = 20;
if (a <= b) {
    System.out.println("a is larger than b");
}
else {
    System.out.println("a is not larger than b");
}
```

This example is similar to previous one, but here we have changed the operator from `>=` to `<=`. Although the operator has changed, the result is the same. Since “a” is equal to “b,” this code will return true, printing “a is larger than b” to the console.

The Logical Operators

Java supports three Logical Operators that can be used in your logic, which will often be used in conjunction with Relational Operators.

- **&&**

This is known as the logical AND operator. If the operands before and after this operator both return true, then the condition returns true. If both of the operands are false, or one operand is false, the condition will return false. BOTH operands MUST be true for the condition to return true.

Example:

```
int a = 20;
int b = 20;
if (a == 20 && a == b) {
    System.out.println("a is equal to 20 and b ");
}
else {
```



```
System.out.println("a is not equal to 20 or a is not equal to b");  
}
```

In this example, we check to see if “a” is equal to 20 AND if “a” is equal to “b.” These two conditions will be referred to as operands. Since “a” is equal to 20 and equal to “b,” the statement returns true, and we print “a is equal to 20 and b” to the console.

Example 2:

```
int a = 20;  
int b = 30;  
if (a == 20 && a == b) {  
    System.out.println("a is equal to 20 and b");  
}  
else {  
    System.out.println("a is not equal to 20 or a is not equal to b");  
}
```

In this next example, we check the exact same thing as in the first example. The difference this time, however, is that we changed the value of “b” from 20 to 30. This means that when we run this code, the if statement will return false because although “a” is equal to 20, it is not equal to “b.” Therefore, we print out “a is not equal to 20 or a is not equal to b.” Take note that the order does not matter. If the first operand was false instead of the second, we would have the same result.

- ||

This Operator is known as the logical OR operator. If the first operand OR the second operand is true, the statement will return true. This means that if the first operand returns false and the second is true, the statement will still return true, and vice versa. The statement will also return true if both the operands are true. Essentially, when using an OR operator, at least one operand must return true.

Example:

```

int a = 20;
int b = 30;
if (a == 20 || a == b) {
    System.out.println("a is equal to 20 or b");
}
else {
    System.out.println("a is not larger than b");
}

```

In this example, we check to see if “a” is equal to 20, OR if “a” is equal to “b.” In this case, “a” is equal to 20 and is not equal to “b.” Since only one of these operands needs to be true, the statement as a whole will return as true and the program will print “a is equal to 20 or b” in the console.

Example 2:

```

int a = 20;
int b = 20;
if (a == 20 || a == b) {
    System.out.println("a is equal to 20 and b ");
}
else {
    System.out.println("a is not equal to 20 or b ");
}

```

In this example, I have changed the value of “b” in the previous example from 30 to 20. This means that “a” is equal to 20 and “a” is equal to “b.” Both operands return true, therefore the statement returns true. This is due to the fact that an OR operator requires one or more of the operators to be true. This program will output “a is equal to 20 and b” in to the console.

- !

This operator is known as the logical NOT operator. It is used to reverse the logical state of its operand. This operand can be used with any other operand to reverse its output. If an operand was returning true, the operand will return

false after applying the logical NOT operator.

Example:

```
int a = 21;
int b = 20;
if ( !(a == 20) && a > b) {
    System.out.println("a is not equal to 20, and a is greater than b");
}
else {
    System.out.println("a is equal to 20, and a is not greater than b ");
}
```

In this example, we check to see if the value of “a” does not equal 20, and we check if “a” is greater than “b.” Since “a” is both not equal to 20 and is greater than “b,” we print to the console “a is not equal to 20, and a is greater than b.”

Example 2:

```
int a = 21;
int b = 20;
if ( !(a == 20 && a > b) ) {
    System.out.println("a is not equal to 20, and a is greater than b ");
}
else {
    System.out.println("a is equal to 20, and a is not greater than b ");
}
```

In this example, the position of the bracket has changed so that the NOT operator extends to both operands. Now for the statement to be true, “a” has to NOT equal 20 AND “a” must NOT be greater than “b.” In this situation the statement will return false, because “a” is greater than “b.” Due to the operand, it needs to be less than “b” for the statement to return true. Therefore, this program will return “a is equal to 20, and a is not greater than b” to the console.

Combining Operators

In Java, it is possible to use as many operators as you require per statement.

Example:

```
int a = 21;
int b = 20;
if ( (a == 20 && a > b) || (a != 20 && a < b)) {
    System.out.println("correct");
}
else {
    System.out.println("wrong");
}
```

With this example, we see how we can achieve much more complicated statements. Here, we apply an OR operator with two operands; however, each operand also incorporates an AND operator. Therefore, for this statement to return true, “a” must be equal to 20 and must be greater than “b” OR “a” must not equal 20 AND “a” must be less than “b.” Note that we use brackets to make the code easier to understand, and to prevent the code from misreading our logic.

Example 2:

```
int a = 21;
int b = 20;
if ( a == 20 && (a > b || a != 20) && a < b) {
    System.out.println("correct");
}
else {
    System.out.println("wrong");
}
```

In the above example, we have significantly changed the logic of the program by merely switching the position of a bracket. Now you may understand that the brackets surround the two inner operands, instead of two pairs of brackets

surrounding each pair of outer operands. For this statement to now return true, three different conditions must return true: Variable “a” must be equal to 20, AND either “a” must be greater than “b” OR it must not be equal to 20, AND “a” must be less than “b.” With this new logic, the first operand must be true in addition to EITHER the second or third having to return true. Also on top of that, “a” must be less than “b.” This statement will return false because it fails the first condition: “a” does not equal 20. It will pass the second condition because it is greater than “b,” and it will fail the last condition because “a” is not less than “b.” Since this would have required three correct conditions but only had one, the statement returns false and prints “wrong.”

Assignment

Now we are going to go through another assignment to make sure we understand everything about operators. We have three friends who want to know how much money each has compared to the others. We will make a program that will output who is the richest among them. We will have three integers named bob, john, and tom. We will give each one of them a different value, and the program must output who is richer. It must also tell us if someone has the same amount of money as someone else.

For example:

If bob = 20, tom = 10, and john = 5, the output must be:
“bob is richer than tom, who is richer than john”

But if bob = 20, tom = 20, and john = 5, the output must be:
“bob is just as rich as tom, both are richer than john”

The program needs to work for each possible outcome. I highly encourage you to try this program by yourself and struggle through it as best you can before you look at the answer. A big part of programming is problem-solving and understanding code.

Answer and Explanation

Note that the program will be explained through comments in the code.

```
// this method calculates who is richer than whom
public static void main(String[] args) {
// feel free to change these values around
int tom = 10; //amount of money tom has
int bob = 20; // amount of money bob has
int john = 10; // amount of money john has
/* We first check if everyone has the same amount of money
 * note how I never check if tom is equal to john, because if
 * tom is equal to bob and bob is equal to john, then obviously
 * tom is equal to john
 */
if (tom == bob && bob == john) {
System.out.println("Everyone has the same amount of money");
}
/*
 * If the first statement returns false
 * I next check the unique case where tom and bob have the same amount
 * but john does not
 */
else if (tom == bob && bob != john) {
/*
 * I now check if bob is greater or less than john
 * and then output the correct answer
 */
if (bob > john) {
System.out.println("tom is just as rich as bob, both are richer than john");
}
else if (bob < john) {
System.out.println("tom is just as rich as bob, both are more poor than john");
}
}
/*
 * I repeat the same process as the previous statement but with different
 people
```

```
*/
```

```
else if (tom == john && john != bob) {  
if (john > bob) {  
System.out.println("tom is just as rich as john, both are richer than bob");  
}  
else if (john < bob) {  
System.out.println("tom is just as rich as john, both are more poor than  
bob");  
}  
}  
}
```

```
/*
```

```
* The last possible combination of names,
```

```
* same check as previous statement
```

```
*/
```

```
else if (john == bob && bob != tom) {  
if (bob > tom) {  
System.out.println("bob is just as rich as john, both are richer than tom");  
}  
else if (bob < tom) {  
System.out.println("bob is just as rich as john, both are more poor than  
tom");  
}  
}  
}
```

```
/*
```

```
* Now I check the last possible combinations
```

```
* where each person has different amounts of money
```

```
* the next 6 statements cover each possible outcome
```

```
*/
```

```
else if (tom > bob && bob > john) {  
System.out.println("tom is richest, followed by bob, followed by john");  
}  
else if (tom > john && john > bob) {  
System.out.println("tom is richest, followed by john, followed by bob");  
}  
else if (bob > tom && tom > john) {
```

```
System.out.println("bob is richest, followed by tom, followed by john");  
}  
else if (bob > john && john > tom) {  
System.out.println("bob is richest, followed by john, followed by tom");  
}  
else if (john > bob && bob > tom) {  
System.out.println("john is richest, followed by bob, followed by tom");  
}  
else if (john > tom && tom > bob) {  
System.out.println("john is richest, followed by tom, followed by bob");  
}  
}
```

When writing code, it is important to comment on all of your logic so that someone who has not written it can easily understand and edit it. I recommend that you leave as many comments as you feel you need. I also encourage you to try to find a better way to solve this problem. Maybe there is a way you can write this in a quarter of the length; with programming, there is never only one answer.

Chapter 9

Loops and Arrays

Loops

When programming, you might run into a situation where you will need to loop through a large amount of numbers. If we used what we learned so far to loop through one hundred numbers, we would need one hundred “if” statements. I think you would agree that would get really messy and frustrating. Luckily, Java supports three types of loops, all of which loop through things in slightly different ways. Below is a quick explanation of each.

While Loop

A while loop is a control structure that will allow you to repeat a task as many times as you program it to. The syntax for a while loop is as follows:

```
while (expression) {  
    // insert code  
}
```

This works in a similar way that if statements work, except WHILE the expression is true, the code within the while loop will run until the expression becomes false (if it ever does).

Example:

```
int x = 10;  
while (x > 0) {  
    System.out.println(x);  
    x--;  
}
```

The preceding code will print out the value of “x”, and then continuously subtract one from it until the value of “x” is no longer greater than zero. If you were to run this code, you would get an output of every number from one

to ten. Once the value of “x” reaches zero, it no longer allows the expression to return true, which is when the while loop finishes. Note that you should watch out for infinite loops. These are errors in your code that cause the loop to never end, essentially freezing the program. This would happen when there is an error in your logic that would prevent the statement from ever becoming false.

Do While Loop

A do while loop works in a very similar way as the while loop except for one major difference: The “do”—what happens when the while statement is true—is called before the condition is asked. What that means is that a do while loop will always run at least once, because it does the task before it checks the condition. Execution of the second loop completely depends on the condition.

Example:

```
int x = 10;
do{
System.out.println(x);
x--;
}
while(x > 0);
```

In this example, the last program logic from the while loop has been directly transferred to the do while loop. As you can see, the only difference is that the logic that runs while the statement is true is *above* the while loop instead of below it. For this specific example, the output of the code is exactly the same. However, if we change the code to the following:

```
int x = 0;
do {
System.out.println(x);
x--;
}
while(x > 0);
```

The output would be different than a while loop. If this was a while loop, the code within the loop would never run. However, in a do while loop, the do happens before the while, so the code will always run at least once. During the first run, it never technically gets checked for a true statement until the block of code is processed first. This can be helpful in specific situations, but it is not used as often as a normal while loop.

For Loops

For loops allow you an easy way to loop or increment through a specific range of values. The syntax for a for loop is as follows:

```
for (initialization; termination; increment) {  
  
//statement  
  
}
```

Initialization: Initializes a counter variable.

Termination: States the expression to evaluate for true.

Increment: Increments the initialized counter variable.

Example:

```
for (int x = 0; x < 10; x++) {  
  
System.out.println(x);  
  
}
```

In this example, we start by initializing the for loop. We create a new int named “x” and set it to zero. Then we state when it will terminate. We say that the loop will continue for as long as “x” is less than ten. Then we set the increment to increase the value of “x” by one. The initialization happens only

once, when the for loop is first called. After the loop is initialized, it should run at least once (unless you initialized it outside its bounds). Once per iteration, when it reaches the end of the code, it will call the increment value. After it increments, it will check whether it should be terminated; if it does not get terminated, the content is called again and the process repeats until termination. For the example code above, the output will be every number from zero to nine.

Arrays

Arrays are data structures that store a fixed number of variables of the same type. Instead of having to create one hundred different integer variables to store one hundred different number values, you can create one integer array that holds one hundred different numeric values. This makes it a lot easier to manage the integers, as well as keeping your code a lot more organized and simple.

Declaring Arrays

Declaring an array is similar to declaring a regular variable. The syntax is as follows:

```
dataType arrayName[];
```

This is way is way to declare an array that was added to Java to accommodate programmers coming from a C/C++ background.

The array above currently holds no data. We declared the array, but we have not declared the size of it, nor the values in each index of the array:

Example:

```
int intArray[] = new int[5];
```

The example above declares an integer array with five available slots. This means that this intArray can hold five different integer numbers within it. Each of these numbers can be independently edited and accessed in the following

way:

```
int intArray[] = new int[5];  
intArray[0] = 2;  
intArray[1] = 3;  
// ...  
System.out.println(intArray[0]);
```

The code above declares the `intArray` with five data slots. We can then access those slots by adding an `int` value inside the square brackets beside the array name. When we do “`intArray[0]`”, we are referencing the value at index zero of the `int` array. The moment that we declared the array with five values, we also created five slots numbered 0-4. When it was first declared, they contained the value zero. We then can access each of them independently, just like any normal variable. We can even read the value of each index just like a normal variable.

Multidimensional Arrays

Arrays can also be made much more complex by adding multiple dimensions to them. In the above examples, the arrays consisted of only one dimension. By adding more dimensions, we are essentially adding arrays within the arrays. It may sound complex, but it’s relatively easy to understand.

Example:

```
int intArray[][] = new int[5][5];
```

The following array consists of two dimensions. The first 5 means that we declare the array with five index values in it. The second array means that for each of those five index values, we can access another five index values within them. As an example, for the index of (`intArray[0]`) we added another five indices that we can access. If we think of index zero as just a regular integer, then we can essentially think of `intArray [0]` as being equal to `intSecondArray[5]`. This is because its array at zero contains five indices. Therefore, since each index of `intArray` can be thought of as an independent array, we have five arrays, all containing five indices, giving us a total of

twenty-five indices.

We can then access each of these indices in the following way:

```
int[][] intArray = new int[5][5];
intArray[0][0] = 1;
intArray[0][1] = 1;
intArray[0][2] = 1;
intArray[0][3] = 1;
intArray[0][4] = 1;
// and so forth
```

The following code demonstrates how to assign each value within `intArray[0]`. If you think of it as the first number, `intArray[0]` represents which inner array we want to access, which would leave the second number `intArray[][0]` as the actual value of each inner array. That might be an easier way we can make sense of it all.

Combining Loops and Arrays

After reading about loops and arrays, you might have noticed the possible potential to combine them to get greater use and production out of your program. Loops allow us to very quickly assign or retrieve values for large ranges, while arrays allow us to create really large ranges of numbers. How convenient would it be to combine the two of them? Below are a few examples of each type of loop, how to quickly loop through an array, and how to easily edit a multidimensional array.

Example 1:

```
int intArray[] = new int[100];
int x = 0;
while (x < 100) {
    intArray[x] = x;
    System.out.println(intArray[x]);
    x++;
}
```

In this example, we simply declare an array with one hundred index values. Then we declare an int that we need to increment. We tell the while loop to continue looping while “x” is less than the length of intArray (which is the number of indices it has); then we increment “x” by one after each iteration. This code quickly and easily edits one hundred different values and prints them to console. All of that in just seven easy lines!

Example 2:

```
int intArray[] = new int[100];
int x = 0;
do
{
intArray[x] = x;
System.out.println(intArray[x]);
x++;
}
while(x < 100);
```

This example is just like the first example, except we applied the same concept to a do while instead of a while. The output is exactly the same; however, you should be wary of do while loops because they always run at least once.

Example 3:

```
int intArray[] = new int[100];
for (int x = 0; x < 100; x++) {
intArray[x] = x;
System.out.println(x);
}
```

This method is probably my favorite way to do it. The advantage of this is that the counter variable is built inside the loop, so you don’t need to declare another variable or manually increment it. In my opinion, it is just a lot more convenient and organized.

Example 4:

```
int[][] intArray = new int[100][100];
for(int x = 0; x < 100; x++){
    for(int y = 0; y < 100; y++){
        intArray[x][y] = y;
        System.out.println(intArray[x][y]);
    }
}
```

This example demonstrates the most practical way to loop through a multidimensional array. It may look intimidating, but it is actually quite simple. We have two for loops inside of each other; each of them will loop through a specific dimension of the array. For this example, the first time it loops, it will start with $x = 0$. It will then loop through every single array index where the first index is zero. Once all indexes of $[0][y]$ get called, it increments “x” by one. It will then repeat the process until every single index has been called.

Assignment

Loops are very useful for editing a large number of values. One thing that can be greatly simplified thanks to loops is image editing. An image can consist of hundreds, maybe even thousands of pixels, each with its own color value. With the power of loops, we can quickly edit those color values. For this assignment, we are going to invert the colors of an image of your choosing. There are some parts to this program that you have not learned in this book yet, so I will try to help you out with that. To edit an image, we first need to create a file object and a buffered image object. To be able to use these, we need to import them. First, add the following lines of code to the top of your class:

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


After that, we will need an image to edit. Get any image of your choosing and add it to your package by dragging it into the main package folder of your project. Make sure you don't accidentally add it into the src folder or any other folders. Next you are going to start your code with the following two lines:

```
File file = new File("image1.jpg");  
BufferedImage image = ImageIO.read (file);
```

You will learn more about file handling later on. Also, be sure to rename "image1.jpg" to the name and file type of the image you added. Adding this code will cause an error, so for now, add a "throws exception" to your class. What this means is beyond the scope of this looping lesson, but the method should look something like this:

```
public static void main(String[] args) throws IOException {
```

If done correctly, the error should have disappeared. The only hint I will give to complete this exercise is that you can use `image.setRGB` and `image.getRGB` to set RGB values and also to edit them. You can also use `image.getWidth` and `image.getHeight` to get pixel dimensions. After you edit the color values, call:

```
ImageIO.write (image, "png", file);
```

This will save the new edited image and replace the old one (replace "png" with the file extension). Check your local folder to see the edited image.

Answer and Explanation

The following code will complete the task that I explained above; there are comments throughout to explain it. As always, there is more than one way to do this, so if you got it working with a different method, you are still correct as long as it works.

```
// this method inverts the colors of an image  
public static void main(String[] args) throws IOException {
```

```

/*
 * Don't worry about the two lines below, they are beyond the
 * looping/array section
 */
File file = new File("image1.jpg"); // sets file to the given file
BufferedImage image = ImageIO.read (file); // sets the image to the file
/*
 * We loop through the X and Y components of the image
 * by getting the amount of pixels in both the x and y coordinates of the image
 */
for(int x = 0; x < image.getWidth(); x ++){
for(int y = 0; y < image.getHeight(); y++){
/*
 * Sets the RGB value at each x and y coordinate of each pixel
 * to the inverse of the current color
 */
image.setRGB(x, y, 255 - image.getRGB(x, y));
}
}
/*
 * Writes the new image to overwrite the old image
 * Prints done to let us know the program has finished
 */
ImageIO.write (image, "png", file);
System.out.println("done");
}

```

This assignment is a lot more advanced than the previous ones, but it really helps to demonstrate the practicality of loops as well as the amount of freedom and power they provide.

Chapter 10

Methods

A method is a collection of code that is grouped together to create a specific function or task. Basically, methods help us to develop blocks of code that will perform any task when you call a method. If you use methods to perform a task, you can reuse them whenever you want. This section will teach you how to create and apply methods to their fullest extent.

Creating a Method

The syntax to create a method is as follows:

```
modifier returnType methodName(parameters parameters){  
//code  
}
```

modifier: Defines what has access to the method, and other properties.

returnType: The dataType that the method will return. Can return nothing if set to void.

methodName: The name of the method. Same syntax as variables except it starts with a capital letter.

Types of Methods

We have two types of methods in Java: Static and Non-Static. Static methods start with the keyword “static.” You don’t need any class objects to call this type of method; however, all non-static methods need a class object to invoke.

Syntax of Static Method

```
static return-type method_name (parameters)  
{  
...  
...}
```

```
}
```

Syntax of Non-Static Methods

```
return_type method_name (parameters)
{
...
...
}
```

Parameters

Parameters are variables that you can give to the method to use when you call it. A method can take zero parameters as well.

Example 1:

```
public static void main(String[] args){
int x = 0;
x = SampleMethod (x);
}
public static int SampleMethod (int x) {
return x + 1;
}
```

Here we have created a method of type int called *SampleMethod* that takes one parameter of the int data type. We also state that the method is public, which means that this method can be accessed from subclasses. We have set it to static, which means only one instance of this method can ever run at once. In the main method, we declare an int “x.” Then we set the value of “x” to *SampleMethod(x)* . Since sample method has a return type of int, whenever the method is called, it will ALWAYS return an int value. We gave it the value of “x,” and the method adds one to that value and returns it. Therefore, “x” will now have a value of 1.

Example 2:

```

public static void main (String[] args) {
double a = 3;
double b = 6;
Average (a,b);
}
public static void Average(double x, double y){
System.out.println("the average is " + (x+y)/2);
}

```

The example method above has a data type of void. This means that the method will not return any value. This is good if you just need the method to perform a specific function and don't need to assign a value to it. In the above example, we have created a method that calculates the average of two numbers. The method simply performs the function of calculating an average and prints it to the console. It will never return any values.

Example 3:

```

public static int a = 1;
public static int b = 2;
public static void main(String[] args) throws IOException {
Sum ();
}
public static void Sum(){
System.out.println("sum is" + ( a + b ));
}

```

In this example, we create a method called Sum() that takes no parameters. When a method takes no parameters, it is called with empty brackets, but the brackets must still be there no matter what. This also shows how you can use public variables from the class within any method in that class. The variable does not have to be an argument in the parameters for it to be useable. We have set the integers “a” and “b” to static. This is because only static variables can be used in a static method.

Method Overloading

Method overloading means that your program contains more than one method with the same name but with different parameters and return types.

Example:

```
public static void main (String args[]) throws IOException
{
    int a = 1;
    int b = 2;
    double c = 3;
    double d = 4;
    System.out.println( Sum (a,b));
    System.out.println( Sum (c,d));
}
public static int Sum(int x , int y){
    return x+y;
}
public static int Sum(double x , double y)
{
    return (int) (x+y);
}
```

In the example above, we have created two methods with the same name that take two different types of arguments. Unlike variables, you can name two methods the same, because Java can differentiate those methods based on their parameters and return types. If you run the above code, you will see that both methods work. One takes the sum of the double values, while the other processes the int values.

Assignment

Create a more advanced calculator. Have three values. The first tells you which kind of operation to perform (for example if a == 1, add the values; if a == 2, subtract them). The next two values will be used for manipulation. Make the calculator support addition, subtraction, division, and multiplication, and give each its own method.

```

// The main method
public static void main(String args[])
{
// Integer that chooses the action
int action = 4;
// The variables to manipulate
double x = 4.5;
double y = 3;
/*
* Based on the value of action we decide what to do with
* the two variables. Each action is performed in its own method
* if action is not valid, we print invalid operation, as we have
* nothing to do for those values.
*/
if (action == 1) {
System.out.println("The sum is " + Addition (x,y));
}
else if (action == 2) {
System.out.println("The difference is " + Subtraction (x,y));
}
else if (action == 3) {
System.out.println("The product is " + Multiply (x,y));
}
else if (action == 4) {
System.out.println("The answer is " + Divide (x,y));
}
else {
System.out.println("Invalid operation");
}
}
// Addition method
public static double Addition (double a, double b) {
return (a+b); // Returns the sum of a and b
}
// Subtraction method
public static double Subtraction (double a, double b) {
return (a-b); // Returns the subtraction of a and b
}

```

```
}  
// Multiply method  
public static double Multiply (double a, double b) {  
return (a*b); // Returns the product of a and b  
}  
// Divide method  
public static double Divide (double a, double b) {  
return (a/b); // Divide the a by b  
}
```

At this point, the above code should be pretty self-explanatory. You should attempt to spice things up a little. Maybe add an array in there to play with a few hundred values instead of two. Add some more functions and see how complex you can make this calculator. You have all the knowledge you need to make the best calculator possible.

Chapter 11

Inheritance and Polymorphism

Let's redefine classes and introduce objects. Imagine your average suburban neighborhood. If you look at a single street, you'll see a variety of houses. In fact, if you look at any two houses in a given neighborhood, they most likely differ in color, model, and other variables. But if you look at all the houses, you may start noticing repetitions; specifically, the general structure of the houses may be rather similar. This is because, rather than having architects design and test hundreds of houses, they just make a "few" houses (this could actually be a significant number), and then just use different colors and orientations to distract people from the fact that they are essentially the same. Classes and objects work in a similar way. Classes are models or structures of how an object must be laid out. Objects are like the differences in the houses: their color, their orientation, etc. In coding, objects can each have unique values assigned to variables, but must always contain the variables defined in the class.

Classes are defined by the class keyword, followed by an identifier called a class name, and then followed by a block ({ }). This is the bare minimum. Here is an example of a class called Fruit.

```
class Fruit
{

}
```

Extremely simple. In this class, we can define variables for use within the class (or outside of the class if the variable is public or protected). For now, simply put public in front of your data types until we explain access modifiers.

```
class Fruit
{
    public String name;
    public int timeSincePicked;
}
```

Now let's create an object. First we need our main method and a class. We will put our main method inside the class. It does not matter where the main method is, as long as it is written out properly. To instantiate an object (which is essentially "building the house" in our neighborhood analogy), we simply write the new keyword, followed by the class name of the object you wish to instantiate, followed by parentheses (like a method, which will be explained in a bit) and a semicolon. So like this:

```
class Fruit
{
public String name;
public int timeSincePicked;
public static void main(String[] args)
{
new Fruit();
}
}
```

This on its own is not very useful. We create our object, but we cannot use it before we assign it to a variable. Classes can act as variable data types, and objects are their values.

```
class Fruit
{
public String name;
public int timeSincePicked;
public static void main(String[] args)
{
Fruit objectOfFruit = new Fruit();
}
}
```

Now we can access the variables inside of our objectOfFruit object. To access these variables, we simply write the object name, a period (.), and the variable name. Methods can also be put inside classes and can access variables in their own object. Remember that variables in objects can have

different values. Be careful where you write code, though, as you can only have code inside methods. Classes can only contain variables, methods, and other classes directly.

```
class Fruit
{
    public String name;
    public int timeSincePicked;
    public static void main(String[] args)
    {
        Fruit newPear = new Fruit();
        newPear.name = "Pear";
        newPear.timeSincePicked = 24894;
        newPear.checkIfRotten(); // Outputs: "This Pear is not rotten."
```

```

        Fruit oldPear = new Fruit();
        oldPear.name = "Pear";
        oldPear.timeSincePicked = 3659246;
        oldPear.checkIfRotten(); // Outputs: "This Pear is rotten!"
    }
    public void checkIfRotten()
    {
        if(timeSincePicked > 100000)
        {
            System.out.println("This " + name + " is rotten!");
        }
        else
        {
            System.out.println("This " + name + " is not rotten.");
        }
    }
}
```

Classes can take on the properties of another class and build on top of them. This is called inheritance. Classes that “extend” another class will have all the variables and methods of the extended class. The class that is extending is called the child, and the class that it is being extended to is the parent. To

extend, after the class name type extends and the name of the parent class.

```
class Food
{
public String name;
}

// Fruit is the child and Food is the parent
class Fruit extends Food
{
public boolean isRotten;
// Has access to both name and isRotten
}
```

This leads to polymorphism. This means that a child class can be the value for an object of the parent class. This is called upcasting. Here is some code to clarify that explanation:

```
class Food {
public String name;
public static void main(String[] args)
{
Food pear = new Fruit();
pear.name = "pear";
// pear.isRotten = true; // This will not work because pear is technically of the
type Food which does not have an isRotten variable
}
}

class Fruit extends Food {
public boolean isRotten;
// Has access to both name and isRotten
}
```

Casting or downcasting can convert an object to another type in order to access variables of that type. This has specific rules in that an object cannot be cast into a type that it is not. For example, if Fruit and Meat both extend

from Food, and Pear extends from Fruit, and you have an object of type Fruit that has been upcasted to type Food, it cannot be cast into type Meat. This is due to the fact that the original object was not of that type. Similarly, the object cannot be cast into type Pear, because it was not of that type before. Casting is done by simply putting the class you wish to cast to in brackets before the object you wish to cast.

```
class Food{ }
class Fruit extends Food{ public int variable; }
class Meat extends Food{ }
class Pear extends Fruit{ }
class Main
{
public static void main(String[] args)
{
Food genericFood = new Fruit();
Fruit fruit = (Fruit)genericFood;
fruit.variable = 1; // Able to access variable of casted object
// Meat meatIsntFruit = (Meat)genericFood; Doesn't work because fruit is not
a meat
// Pear fruitIsntPear = (Pear)genericFood; Doesn't work because fruit is not
specific enough to be a pear
}
}
```

Access modifiers pretty much explain themselves: they modify what can access them. Access modifiers can be put on variables, methods, and classes. They always go before the data type of a variable, the return type of a method, or the class keyword. There are three access modifiers:

- Public: Anything can access it.
- Protected: Only the class itself and its children can access it.
- Private: Only the class itself can access it.

If you omit an access modifier, it will default to no modifier, which is more

or less similar to public. Access modifiers essentially prevent other programmers from messing up processes within your class. There are very few requirements for placing access modifiers—you could make pretty much everything public and your code would run fine. They are simply to prevent errors by people who do not know what they are doing when interacting with your code (or you, if you forget how your code works).

One common thing programmers use the private access modifier for is to make read-only variables (outside the class, at least). This is done to make methods that simply return the value of the variables. This works because outside classes can access the methods, and the method—since it is inside the class—can access the variables. This allows outside classes to access the value, but since they cannot access the variable, they cannot modify the value. Be careful with this system though, because objects are “references,” meaning that changing values within them will change the original object, no matter where it is in your code.

```
class Something
{
private int aNumber;
private AnotherThing anObject;
public Something(int aNumber, AnotherThing anObject)
{
this.aNumber = aNumber;
this.anObject = anObject;
}

public int getNumber()
{
return aNumber;
}

public AnotherThing getObject()
{
return anObject;
}
```

```

public static void main(String[] args)
{
    AnotherThing thing = new AnotherThing(4);
    Something something = new Something(3, thing);
    // something.aNumber = 6; // aNumber cannot be accessed here
    System.out.println(something.getNumber()); // "3"
    something.getObject().number = 7;
    System.out.println(thing.number); // "7"
}
}

```

```

class AnotherThing
{
    public int number;
    public AnotherThing(int number)
    {
        this.number = number;
    }
}

```

Some rules do apply to access modifiers. Specifically, when creating classes, they cannot have the access modifier `public` unless they are either inside another class or in a file of that name.

When referencing classes contained within classes, simply type the container class, a period (`.`), and the contained class. Inner classes are generally used when you want a class that will not be used much outside of the container class.

```

//This is in a file called Food.java
public class Food
{
    public static void main(String[] args)
    {
        // example of classes inside classes.
        Food.Core core = new Food.Core();
    }
}

```

```
public class Core{}  
}
```

```
//public class DoesntWork{}  
class DoesWork{}
```

Children can also “override” a method from their parents by duplicating the method declaration in its own class. This allows different outcomes depending on the type of class, but you will simply need to call the method from an object that has been upcasted to the parent type.

```
class Food  
{  
    public void saySomething()  
    {  
        System.out.println(“What is this food, ma’am?”);  
    }  
    public static void main(String[] args)  
    {  
        Food food = new Food();  
        Food fruit = new Fruit();  
        food.saySomething(); // “What is this food, ma’am?”  
        fruit.saySomething(); // “What a lovely fruit!”  
    }  
}
```

```
class Fruit extends Food  
{  
    public void saySomething() // Exact same declaration as its parent  
    {  
        System.out.println(“What a lovely fruit!”);  
    }  
}
```

Constructors are special methods. They can only be called when creating an object and can only be run once. They do not have a return type (as they technically return the object being created), and they can have parameters. Constructors must always have the same name as the class they are in.

Constructors can be used to set the initial values of an object.

```
class Fruit
{
private String name;
public Fruit(String nameParam)
{
name = nameParam; // We can set the private variable of the fruit being
instantiated because we are within the class.
}
public void saySomething()
{
System.out.println("This is a " + name);
}
public static void main(String[] args)
{
Fruit f = new Fruit("pear");
f.saySomething(); // "This is a pear"
}
}
```

Constructors can also be overloaded, meaning you can have multiples of the same method as long as the parameter types have a different order or number.

```
class Fruit
{
public String name;
public Fruit()
{
name = "Generic fruit";
}
public Fruit(String nameParam)
{
name = nameParam;
}
public static void main(String[] args)
{
```

```
Fruit f1 = new Fruit();//name is "Generic fruit"  
Fruit f2 = new Fruit("pear");//name is "pear"  
}  
}
```

The `this` keyword is especially useful in the case of constructors. It allows programmers to refer to the variables of a class while ignoring the local variables in a given method. This makes it possible to give the parameters of a constructor the same name as the variables in your class. When using this, simply treat it as any object.

```
public class Example  
{  
    public int aNumber;  
  
    public Example(int aNumber)  
    {  
        this.aNumber = aNumber;  
        //this.aNumber refers to the variable in the class, while aNumber refers to the  
        //constructor's parameter  
    }  
}
```

There is a special type of class called an “anonymous inner class.” This means it has no name, hence “anonymous.” It also means that it can only exist once (unless you copy the code twice). The uses for anonymous inner classes are limited, but if you are ever in a situation where you only need a specific class once, you can create one rather than having to create a file and make it a class. Anonymous inner classes are cemented in inheritance, most commonly used when dealing with interfaces and abstract classes, but we’ll get more into that soon. Anonymous inner classes are similar to the instantiation of objects. Begin by typing “new” followed by the name of the class you wish your anonymous inner class to inherit from, followed by your parameters and then a block. This can then override or create methods from the parent class and can create variables. Finally, end it with a semicolon. Be careful, though, as the block cannot create constructors and cannot create methods for use outside of the anonymous inner class.

```

class Food
{
public void eat()
{
System.out.println("Mmmmm");
}

public static void main(String[] args)
{
Food specialFood = new Food(){
public void eat()
{
System.out.println("Yuck! This food is rotten!");
}
};
specialFood.eat();// "Yuck! This food is rotten!"
}
}

```

A common use for anonymous inner classes is dealing with input listeners. Rather than creating classes for each type of listener—key listener, mouse listener, mouse wheel listener, etc.—it may be easier to make anonymous inner classes. This heavily depends on what you are using it for.

Chapter 12

Interfaces and Abstract Classes in Java

There are also some special types of classes, specifically, interfaces and abstract classes.

Abstract classes are classes that cannot be instantiated—you cannot make an object of an abstract class. You can, however, make objects of children of the abstract class. This is useful when you want a group of classes to have a common set of variables or methods, but do not want to have an unspecified object. Abstract classes require the use of upcasting and downcasting. Abstract classes are defined by the `abstract` keyword before the `class` keyword. On top of this, abstract classes can contain abstract methods. This means that you can call a method in an abstract type and it will run the method of the child class. Abstract methods must always be contained inside abstract classes, and are defined by placing the `abstract` keyword before the return type of the method. Abstract methods also do not have a body; they simply end in a semicolon after their parameters. When a child inherits from the parent abstract class, it must override or define abstract methods. Abstract classes can also contain normal methods in order to add common functionality, as in normal classes.

```
abstract class Food
{
    public abstract void saySomething();
    public static void main (String args[])
    {
        Food a = new Fruit();
        Food b = new Meat();
        a.saySomething();//"This fruit is delicious!"
        b.saySomething();//"This meat is delicious!"
    }
}
```

```
class Fruit extends Food
{
    public void saySomething()
```

```

{
System.out.println("This fruit is delicious!");
}
}

```

```

class Meat extends Food
{
public void saySomething()
{
System.out.println("This meat is delicious!");
}
}

```

Interfaces are similar to abstract classes in that they can provide methods that must be overridden and they cannot be objects. Interfaces cannot have variables in them. Interfaces cannot be “extended” to, but are instead “implemented” to. The difference is that while extending can only be done onto one class—meaning you can only have one parent—implementing can be done onto several interfaces, not classes. Interfaces are used when you need a number of unrelated classes to share common methods. To create an interface, simply create a class, but replace the class keyword with the interface keyword. Inside, you may only have method declarations—the return type, method name, and parameters—followed by a semicolon. This is similar to abstract methods, except you do not use the abstract keyword. To implement, simply put the implements keyword after either the class name or (if there is a parent class) after the parent class name.

```

interface Eatable
{
public void eatObject();
public static void main(String[] args)
{
Food fruit = new Fruit();
Eatable stapler = new Stapler();
fruit.eatObject(); // “Mmmm...juicy”
stapler.eatObject(); // How do you think eating a stapler will end?
}
}

```

```
}
```

```
abstract class Food implements Eatable{}
```

```
class Fruit extends Food
```

```
{
```

```
public void eatObject()
```

```
{
```

```
System.out.println("Mmmmm...juicy");
```

```
}
```

```
}
```

```
class Stapler implements Eatable
```

```
{
```

```
public void eatObject()
```

```
{
```

```
System.out.println("AAAAHHHH! THIS WAS A TERRIBLE IDEA! CALL  
AN AMBULANCE! WHY DID I EAT A STAPLER?");
```

```
}
```

```
}
```

Notice that since the Food class is abstract, you do not need to override Eatable's method, unless you want all foods to have a default action for this method. Be aware that since Food implements Eatable, you can call the eatObject method from the type Food and it will run the Fruit eatObject (in this case).

Chapter 13

Packages

Packages are essentially ways to organize your code, specifically for use in other projects. When you import things, you are using packages. `java.util` is a package, just as `java.io` is a package. When you type the asterisk (*) after, you say to import all the files in that package. To create a package, it is best to create one in your IDE of choice, as it will automatically setup everything for you. To define which package a class is in (which the package it is physically in must match), type package before the class you are defining and then the package path. As said before, having your IDE deal with this reduces headaches. Technically, packages are not required, but using packages for your code is good practice and helps to reduce clutter for you or other programmers in the future. Some programmers even use packages to separate different parts of their code for themselves.

Types of Packages

There are two types of packages available in Java programming:

- Pre-Defined Packages
- User-Defined Packages

Pre-Defined Packages

Pre-defined packages are already defined; they contain collections of pre-defined classes that help us to develop well-structured applications. There are several pre-defined packages in Java. In the following section, you will notice some pre-defined packages and their classes.

`Java.io`
`Java.lang`
`Java.awt`
`Java.applet`
`Java.util`
`Java.sql`

Java.net

- **Java.io**

This package contains Input and Output oriented classes, such as `DataInputStream`, `BufferedReader`, `FileReader`, and so on. If you want to use these classes in your program, you will need to import this package. By default, this package is automatically imported to your program.

```
import java.io.*; // Importing format of Java.io package in your program.
```

Important classes in IO Package

1. `FileInputStream`
2. `SequenceInputStream`
3. `PipeInputStream`
4. `ByteArrayInputStream`
5. `ObjectInputStream`
6. `StringBufferInputStream`
7. `FilterInputStream`
8. `BufferedInputStream`
9. `PushbackInputStream`
10. `DataInputStream` Class
11. `DataInput` Interface
12. `Java.lang`

This package contains Primitive data type classes, Exception classes, Wrapper classes, and so on. This class method helps us to convert strings to other formats that handle exceptions. In the following section, you will see important classes inside the `Java.lang` package.

- **Java.lang**

```
Import java.lang.*; // importing style of the java.lang package.
```

Classes in `Java.lang` package

1. Exception
2. Throwable
3. String
4. Integer
5. Float
6. Double
7. Byte

- **Java.util**

This package contains some important classes such as Random, Date, and so on.

- **Java.applet**

This class contains the Applet class, which is used to create the GUI Application using the Applet. It is nothing but a container, just like Frame, Panel, and Dialog.

- **Java.awt**

AWT stands for Abstract Window Toolkit. It contains Event Handling and Component classes like Label, Button, Choice, Checkbox and Scrollbar, ActionEvent, and so on. It will be discussed in depth in a later part.

- **Java.net**

This package contains network programming classes. This package's classes help developers to develop Network-based applications.

- **Java.sql**

This package contains Database Handling classes. You can retrieve Database records by using some special classes, which will be discussed in depth in a later section.

User Defined Packages

These are developed by users or developers for their own usage. You may find it helpful to develop your own Package because all class files are grouped in single directory and you can easily reuse all files whenever you need to. There are important steps involved in creating user defined packages:

Step 1: Create a directory in bin folder of Java Installed location.

Step 2: Give any name for that folder; it will be the name for your package, too.

Step 3: Define your classes and store them in your newly created package folder.

Step 4: Import just like pre-defined packages.

The following table explains the accessibility level of packages and their classes.

	Private	Public	Protected	No Modifier
Same Package Same Class	Yes	Yes	Yes	Yes
Same Package Sub Class	No	Yes	Yes	Yes
Same Package Non Subclass	No	Yes	No	Yes
Different Package Subclass	No	Yes	Yes	No
Different Package Non Subclass	No	Yes	No	No

Syntax for Creating Package

```
package package_name;
```

```
class class_name
```

```
{
```

```
Members of the class;
```

```
.....
```

```
.....
```

```
.....
```

```
}
```

Example Program

Create a new package and save your package as pack1.

Class number 1 in pack1

```
package pack1;
public class Factorial
{
    public int getFact(int n)
    {
        int fact1=1;
        for(int i=1;i<=n;i++)
            fact = fact * i;
        return fact;
    }
}
```

Class number two in pack1

```
package pack1;
public class Fibonacci
{
    public void fib(int n)
    {
        int f1=0,f2=1,f3,i;
        System . out. println(f1);
        System . out. println(f2);
        For(i=2;i<=n;i++)
        {
            f3=f1+f2;
            System . out. println(f3);
            f1=f2;
            f2=f3;
        }
    }
}
```

Importing package pack1 to your program

```
import pack1.Factorial;
import pack1.Fibonacci;
class SamplePack
{
public static void main(String args[])
{
Factorial f1=new Factorial();
f1.getFact(5);          //It will print the Factorial value of Integer 5 as 120
Fibonacci f2=new Fibonacci();
f2.fib(5);             // It will generate the Fibonacci series.
}
}
```

In this program, we have created a new package called by the name pack1 that contains two different classes: Factorial and Fibonacci. Both are public classes with only one method. These two methods are also public methods. The first method is getFact(), which will calculate the factorial value of the given parameter data. The second method will generate the Fibonacci series of the passed parameter value. In the SamplePack class, we have created an object for the Factorial as well as the Fibonacci class with the help of objects that you can invoke using the getFact and fib method to get the result.

Chapter 14

Debugging

Sometimes, your code may give the wrong outputs. This is called a logic error. Your IDE will not tell you that anything is wrong, because technically your code follows the rules of Java. It's like writing in any language. You may follow the rules of the language, like grammar, syntax, etc., but your paragraphs won't necessarily make sense. The most common method of debugging is simply using `System.out.println()` to print out variables that will help you to find where values stop making sense. Then you can find the lines that are causing the error and debug them. Another tool for debugging is the use of breakpoints. These are points that act as "checkpoints" to stop the code from executing until the programmer lets it continue. In Eclipse, on the line where you wish to put a breakpoint, simply move your cursor to the left-hand side, right-click, and press Toggle Breakpoint. This way you can slow down your code to find out where the error is occurring (or when, if you are doing lots of iterations in a loop).

One thing that may or may not be considered debugging is optimization. Once you've written your code, it may be tempting to just leave it and continue on with other tasks, but this may not be the best idea. If your program eventually returns the desired results but it takes a long time, then it is not very good. With that being said, there is no single strategy to optimize code. Sometimes your logic could simply be a longer method of doing something, and sometimes it could be how you have implemented that logic. Be wary of creating unused variables, as they can take up processing time and memory, thanks to a system called "garbage collection" that goes through all the variables in your code and removes any that are no longer valid. For example, if you make a for loop, that variable will no longer be valid once it completes, so it should be deleted to save memory. This takes up (small) amounts of processing time.

Here is an optimization example: Say you want to find the distance from one point to another. Math class tells us to use the Pythagorean Theorem, but this is rather slow. First the computer must square both sides (which is quite expensive, performance-wise), then add them, and finally take their square roots. This calculates the accurate distance between two points. Another

solution, however, could be to use “Manhattan distance.” This will not get you accurate distances, but it could be good in situations where speed is more important than the exact value. To do Manhattan distance, simply absolute (which means to make a value positive, so -7 becomes 7 and 8 becomes 8) both the x and y components and then add them. This gives you an inaccurate distance, but it is much faster than its more accurate counterpart. This is particularly good when you are guessing the fastest route. Rather than constantly calculating the accurate distance, Manhattan distance is a cheap alternative that will get you near enough.

Chapter 15

Enums

Enums are essentially pre-sets that have a limited number of possibilities. They are similar in definition to classes and interfaces, except they use the enum keyword instead. They can have constructors and variables. Enums cannot be instantiated or be children (but they can implement interfaces). Be careful when accessing variables within an enum, because if you modify an enum's variables, all instances of that enum will be modified. To fill an enum, you need a list of all pre-sets. This list must be at the top of your enum block. If your enum does not have a constructor, you may simply write out the possible values as if they were variable names. The rest you would separate by commas and end with a semicolon. If your enum does have a constructor, you must write the variable name, then the parameters of the constructor (meaning the values) separated by commas, and end with a semicolon. After the semicolon, you may go on to write your variable declarations, constructor(s), and any methods you wish. The constructor(s) cannot be public, only protected or private.

```
enum Number
{
    One , Two , Three
}
```

```
enum Color
{
    Red (255, 0, 0),
    Green (0, 255, 0),
    Blue (0, 0, 255),
    Purple (255, 0, 255),
    Yellow (255, 255, 0),
    Cyan (0, 255, 255);
    public int r;
    public int g;
    public int b;
    Color(int redChannel, int greenChannel, int blueChannel)
    {
```

```
r = redChannel;
g = greenChannel;
b = blueChannel;
}
public static void main(String[] args)
{
    Number num = Number. Two ;
    Color red1 = Color. Red ;
    Color red2 = Color. Red ;
    red1.g = 255;
    System. out .println(red2.g); // “255”, red2 was modified because both red1
    and red2 are the same object. That object is stored in both variables, and so is
    affected by changes in either object.
}
}
```


Chapter 16

Generic Types

Generic types allow us to make classes that are generic, which means that they can act differently based on what type is given. Generic types are given a list of classes by the programmer that will be used to define certain object types. Generic types are created by placing angled brackets (< >) after the class name, followed by a list of type identifiers. These work just as variable names, but the convention is to use capital letters, specifically T (for type), for one of them. These type identifiers can be used in place of data types. This allows you to keep certain variables a consistent type in order to prevent the need for casting. When creating a variable with the data type of the class using generic types, the data type must also have angled brackets containing a list of types corresponding to the class's list of types. The object you create will also need the same set of angled brackets before the parameters

```
class Something<A, B>
{
    public A variableA;
    public B variableB;
    public Something(A varA, B varB)
    {
        this.variableA = varA;
        this.variableB = varB;
    }

    public static void main(String[] args)
    {
        // Class1 will be A, Class2 will be B
        Something<Class1, Class2> somethingElse = new Something<Class1,
        Class2>(new Class1(), new Class2());
        Class1 class1 = somethingElse.variableA; // This does not need to be type
        cast because variableA is actually of the Class1 type
    }
}

class Class1{}
```

```
class Class2{}
```

One common use of generic types is `ArrayList`, a class in the `java.util` package. `ArrayList` is a dynamic array, meaning that you can change the number of elements in the array after it has already been initialized. `ArrayList` uses generic types to prevent programmers from adding objects of the wrong type, and also helps programmers to cast objects to the desired type. Here is a small example using `ArrayList`.

```
class Food
{
    public String name;

    public Food(String name)
    {
        this.name = name;
    }

    public static void main(String[] args)
    {
        ArrayList<Food> basket = new ArrayList<Food>();
        basket.add(new Food("Pear"));
        basket.add(new Food("Apple"));
        basket.add(new Food("Orange"));
        basket.add(new Food("Apple"));
        basket.add(new Food("Pomegranate"));
        System.out.println(basket.get(2).name); // "Orange"
        basket.remove(3);
        System.out.println(basket.size()); // "4"
    }
}
```

A good use of Array Lists involves “queues.” This is a way to do recursion (calling a method from within that method) without having to do recursion. Recursion is rather dangerous, since improper use can cause infinite loops that can quickly crash your program. Recursion is actually rather slow, for a variety of reasons, so it’s a good thing we have the option of using queues.

Essentially, you create an ArrayList of data (usually in the form of a class), add a starting element to the array, and then for as long as the ArrayList is not empty (or hasn't completed your task in some way), it will continue to perform some code given the values in your ArrayList before it ultimately removes that element from the ArrayList. Your code could add elements to the ArrayList that add new cycles, just as in a recursive method, and you could call the method again to add a new layer of depth (like a cycle).

Queues are not necessarily "better" than recursive methods. Queues are meant for branching recursive "paths," specifically in the case of finding optimal solutions. This could be used in path finding for video games, or for real life. Logically, we could try a path that seems closest to the target, and if that path dead-ends, we can remove it from the queue and try the next closest path. Recursion could do this, but it would work like a tree (iteration-wise). Each path would most likely go from left to right and would go through each possibility. With queues, you can go from any part of the iteration tree to any other part of the iteration tree, which allows computers to potentially find a route using the fastest method. Both recursion and queues could reach the same end, but it depends on the way you have structured your system. For single-path recursion, meaning the recursive method only calls itself once per method call, it is usually best to use a recursive method. If you are doing something where you are trying to find some sort of path, it may be faster to use a queue. But this is all speculation. Realistically, if you want to properly optimize your code, you simply need to test and see which version is faster. Remember that you need a variety of test cases when optimizing. Simply testing a few similar types of test cases can cause performance increases in other tests. For example, your code could take an exponential amount of time, or a linear amount of time, and different ways of doing something could have different "equations" for their amount of run time, which certain test cases may not take advantage of.

Chapter 17

Threading

Threading is a way to run two “threads” of code at the same time. Threading does not necessarily make your program run faster, but it can in select situations. Threading shares processing power, so it does not speed up any code. Threading works like two very nice people walking through a corridor divided by tons of doors. The first person lets the second go through the first door, then the second person lets the first go through the second door, and repeat. What threading allows programmers to do is run loops that will run for a very long time while still doing other things in the background. For example, if you want your program to accept a connection from another computer, you could run a loop that constantly “listens” for a connection. The problem with this is that your program will freeze while it listens. Instead, we use a thread to listen for a connection, and then do other things in our main thread. We would show the user something like a GUI (next section) and give them feedback based on the listening status. So how do we make a thread? First, we make a “Runnable.” Simply implement Runnable and then override its run method. Next, we create a thread object with the parameter of a MyRunnable object (or any class that implements Runnable), and finally call <thread object>.start().

```
class MyRunnable implements Runnable
{
    public void run()
    {
        while(true) // This loops runs forever!
        {
            System.out.println("Groundhog Day!");
        }
    }
}
```

```
public static void main(String[] args)
{
    Runnable runThis = new MyRunnable();
    Thread aThread = new Thread(runThis);
    aThread.start();
}
```

```
while(true)
{
System.out.println("Bill Murray!");
}
}
}
```

The code above will now run two infinite, unrelated loops, and will “randomly” print out either “Groundhog Day!” or “Bill Murray!” Threading is not limited to infinite loops, though; they are simply for whenever you need two concurrent lines of processing. For example, say you’re making a loading menu for a video game. You may make one thread load the game data, and then simply put its status into a variable (like how many files have been loaded). Then you could have another thread modify your window or console or whatever to tell the user the status of loading based on the variable that the original thread is using. This is how (most) loading bars are made.

Life Cycle of Thread

There are four stages are involved in the life of cycle of a thread. They are

- Born state
- Runnable state
- Blocked state
- Dead state

New Born State

When a thread is created, it is in the new state. New implies that the thread object has been created but it has not started running. It requires the start () method to start it.

Runnable State

A thread is said to be in runnable state while it is executing a set of instructions. The run () method contains the set of instructions. This method is called automatically after the start () method.

Blocked State

The thread could be run but there is something preventing it. While a thread is in a blocked state, the scheduler will simply skip over it and not give it any CPU time. Until a thread reenters a runnable state, it will not perform any operations.

wait () - notify()
suspend () - resume()
sleep(milliseconds)

Dead State

The normal way for a thread to die is by returning from its run () method. We can also call stop (), but this throws an exception that is a subclass of Error (which means we normally do not catch it).

Creation of Threads

In Java, you can create a thread in two different ways. The first one would be to use Runnable interface, and second one uses the Thread Class. Runnable interface has only one method, which is run() method and a constructor. Thread class contains so many methods with help of that methods you can know the basic information's of the thread.

Common Methods

- **start()**

The start() starts execution of the invoking object. It can throw an IllegalStateException if the thread was already started.

- **stop()**

This method terminates the invoking object.

- **suspend()**

This method suspends the invoking object. The thread will become runnable again if it gets the resume () method.

- **sleep()**

This method suspends execution of the executing thread for the specified number of milliseconds. It can throw an InterruptedException.

Thread Class Methods

- **activeCount()**

This method will return the current number of active Threads in this thread group.

- **currentThread()**

This will return a reference to the currently executing thread object; it is a static method.

- **isAlive()**

This method will check whether a given thread is alive.

- **getName()**

This will get and return this Thread's name.

- **getPriority()**

This will get and return the Thread's priority.

- **setName(String)**

This method is used to set the Thread's name.

- **setPriority(int)**

This is the method used to set the Thread's priority.

- **toString()**

This will return a string representation of the Thread, including the thread's name, priority, and thread group.

Example: Thread program created with Runnable interface

```
class RunnableDemo implements Runnable
{
    Thread t;
    RunnableDemo(String name)
    {
        t = new Thread(this,name); //new born state
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println(t.getName() + " : " + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Exception : " + e);
        }
    }
}
```



```

}
public static void main(String args[])
{
    System.out.println("Main Thread");
    RunnableDemo d1 = new RunnableDemo("Thread No");
    try
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("Thread Yes : " + j);
            Thread.sleep(1000);
        }
    }
    catch(InterruptedException e)
    {
        System.out.println("Exception : " + e);
    }
}

```

In this example program, we have created two threads with the help of Runnable interface. The name of the first thread is Thread No and the second one is Thread Yes. First we create an object for a RunnableDemo class that will invoke the constructor the RunnableDemo, where we are creating a new thread called Thread No. This thread goes into Sleep mode every 500 milliseconds so the second thread will be executed and print Thread Yes 1 before it goes into sleep mode. Next, Thread No will restart its execution and run until the given condition becomes false.

Thread Priority

Thread Priority changes the execution order of the thread. Maximum Priority thread will start its execution first, and Minimum Priority starts its execution last. You can specify the priority level as 1 to 10; we have some constants, too: MIN_PRIORITY – 1, NORM_PRIORITY-5, MAX_PRIORITY-10.

The setPriority() method helps us to specify the Priority of the thread, as in the following example:

Example

```
class Prio extends Thread {
    Thread t;
    Prio(String m) {
        t = new Thread(this,m);
        System.out.println("\n Thread : "+t.getName());
        System.out.println("Priority : "+t.getPriority());
        t.start();
    }
    public void run() {
        System.out.println("New Priority : "+t.getPriority());
        try {
            System.out.println("\n"+t.getName()+" is in wait stage");
            t.sleep(1000);
        }
        catch(Exception e) { }
        System.out.println(t.getName()+" completed the Process");
    }
}

class Priority {
    public static void main(String a[]) {
        System.out.println("\n Main Starts the Process");
        Prio x = new Prio("First Thread");
        Prio y = new Prio("Second Thread");
        x.t.setPriority(3);
        y.t.setPriority(8);
        System.out.println("\n Main Completed the Process");
    }
}
```

In this program we are creating two threads named First Thread and Second Thread. In the beginning, First Thread has Normal priority, so it starts its process. After calling the first thread constructor, it goes to the Sleep mode and then the Second Thread execution starts; it also goes into Sleep mode but Second Thread will restart its execution because it has higher priority than First Thread.

Chapter 18

Graphical User Interface (GUI)

Throughout this guide, you have been programming and expecting output from a console. The time has come to learn how to develop your own interface. In this part of the guide, you will learn Java Swing and Applet Programming. There are many different ways to develop an interface in Java, but we are going to be using the “Swing and Applet” method.

Applet

First we will explore Applet Programming. This is also one way to develop a user interface. Applet is a kind of container where we can place our components to create our user interface. Applet class is defined under the `java.applet.*` package and all other components and event handling classes are defined under the `java.awt.*` and `java.awt.event.*`; packages. In this part, we are going to learn how to develop a user interface with the help of Applet and AWT components. Applet is used to create an interactive dynamic program that is run in Applet Viewer or any browser. There are, however, some restrictions: Applet applications are easily affected by viruses and we cannot use file systems such as read and write.

Structure of Applet Program

```
import java.applet.*;
import java.awt.*;
public class SampleApplet extends Applet
{
    public void init()
    {}
    public void start()
    {}
    public void paint(Graphics g)
    {}
    public void stop()
    {}
}
```

```
public void destroy()  
{  
}
```

In above sample program, you can see five methods that are involved in the life cycle of the applet. You don't need to call those methods, because all methods are called automatically. You can only call paint() method by yourself to update the applet window, but you have to use repaint() method to call paint() method.

All applet programs require an applet tag because it is important to define the appearance of the applet window. The following syntax will help you to understand the applet tag and its attributes.

Applet Tag

```
/*<applet  
code="Applet class file name"  
codebase="path of the applet class file"  
width="width of the applet window in pixels"  
height="height of the applet window in pixels"  
vspace="vertical space"  
hspace="horizontal space"  
>  
</applet>*/
```

In this applet tag code, width and height are the conditional attributes, so you have to mention them. The remaining three (vspace, hspace, codebase) are optional.

Parameter Tag

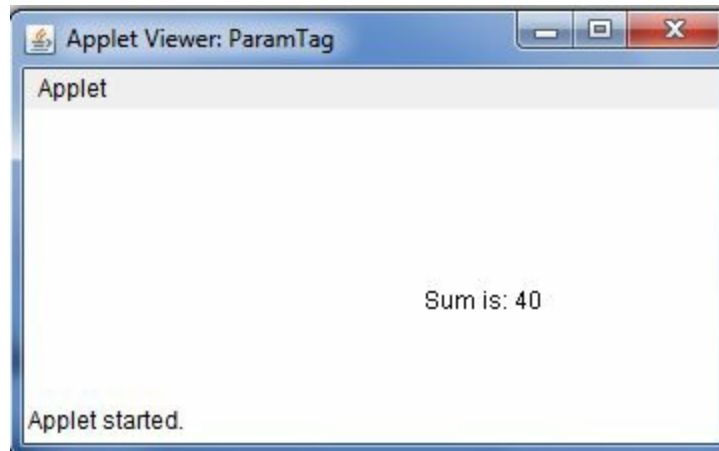
Param tag helps us to pass some parameter data to the applet program. This tag has two attributes: Name and Value. Name contains the name of the parameter and value holds the data.

```
<param name="name of the parameter name" value="data"></param>
```

Sample Program

```
import java.applet.*;
import java.awt.*;
public class ParamTag extends Applet
{
    int a,b;
    String a1;
    public void init()
    {
        a=Integer.parseInt(getParameter("Data1"));
        b=Integer.parseInt(getParameter("Data2"));
        a1="Sum is: "+(a+b);
    }
    public void paint(Graphics g)
    {
        g.drawString(a1,200,100);
    }
}
/*<applet code="ParamTag" width="200" height="200">
<param name="Data1" value="15"></param>
<param name="Data2" value="25"></param>
</applet>*/
```

In this example program, we have created two param tags named Data1 and Data2 with values of 15 and 25 respectively. Then we are getting those data with the help of the `getParameter()` method. This will always return the data in String format, so we have to convert those data to integer data. Then we add those data and store them in string variable in order to print that data to the applet window.



Classes in Applet Programming

- Graphics class
- Font class
- Color class
- Image class
- Graphics Class

This class contains methods for drawing strings, lines, rectangles, and other shapes defined in the graphics class.

Graphics Class

- **Drawing Characters, Bytes, and Strings**

“g,” “r,” “a,” “p,” “h,” “i,” “c,” “s”—Characters

100, 101, 108, 114, 67, 47—Bytes

“World”—String

void drawBytes (byte[] data , int offset , int length , int x , int y)

This method is used to draw byte data to the Applet window.

data -> the data to be drawn

offset-> the start offset in the data

length-> the number of bytes that are drawn

x-> x coordinate

y-> y coordinate

void drawChars(char[] data , int offset , int length ,int x , int y)

This method is used to draw character data into the applet window.

abstract void drawString(String str,int x,int y)

This method is used to draw string data into the applet window.

- **Shape Drawing Methods**

We have plenty of methods to draw different types of shapes like rectangles, circles, arcs, and so on.

Method name	Parameters	Usage of methods
drawLine(int x1,int y1,int x2,int y2)	x1, y1: starting point x2, y2: ending point	This method helps us to draw a line shape to the applet window.
drawRect(int x1,int y1,int width,int height)	x1, y1: starting point width: width of the rectangle height: height of the rectangle	This method helps us to draw a rectangle shape to the applet window.
fillRect(int x1,int y1,int	x1, y1: starting point	This method helps us

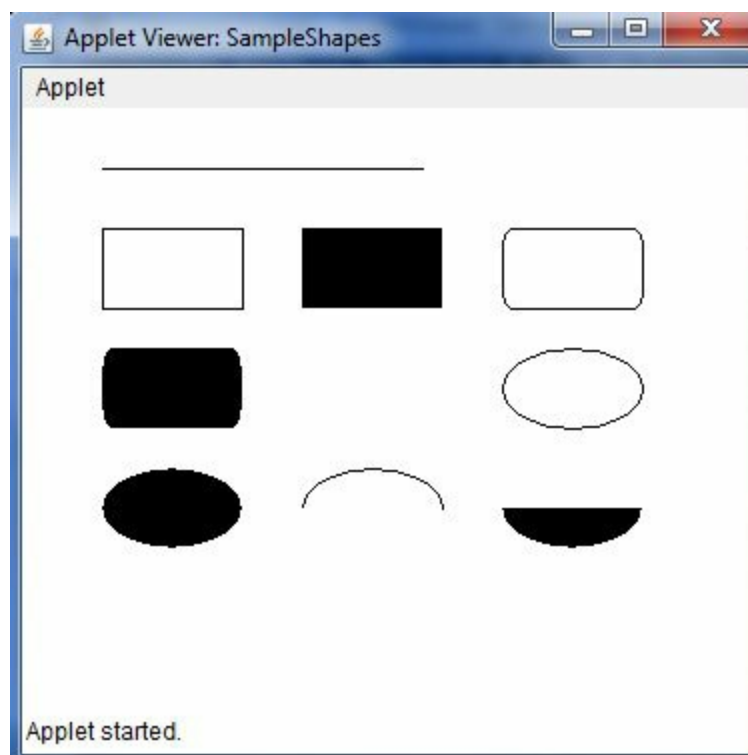
width,int height)	width: width of the rectangle height: height of the rectangle	to draw a rectangle shape to the applet window with its interior region filled with the same color as the outline.
drawRoundRect(int x1,int y1,int width,int height,int angle1,int angle2)	x1, y1: starting point width: width of the rectangle height: height of the rectangle width1: width1 of the angle corners height1: height1 of the angle corners	This method helps us to draw a rounded rectangle shape to the applet window.
fillRoundRect(int x1,int y1,int width,int height,int angle1,int angle2)	x1, y1: starting point width: width of the rectangle height: height of the rectangle width1: width1 of the angle corners height1: height1 of the angle corners	This method helps us to draw rounded rectangle shape to the applet window with its interior region filled with the same color as the outline.
drawPolygon(int xs[],int ys[],int pts)	xs: an array of integers representing x coordinates ys: an array of integers representing y coordinates pts: an integer representing the total number of points	This method helps us to draw a polygon shape to the applet window.
fillPolygon(int xs[],int	xs: an array of integers	This method helps us

ys[],int pts)	representing x coordinates ys: an array of integers representing y coordinates pts: an integer representing the total number of points	to draw a polygon shape to the applet window with its interior region filled with the same color as the outline.
drawOval(int x1,int y1,int width,int height)	x1, y1: starting point of the oval width, height: width and height of the oval	This method helps us to draw an oval shape to the applet window.
fillOval(int x1,int y1,int width,int height)	x1, y1: starting point of the oval width, height: width and height of the oval	This method helps us to draw oval shape to the applet window with its interior region filled with the same color as the outline.
drawArc(int x1,int y1,int width,int height,angle1,angle2)	angle1: degree value at which the arc starts angle2: degree value at which the arc ends	This method helps us to draw an arc shape to the applet window.
fillArc(int x1,int y1,int width,int height,angle1,angle2)	angle1: degree value at which the arc starts angle2: degree value at which the arc ends	This method helps us to draw an arc shape to the applet window with its interior region filled with the same color as the outline.

Sample Program

```
import java.applet.*;
import java.awt.*;
```

```
//<applet code="SampleShapes" width="300" height="300"></applet>
public class SampleShapes extends Applet
{
public void paint(Graphics g)
{
g.drawLine(40,30,200,30);
g.drawRect(40,60,70,40);
g.fillRect(140,60,70,40);
g.drawRoundRect(240,60,70,40,10,20);
g.fillRoundRect(40,120,70,40,10,20);
g.drawOval(240,120,70,40);
g.fillOval(40,180,70,40);
g.drawArc(140,180,70,40,0,180);
g.fillArc(240,180,70,40,0,-180);
}
}
```



Font Class

We can write our data in various fonts inside the Applet window. Java offers

many fonts, including Courier, Times New Roman, and so on. Font class helps us to define the new font object and can be set to the Applet window by a pre-defined method.

Constructor

Font(String name,int style,int size)____Creates a new font from the specified name, style, and size

Methods

abstract void setFont(f)____Sets this graphic context font to the specified font

String getStyle()____Returns a string value indicating the current font style

int getSize()____Returns an integer value indicating the current font size

String getFamily()____Returns the font family name as a string

boolean isPlain()____Returns true if the font is plain (roman)

boolean isBold()____Returns true if the font is bold

boolean isItalic()____Returns true if the font is italic

Style Constants

Font.BOLD ____This is for bold font style

Font.ITALIC ____This is for italic font style

Font.PLAIN ____This is for plain (roman) font style

Sample Program

```
import java.awt.*;  
import java.applet.*;
```

```
//<applet code="SampleFont" width="600" height="600"></applet>
public class SampleFont extends Applet
{
    Font f1, f2;
    public void init()
    {
        f1=new Font("TimesRoman",Font.BOLD,25);
        f2=new Font("Courier",Font.ITALIC,40);
    }
    public void paint(Graphics g)
    {
        g.setFont(f1);
        g.drawString("This is Times Roman Font",100,100);
        g.setFont(f2);
        g.drawString("This is Courier Font",100,200);
    }
}
```



Color Class

Color class helps us to create different sets of colors with color class constructors. You can set the background and foreground color of the Applet window by using the color class methods. We have many pre-defined methods in color class.

Color Class Constructor

```
Color c1=new Color(Red,Green,Blue);
```

In this constructor, we have to specify the RGB values. These values start from 0 and go up to 255. When you specify the RGB values in this range, a new color will be produced depending on your data. We have color constants too, which are listed here with their equal RGB values.

Color Constant ___RGB Value

Color.red ___255, 0, 0

Color.green ___0, 255, 0

Color.blue ___0, 0, 255

Color.magenta ___255, 0, 255

Color.cyan ___0, 255, 255

Color.yellow ___255, 255, 0

Methods of Color Class

setColor()___This method will set color to an object. It has only one parameter, so you can pass any color constant value or color class object.

setBackground()___This method will set the background color of an applet. It also has only one parameter, which is color class object or color constant value.

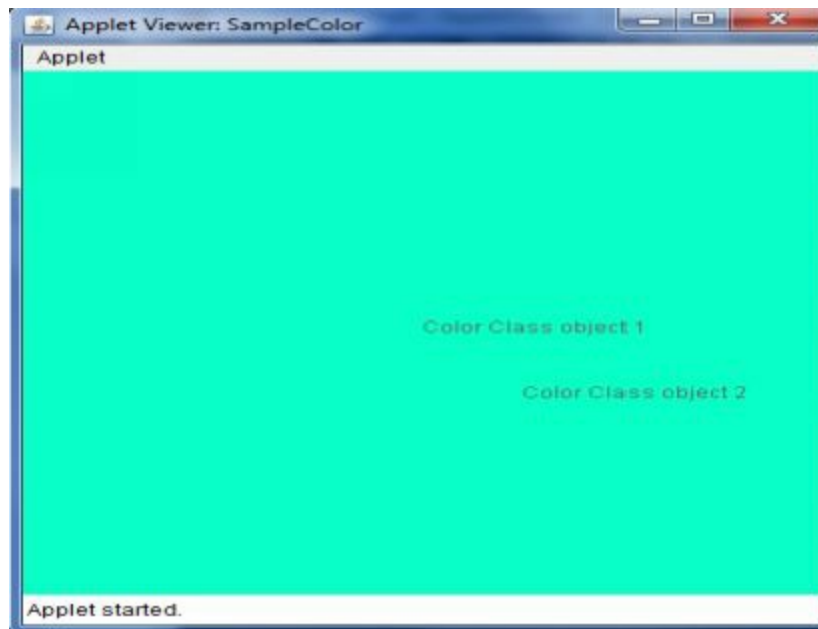
setForeground()___This method is used to change the color of the whole applet after it has been drawn.

Color getBackground()___This method will return the background color of the Applet window.

Color getForeground()___This method will return the foreground color class object.

Sample Program

```
import java.applet.*;
import java.awt.*;
//<applet code="SampleColor" width="400" height="400"></applet>
public class SampleColor extends Applet
{
    Color c1,c2;
    public void init()
    {
        c1=new Color(10,255,200);
        c2=new Color(100,100,100);
    }
    public void paint(Graphics g)
    {
        g.setBackground(c1);
        g.drawString("Color Class object 1",200,200);
        g.setForeground(c2);
        g.drawString("Color Class object 2",250,250);
    }
}
```



In this example program we have declared two color class objects: c1 and c2.

Then, c1 and c2 have been created inside the init() method by using the Color class constructor. Inside the paint() method, we have set c1 as the Background color of the applet window and c2 as the foreground color.

Components of the Applet Window

Components are nothing more than controls of the AWT package such as a text box, list box, choice box, button, scrollbar, and so on. Each component class has its own methods and constructor. Component constructors help us to create the component and add to the applet window. Here we have listed the important component classes along with their methods and constructors.

Component Class

We can distinguish the following components:

1. Label Components

The Label component helps us to display some text in the applet window. Of all the components, it is the only one that doesn't have any events. It just shows the text given inside the constructor of the label. Label class has its own constructor and methods.

Constructor of the Label Components

Label()___Empty constructor; will just create the label without any text information.

Label(String str)___Will create the label with given text.

Label(String str,int Alignment)___Will create a label that you can align wherever you want.

Methods:

getText()___Will return the label text from the component.

setText()____Will set the text as a label.

Sample Program

```
import java.awt.*;
import java.applet.*;
//<applet code="SampleLabel" width="200" height="200"></applet>
public class SampleLabel extends Applet
{
    Label L1,L2,L3;
    public void init()
    {
        L1=new Label("First Label",Label.LEFT);
        L2=new Label("Second Label",Label.RIGHT);
        L3=new Label("Third Label");
        add(L1);
        add(L2);
        add(L3);
    }
}
```

In this example we have added three label box components: L1, L2, and L3. The first label box, with the label "First Label," is aligned on the left side; the second label box, with the label "Second Label," is aligned to the right. The final one, with the label "Third Label." All label boxes are added to an applet window. As has already been said, label boxes don't have any events and so they only display the label specified in the label component constructor.

2. Button Component

Button components are used to perform any action or task when the user presses that button. We have a button component class with two constructors and some predefined methods to handle the button component. If you press the button, it will generate an Action Event that is handled by the ActionListener Interface. It has a method that is executed when we press the button. Once the Action Event is generated, the void actionPerformed(ActionEvent e) method will be executed. In the following

section, we will see what kind of methods are defined under the Button Component class.

Constructors of Button Class

Button() ____This is an empty constructor. It will create a button without any label text.

Button (String str) ____This is a parameterized constructor. It will create a button with label text.

Methods of Button Class

addActionListener(ActionListener L)____Adds an Event to the newly created component.

getActionCommand() ____Returns label text of the Button after you press the button.

getLabel()____Returns the label text of the button.

setLabel(String text)____Sets the given text as a Button label.

getSource()____Returns the pressed button object.

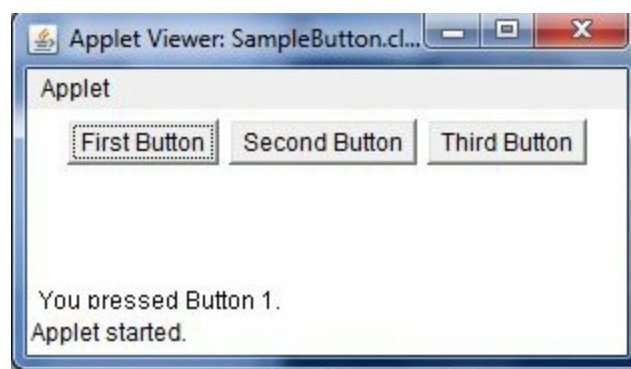
Sample Program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SampleButton extends Applet implements ActionListener {
String str = " ";
Button b1, b2, b3;
public void init() {
b1 = new Button ( "First Button");
b2 = new Button ("Second Button");
b3 = new Button ("Third Button");
```

```

add(b1);
add(b2);
add(b3);
b1.addActionListener (this);
b2.addActionListener (this);
b3.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
if(ae.getSource()==b1) {
str = "You pressed Button 1.";
}
else if(ae.getSource()==b2) {
str = "You pressed Button 2.";
}else {
str = "You pressed Button 3.";
}
repaint();
}
public void paint(Graphics g) {
g.drawString(str,6,100);
}
}
//<applet code="SampleButton.class" width = 300 height = 100></applet>

```



In this example, we have created the three buttons b1, b2 and b3 with the labels First Button, Second Button, and Third Button. If you click the First Button, it will immediately generate the Action Event so that the actionPerformed () method will be called and it will print the “You pressed

Button 1” message on the Applet window. This same functionality also applies to the second and third buttons.

3. Check Box Component

The check box component helps us to choose more than one item from multiple choices. We have a predefined class called Checkbox that has more than one constructor to create checkboxes and methods to manipulate those checkboxes and retrieve the checkbox details.

Constructors of the Checkbox Component

Checkbox () ____Creates a checkbox without label text.

Checkbox (String str)____Creates a checkbox with label text.

Checkbox (String str, Boolean state)____Creates a checkbox with label text and you can test the checkbox with Boolean state. If you pass a true value, then it will create the checkbox with a check mark; if false, then it won't have a check mark.

Checkbox (String str, CheckboxGroup, Boolean state)____Used to convert a checkbox to a Radio button. If you pass the CheckboxGroup object, then it will create a Radio button.

Methods of Checkbox

getState ()____Returns the state of the Checkbox. It will return true if the checkbox is selected or false if not selected.

getItem()____Returns the item object of the particular Checkbox.

setCheckboxGroup(CheckboxGroup g)____Sets the corresponding checkbox group to the specified one.

setLabel(String label)____Sets the label of the corresponding checkbox to the value specified.

setState(boolean state)____Sets the state of the corresponding checkbox to the value specified.

CheckboxGroup getCheckboxGroup()____This method determines the group of the corresponding checkbox.

getLabel()____Returns the name of the corresponding checkbox.

getSelectedObjects()____Returns an array (length 1) containing the checkbox label or null if the checkbox is not selected.

Sample Program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SampleCheckbox extends Applet implements ItemListener {
String str = new String();
Checkbox fe,lam,jag,au;
public void init() {
fe = new Checkbox("Ferrari",null,true);
lam = new Checkbox("Lamborghini");
jag = new Checkbox("Jaguar");
au = new Checkbox("Audi");
add(fe);
add(lam);
add(jag);
add(au);
fe.addItemListener(this);
lam.addItemListener(this);
jag.addItemListener(this);
au.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
```

```
public void paint(Graphics g) {  
    str = "Selected Brand : ";  
    g.drawString(str,6,80);  
    str = " Ferrari: " + fe.getState();  
    g.drawString(str,6,100);  
    str = " Lamborghini: " + lam.getState();  
    g.drawString(str,6,120);  
    str = " Jaguar: " + jag.getState();  
    g.drawString(str,6,140);  
    str = " Audi: " + au.getState();  
    g.drawString(str,6,160);  
}  
}  
//<applet code="SampleCheckbox.class" width=300 height=200></applet>
```



In this example, we have created four checkboxes with the help of constructors. First we need to declare the checkbox objects along with the checkbox labels. Checkbox generates the /Item Event, so all checkbox events are handled by the ItemListener Interface. The itemStateChanged() method will call the repaint () method. There we can get all the states of the checkbox by using the getState() method. Since it is a checkbox, you can select more than one.

4. Radio Button

This is similar to checkbox, but you cannot select more than one item. In Java, there is not a separate class for the Radio button. You can create a radio button using the Checkbox constructor. You simply pass the checkbox group and it will convert the Checkbox to a Radio button. The following program explains everything.

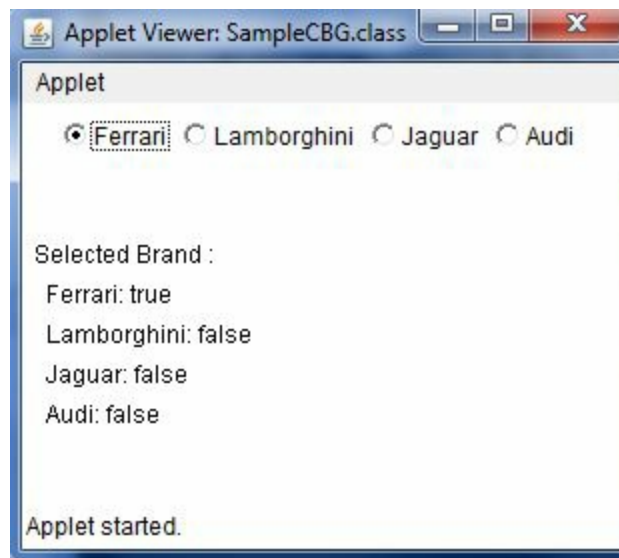
Sample Program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SampleCBG extends Applet implements ItemListener {
    String str = new String();
    Checkbox fe,lam,jag,au;
    CheckboxGroup cbg;
    public void init() {
        fe = new Checkbox("Ferrari",cbg,true);
        lam = new Checkbox("Lamborghini",cbg,false);
        jag = new Checkbox("Jaguar",cbg,false);
        au = new Checkbox("Audi",cbg,false);
        add(fe);
        add(lam);
        add(jag);
        add(au);
        fe.addItemListener(this);
        lam.addItemListener(this);
        jag.addItemListener(this);
        au.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    public void paint(Graphics g) {
        str = "Selected Brand : ";
        g.drawString(str,6,80);
        str = " Ferrari: " + fe.getState();
```

```

g.drawString(str,6,100);
str = " Lamborghini: " + lam.getState();
g.drawString(str,6,120);
str = " Jaguar: " + jag.getState();
g.drawString(str,6,140);
str = " Audi: " + au.getState();
g.drawString(str,6,160);
}
}
//<applet code="SampleCBG.class" width=300 height=200></applet>

```



In this program we have created a radio button by using the Checkbox and CheckboxGroup constructor. Again, we don't have a separate component class for the Radio button, but we can create the Radio button with the help of the Checkbox Constructor when we pass the CheckboxGroup object as an argument. In this program we have added four radio buttons, each representing the brand name of a car. Once we select any brand name, it will immediately call the `itemStateChanged()` method, here called "repaint" method. Inside the `paint()` method we have printed the selected radio button state by using the `drawString()` method.

5. Choice Component

A choice box is nothing but an ordinary combo box control, which is a

combination of the Text and List boxes. You can select any item from the list box and it will immediately be displayed in the text box. In Java, we have a Component class choice to create the choice box. In Choice box, we have a constructor and a set of methods to handle and retrieve the information of the choice box.

Constructor of the Choice Component

```
Choice ch=new Choice();
```

Methods of the Choice Component

`add(String item)`____Adds an item to the corresponding choice box.

`addItem(String item)`____This method also adds an item to the corresponding choice box

`getItem(int index)`____Returns the string at the specified index of the corresponding choice box.

`getItemCount()`____Returns the number of items in the corresponding choice box.

`getSelectedIndex ()`____Returns the index of the currently selected item of the corresponding choice box.

`getSelectedItem()`____Returns the current selected item of the choice as string data.

`insert(String item, int index)`____Inserts the item into the corresponding choice at the specified position.

`remove(int position)`____Removes an item from the corresponding choice box at the specified position.

`remove(String item)`____Similar to the previous one, this will remove the first occurrence of an item from the corresponding choice menu.

removeAll()____Removes all items from the corresponding choice menu.

select(int pos) _____Sets the selected item in the corresponding choice menu to be at the specified position.

select(String str)____Sets the selected item in the corresponding Choice menu to be the item whose name is equal to the specified string.

Sample Program for Choice Component

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SampleChoice extends Applet implements ItemListener {
    Choice country,city;
    String str = new String();
    public void init() {
        country = new Choice();
        city = new Choice();
        country.add("India");
        country.add("USA");
        country.add("England");
        country.add("Canada");
        city.add("Newyork");
        city.add("New Delhi");
        city.add("London");
        city.add("Toronto");
        add(country);
        add(city);
        country.addItemListener(this);
        city.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    public void paint(Graphics g) {
        str = "Selected Country : ";
```

```

str += country.getSelectedItemAt();
g.drawString(str,6,120);
str = "Selected City : ";
str += city.getSelectedItemAt();
g.drawString(str,6,140);
}
}
//<applet code="SampleChoice.class" width=500 height=500></applet>

```



In this program there are two choice boxes that were added with the help of the Choice() component constructor. We have then added a list of items in the choice box using the add() method. Both choice boxes have four items. Choice box generates the Item Event, so this event is handled by the Item Listener interface method itemStateChanged(). Each time you select any item from the list, the itemStateChanged() method will be called; here we are calling the repaint() method. Inside the repaint method, the selected item of both choices will be printed in the applet window using the drawString() method.

6. List Box

This component contains a list of items that we can select from. This

component also generates an Item Event, so it is also handled by the Item Listener method `itemStateChanged()`. This component class contains three constructors to create the List box and a set of methods to handle the list box item.

Constructors of List box

`List()`____Creates a scrollable list box.

`List(int rows)`____Create a scrollable list box that only shows the number of items specified in the constructor; the rest of the items will become visible as you scroll through the list box.

`List(int rows, Boolean Multiple)`____Creates a scrollable list box that only shows the number of items specified in the constructor; the rest of the items will become visible as you scroll through the list box. If you set multiple mode as true mean, then you can select more than one item in the list box.

Methods of the List component class

`add(String item)`____Adds the specified item at the end of the scrolling list.

`add(String item, int index)`____Adds the specified item at the specified position.

`deselect(int index)`____Deselects the item at the specified index.

`getItem(int index)`____Returns the item at the specified index.

`getItemCount()`____Returns the number of items in the list component.

`getItems()`____Returns the items in the list

`getRows()`____Returns the number of visible lines in the corresponding list.

`int[] getSelectedIndexes()`____Returns the index of the selected item in the list.

getSelectedItem()____Returns the selected item in the corresponding list.

select(int index)____Selects the item at the specified index in the corresponding list.

setMultipleMode(Boolean b) _____Sets the flag that allows multiple selection in the corresponding list.

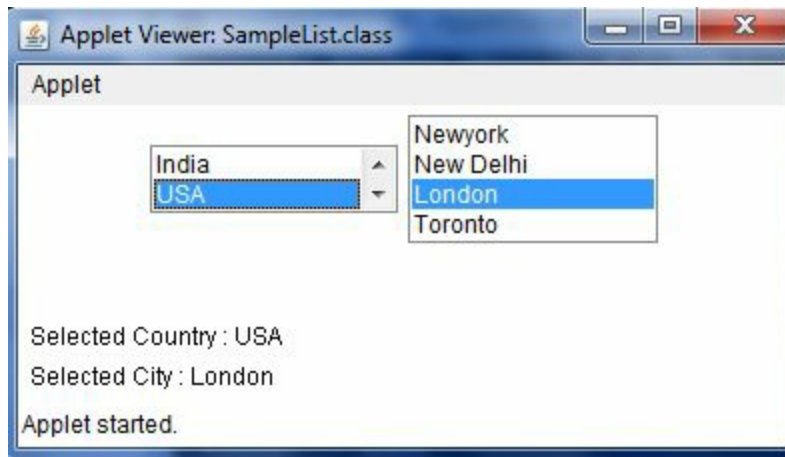
Sample Program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SampleList extends Applet implements ItemListener {
    List country,city;
    String str = new String();
    public void init() {
        country = new List(2);
        city = new List(4);
        country.add("India");
        country.add("USA");
        country.add("England");
        country.add("Canada");
        city.add("Newyork");
        city.add("New Delhi");
        city.add("London");
        city.add("Toronto");
        add(country);
        add(city);
        country.addItemListener(this);
        city.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    public void paint(Graphics g) {
        str = "Selected Country : ";
```

```

str += country.getSelectedItemAt();
g.drawString(str,6,120);
str = "Selected City : ";
str += city.getSelectedItemAt();
g.drawString(str,6,140);
}
}
//<applet code="SampleList.class" width=500 height=500></applet>

```



In this example, there are two list boxes that were added using the List() component constructor. Then we have added a list of items in our choice box with the add() method. Both list boxes have four items. list box generates the Item Event, so this event is handled by the Item Listener interface using the itemStateChanged() method. Each time you select any item from the list, the itemStateChanged() method will be called; here we are calling the repaint() method. Inside the repaint method, the selected items from both list boxes will be printed in the applet window by the drawString () method.

7. Text Field Component

This is a single line data editor that allows you to enter text data on a single line. TextField Component class contains a set of constructors and methods to handle the TextField Data. Here we have listed the constructors and methods of the TextField Component class. It will generate a TextEvent, so it is handled by the TextListner textValueChanged() method.

Constructors of the TextField

TextField()
TextField(int length)
TextField(String str)
TextField(String str, int length)

Methods of the TextField Component class

getText()___Returns the text of the corresponding TextField.
setText(String str)___Sets the given text to the corresponding TextField.
getSelectedText()___Returns the selected text part of the TextField.
select(int startindex,int endindex)___Used to select the text.
setEditable(boolean canEdit)___If false, the text cannot be altered
isEditable()___Helps us to check if the text is editable or not.
setEchoChar(char ch)___Sets the printed character in text (password field).
echoCharIsSet()___Checks if the echo char is set or not.
getEchoChar()___Returns the echoed character.

8. TextArea Component

This is a multi-line data editor. Here you can enter your text data on more than one line. It also has a set of constructors and methods.

TextArea Constructors

TextArea()
TextArea(int numLines,int numChars)
TextArea(String str)

```
TextArea(String str,int numLines,int numChars)
TextArea(String str,int numLines,int numChars,int sBars)
```

Methods of the TextArea Component

All Text field components are also supported by Text Area, although they have some unique methods.

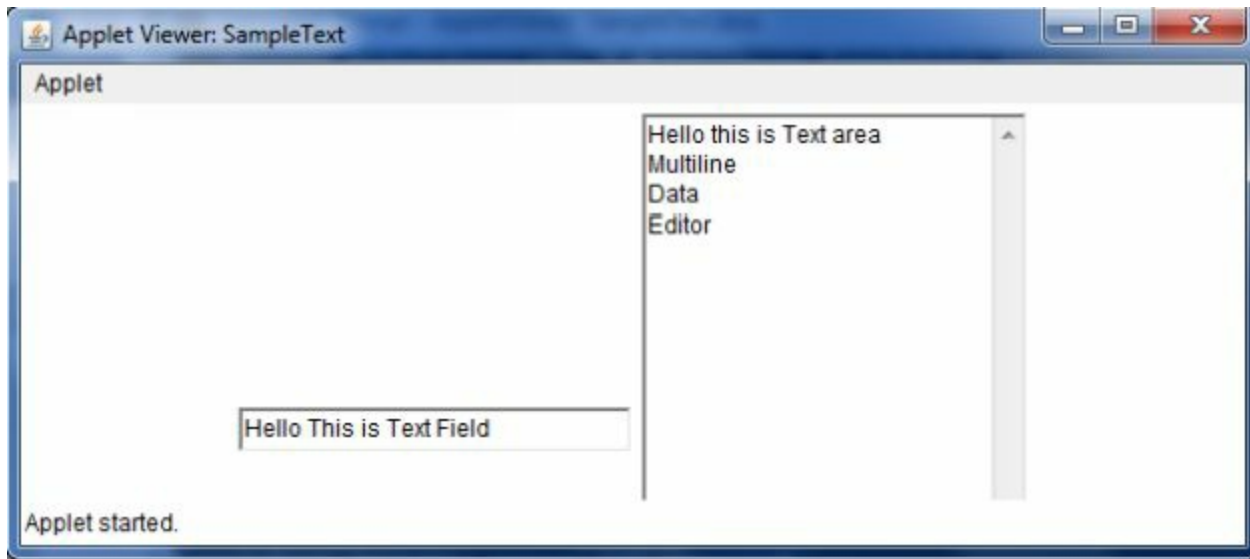
`void append(String str)` ____Appends the text at end of the line.

`void insert(String str,int index)`____Inserts the string in the corresponding index position.

`void replaceRange(String str,int startIndex,int ndIndex)`____Replaces the string with the corresponding start and end index.

Sample Program Using Text Field and Text Area

```
import java.awt.*;
import java.applet.*;
//<applet code="SampleText" width="200" height="200"></applet>
public class SampleText extends Applet
{
    TextField t1;
    TextArea ta1;
    public void init()
    {
        T1=new TextField(25);
        Ta1=new TextArea("Hello this is Text area",20,25,0);
        add(t1);
        add(ta1);
    }
}
```

9. Scrollbar Component:

Scrollbar is a predefined class under the `java.awt` package. This component class helps developers to add a new scrollbar to the applet window. This class has several types of constructors that we can use to create a control. We can create two different types of scrollbars with help of this component class: horizontal and vertical. You will need to specify what kind of scrollbar you want to create, because there are separate constants for horizontal and vertical scrollbars.

Constructors of the Scrollbar Component

Scrollbar()

This constructor will create a horizontal scrollbar. By default, a horizontal scrollbar will be created if you have not mentioned any style inside the constructor or if you use this empty constructor to create the scrollbar.

Scrollbar(int style)

This constructor also creates a scrollbar, but allows you to specify what kind of scrollbar you want to create by sending the style parameter. As we said earlier in this part, we can create two types of scrollbars.

Style:

Scrollbar.VERTICAL ____Creates a vertical scrollbar.

Scrollbar.HORIZONTAL ____Creates a horizontal scrollbar.

Scrollbar(int style, int initial value, int thumbsize, int min, int max)

This constructor will create a scrollbar with some additional information such as style, initial value, thumbsize, and minimum and maximum values of the scrollbar.

Methods of the Scrollbar Component

setValues (int initialvalue, int thumbsize, int min, int max)____Helps the developer to access additional information on the already created scrollbar.

int getMinimum()____Returns the minimum value of the particular scrollbar.

int getMaximum()____Returns the maximum value of the particular scrollbar.

int getValue()____Returns the current value of the particular scrollbar.

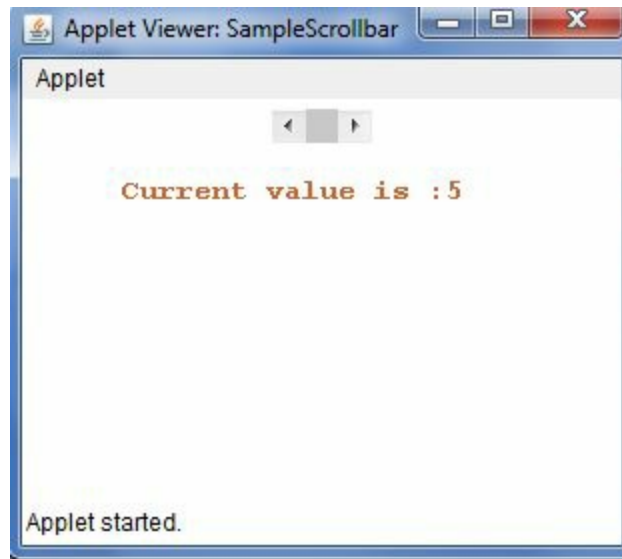
void setUnitIncrement(int newincr)____Increments or moves the thumb by the mentioned value. By default, if you click the scrollbar maximum it will increment the current value by one.

void setBlockIncrement(int newIncr)____Used to set the block increment. The default value of the block increment is ten.

If you make any changes on the scrollbar, it will generate the Adjustment event, which is handled by the Adjustment Listener interface. It has a method called by the name “getAdjustmentValueChanged (AdjustmentEvent aje).” This method will perform any action on the scrollbar. The following example will help you to understand the signature of the adjustment value changed method.

```
void adjustmentValueChanged(AdjustmentEvent obj)
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class SampleScrollbar extends Applet implements AdjustmentListener
{
    Scrollbar hs1;
    Font f1;
    Color c1;
    public void init()
    {
        f1=new Font("Courier",Font.BOLD,15);
        c1=new Color(200,100,50);
        hs1=new Scrollbar(Scrollbar.HORIZONTAL,5,1,0,255);
        add(hs1);
        hs1.addAdjustmentListener(this);
    }
    public void adjustmentValueChanged(AdjustmentEvent ae)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(c1);
        g.setFont(f1);
        text="Current value is :"+hs1.getValue();
        g.drawString(text,50,50);
    }
}
//<applet code="SampleScrollbar" height="200" width="300"></applet>
```



In this sample, we have demonstrated the usage and functionality of the scrollbar. We have created only one scrollbar with help of scrollbar constructor. Then we have created the font and color object to set the new font and color of the applet window. After successful creation of the horizontal scrollbar (having specified `Scrollbar.Horizontal` inside the `Scrollbar` constructor), it will be added to the applet by the `add()` method. Then, the adjustment listener event will perform the adjustment event. If you adjust the scrollbar immediately the adjustment event has been generated, it will call the adjustment value changed method. There we simply call the paint method by the repaint method to update the applet window. Inside the paint, we have set the color and font of the applet window by using the corresponding `setFont` and `setColor` methods. Finally, the `drawstring` method prints the current value of the scrollbar.

Menu

Basically all GUI applications have a menu and a menu bar. Generally, these types of menu items can perform any task if you click them. We have a set of methods to handle these menu events. If you click a menu item, it will generate an Action Event. We have already discussed event handling; the Action event is handled by the Action Listener Interface. Let's see the classes and their constructors and methods.

The `MenuBar` class is used to create the menu bar. This menu bar contains a

menu and menu items.

Frame.setMenuBar(MenuBar object);

This method will set the menu bar on the frame. Menu class is used to create a new Menu. It has a constructor that is used to create a new menu that can be added to the menu bar using the MenuBar.add (Menu item) method. The Menu Item class is used to create a new menu item. It has a constructor that will create a new menu item that can be added to the menu by the same add(MenuItem obj) method.

Sample Menu With Items

```
MenuBar mb = new MenuBar();  
Menu mFile = new Menu("File");  
MenuItem mNew = new MenuItem("New");  
mFile.add(mNew);  
mb.add(mFile)  
f.setMenuBar(mb);
```

Methods of the Menu Class

add(Menu)____Adds the specified menu item to the menu.

add(String)____Adds the specified string to the menu. A new menu item with the specified string as the label is created and added to the menu.

addSeparator()____Adds a separator to the menu.

getItem(int index)____Returns the item at the specified index as a string.

remove(int index)____Deletes the item at the specified index from the menu.

Menu Item Class Methods

getLabel()____Returns the label of the menu item in a string format.

setLabel(String)___Changes the label of the menu item to the specified string.

setEnabled(boolean)___Makes the menu item selectable/deselectable depending on whether the parameter is true or false.

Checkbox Menu Item

This class will create the checkbox menu item, which will show a checkmark with the menu item. This menu item can also perform some tasks, just like other menu items.

Checkbox Menu Item Methods

getState()___Returns the state of the menu item as a Boolean value. A value of true implies that the checkbox menu item is selected.

setState(boolean)___Sets the state of the menu item. If the value passed is true, the menu item is set to the selected state.

Sample Program for Creating a Menu

```
import java.awt.*;
import java.awt.event.*;
public class SampleMenu extends Frame implements ActionListener
{
    MenuBar mb;
    Menu mFile,mEdit,mFormat;
    MenuItem mNew,mOpen,mSave,mExit;
    MenuItem mCut,mCopy,mPaste;
    MenuItem mWord,mFon;
    public SampleMenu(String title)
    {
        super(title);
        mb = new MenuBar();
        mFile = new Menu("File");
        mEdit = new Menu("Edit");
```

```
mFormat=new Menu("Format");
mNew = new MenuItem("New");
mOpen = new MenuItem("Open");
mSave = new MenuItem("Save");
mExit = new MenuItem("Exit");
mCut = new MenuItem("Cut");
mCopy = new MenuItem("Copy");
mPaste = new MenuItem("Paste");
mWord=new MenuItem("Word wrap");
mFont=new MenuItem("Font");
mEdit.add(mCut);
mEdit.add(mCopy);
mEdit.add(mPaste);
mFile.add(mNew);
mFile.add(mOpen);
mFile.add(mSave);
mFile.addSeparator();
mFile.add(mExit);
mFormat.add(mWord);
mFormat.add(mFont);
mb.add(mFile);
mb.add(mEdit);
mb.add(mFormat);
setMenuBar(mb);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we)
    {
        dispose();
        System.exit(0);
    }
});
mExit.addActionListener(this);
mNew.addActionListener(this);
setSize(400,400);
setVisible(true);
}
public void actionPerformed(ActionEvent ae)
```

```

{
if(ae.getSource() == mExit)
{
dispose();
System.exit(0);
}
else if(ae.getSource() == mNew)
{
System.out.println("A new menu item selected");
}
}
public static void main(String args[])
{
SampleMenu sm = new SampleMenu("Menu Creation");
}
}

```

In this sample program, we have created a new menu bar with object mb, as well as the three menus File, Edit, and Format. The File menu has four menu items: new, open, save, and exit. The Edit menu has three menu items: cut, copy, and paste. The third and final menu is the Format menu, which has two menu items: Font and Word Wrap. Then we have added those options with the corresponding menus to our already created menu bar. setMenubar() will set the menu bar with the frame. Currently, the “exit” and “new” options only perform an action because we added ActionListener(). If you select “new,” System.out.println() will print “A new option selected.” If you click the exit option, the program will be terminated.

Dialog Box

Dialog is also a container in Java. We can store a set of components in the dialog box, but the dialog box cannot move freely because it depends on another container such as Frame. Basically, Dialog is the window that gets data from the user to perform any assigned task. We have two different types of dialog box: predefined and user defined. In Java, we call the predefined version a File Dialog box because it is associated with file operations. User defined dialog boxes are created by the user. Dialog class will help us to

create a user defined dialog box.

Constructors of the Dialog Class

`Dialog(Frame,boolean)`____Creates a new dialog that is initially invisible. In this constructor, we have two parameters. The first one, frame object, refers to the parent window for this newly created dialog. The second, Boolean data, refers to whether it is a modal or modeless dialog.

`Dialog(Frame,String,boolean)`____Creates a new dialog that is initially invisible. In this constructor, we have three parameters. The first one, frame object, refers to the parent window for this newly created dialog. The second, String data, is the title text of the dialog box. The third, Boolean data, refers to whether it is a modal or modeless dialog.

Modal and Modeless Dialog Boxes

Modal dialog will never allow the parent window to activate until this dialog closes, but in modeless dialog you can do anything in the parent window.

Methods of the Dialog Class

`boolean isModal()`____Returns true if the dialog is modal. We have already seen the working style of the modal dialog box. It will return false if the dialog is modeless.

`setResizable(boolean)`____Sets the resizable flag of the dialog box. If you set this value as true, it will allow us to resize the dialog box. If you set it as false, it cannot be resized.

`boolean isResizable()`____Used to check if the dialog is resizable or not. If it returns true, the dialog is resizable.

Sample Program for Dialog Box

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

```

public class Dialog1 extends Dialog implements ActionListener
{
    public Dialog1(Frame fm,String msg)
    {
        super(fm,msg,false);
        setLayout(new GridLayout(2,1,0,0));
        Panel p1 = new Panel();
        Panel p2 = new Panel();
        p1.setLayout(new FlowLayout());
        p1.setFont(new Font("Courier",Font.BOLD,20));
        p2.setLayout(new FlowLayout());
        p1.add(new Label(msg));
        Button b1,b2;
        b1 = new Button("Ok");
        b2 = new Button("Cancel");
        p2.add(b1);
        p2.add(b2);
        b1.addActionListener(this);
        add(p1);
        add(p2);
        setSize(250,150);
        setResizable(false);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we)
            {
                dispose();
            }
        });
    }
    public void actionPerformed(ActionEvent ae)
    {
        dispose();
    }
}

public class SampleDialog extends Frame implements ActionListener
{
    MenuBar mb;

```

```

Menu myMenu;
MenuItem mDialog,mExit;
public SampleDialog(String text)
{
    super(text);
    mb = new MenuBar();
    mMenu = new Menu("My Menu");
    mDialog = new MenuItem("Open Dialog");
    mExit = new MenuItem("Exit");
    mMenu.add(mDialog);
    mMenu.add(mExit);
    mb.add(mMenu);
    setMenuBar(mb);
    setSize(400,400);
    setVisible(true);
    mDialog.addActionListener(this);
    mExit.addActionListener(this);
    addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we)
    {
        dispose();
        System.exit(0);
    }
    });
}
public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource() == mDialog)
    {
        Dialog1 d1 = new Dialog1(this,"My New Dialog");
        mb.setVisible(true);
    }
    else if(ae.getSource() == mExit)
    {
        dispose();
        System.exit(0);
    }
}

```

```

}
public static void main(String args[])
{
SampleDialog sd = new SampleDialog("Dialog Demo");
}
}

```

In this program, we have used the Dialog class to create a new dialog box with two panels. The first panel has been set as a FlowLayout so all the controls are added to the center with five pixels of horizontal and vertical space. The first panel contains only one Label box. The second panel, also in FlowLayout, has two buttons. The first button is labeled “ok” and the second one is labeled “cancel.” If you click any button on this dialog, the action event will immediately generate.

In the sample Dialog program, we have created a menu with the name My Menu. Inside that menu we have two options. The first option is Open Dialog, which will open the dialog box. The second option is Exit, which will close the opened window.

File Dialog

We have already mentioned the file dialog box in previous sections. In Java, we have two important dialog boxes associated with file operations: Open Dialog and Save Dialog. We can open this dialog box by using the File Dialog constructor. We have three constructors in the FileDialog class.

Constructors of the FileDialog

FileDialog (Frame parent)____Creates a filedialog for loading a file. Note: Default file dialog is open, so if you use this constructor to create the file dialog, it will load an open Dialog.

FileDialog (Frame parent,String title)____Creates a file dialog for loading a file. Here we have two parameters: parent window object and the title of the newly created File dialog.

FileDialog (Frame parent, String title, int mode)____Similar to previous constructors, but here we have three parameters. The first two parameters are the same as previous constructors, but the third one is used to specify what kind of dialog you want to open.

Methods of the File Dialog Class

getDirectory()____Reads the directory of the corresponding file dialog.

getFile()____Reads the file of the corresponding file dialog.

getFilenameFilter()____Specifies the file dialog's filename filter.

getMode()____Returns the mode of the current dialog.

setDirectory(String)____Sets the directory of the file dialog window to the one specified.

setFile(String file)____Sets the selected file for the corresponding file dialog window to the one specified.

setFilenameFilter(filenamefilter filter)____Sets the file name of the filter to the file dialog.

setMode(int mode)____Sets the mode of the file dialog.

In the following section, we will see a sample program that uses the file dialog. This program will create a basic text editor like Notepad with minimum functionality. Here we will use the File dialog to load and save the file content of the text area. Let's see the code behind our new Java Notepad application using the user interface that we have already seen in our example program for the Menu.

Java Notepad App

```
import java.awt.*;  
import java.awt.event.*;  
import java.io.*;
```

```

public class Java_Notepad extends Frame implements ActionListener
{
MenuBar mb;
Menu mFile;
MenuItem mNew,mOpen,mSave,mExit;
TextArea ta;
public Java_Notepad(String text)
{
super(title);
setLayout(new FlowLayout());
mb = new MenuBar();
mFile = new Menu("File");
mNew = new MenuItem("New");
mOpen = new MenuItem("Open");
mSave = new MenuItem("Save");
mExit = new MenuItem("Exit");
mFile.add(mNew);
mFile.add(mOpen);
mFile.add(mSave);
mFile.addSeparator();
mFile.add(mExit);
mb.add(mFile);
setMenuBar(mb);
setSize(400,400);
setVisible(true);
ta = new TextArea(2000);
add(ta);
mNew.addActionListener(this);
mOpen.addActionListener(this);
mSave.addActionListener(this);
mExit.addActionListener(this);
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent we)
{
dispose();
System.exit(0);
}
}
}

```

```

});
}
public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource() == mNew)
    {
        ta.setText("");
    }
    else if(ae.getSource() == mOpen)
    {
        FileDialog fd =new FileDialog(this,"Open Dialog",FileDialog.LOAD);
        fd.show();
        String fname = fd.getDirectory() + "/" + fd.getFile();
        try
        {
            FileInputStream fin = new FileInputStream(fname);
            byte b[] = new byte[fin.available()];
            fin.read(b);
            String data = new String(b,0,b.length);
            ta.setText(data);
            fin.close();
        }
        catch(Exception e) {}
    }
    else if(ae.getSource() == mSave)
    {
        FileDialog fd = new FileDialog(this,"Save Dialog",FileDialog.SAVE);
        fd.show();
        String fname = fd.getDirectory() + "/" + fd.getFile();
        String data = ta.getText();
        byte b[] = new byte[data.length()];
        data.getBytes(0,data.length(),b,0);
        try
        {
            FileOutputStream fo = new FileOutputStream(fname);
            fo.write(b);
            fo.close();
        }
    }
}

```

```

} catch (Exception e) {}
}
else if (ae.getSource() == exit)
{
dispose();
System.exit(0);
}
}
public static void main(String args[])
{
Java_Notepad jn = new Java_Notepad("Untitled-Notepad");
}
}

```

In this Java notepad application, we have created a basic text editor with simple functionalities such as new, open, save, and exit. This should tell you everything you need to know about creating menus with text area components. If you have any confusion, check out our sample program in the menu creation section. All menu items will generate an action event, so you need to implement the action listener interface to handle the action event generated by each option. If you select a new option, an action event will immediately generate and call the action performed method. There we check what option has been selected. If you click the “new” option, it will erase all content inside the text area. If you select the “open” option, it will open a load dialog where you can select any file, which will immediately load in the text area. If you select the “option” option, it will open a save dialog box where you can save the text written in the text area. The final option is “exit,” which will stop the execution of our notepad application.

JFC Swing

Java Swing is a fully featured user interface development kit used to create interfaces in Java applications. The toolkit includes a vast variety of widgets and tools used to help create graphical user interfaces. It has built-in controls such as buttons, layout managers, sliders, text fields, labels, etc. It also has some advanced controls such as tree, table, scroll pane, and tab controls.

In this guide, you will learn the foundations of how to develop an interface in Java Swing. This should be enough for you to get started, but the rest is up to you. Remember, the Java API is at your disposal.

Let's start by going over a few objects that will be important to remember and understand.

Before we begin, it is important to remember the following three packages in Swing:

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

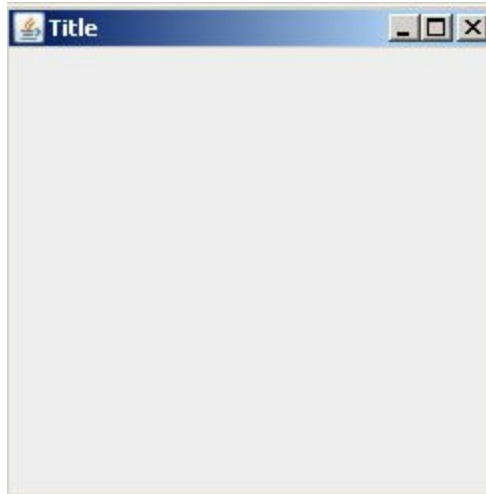
`javax.swing` is the package that has all required classes of swing application development. `JApplet`, `JButton`, `JTree`, and `JScrollPane` are some example classes of the swing package.

JFrame

A `JFrame` object is simply an object that allows a window to appear on the screen. You can think of a `JFrame` as a canvas, the base foundation that is needed in order to add elements to the interface. These elements are called “components” in Swing and will be referred to as such for the remainder of this guide.

To declare a `JFrame` object, you can do something as simple as:

```
JFrame frame = new JFrame("Title");  
frame.setSize(250,250);  
frame.setDefaultCloseOperation(frame. EXIT_ON_CLOSE );  
frame.setVisible(true);
```



Let's analyze every line in the sample code above.

```
JFrame frame = new JFrame("Title");
```

In this line, the JFrame object is being declared. The arguments of the JFrame constructor take in a string, which is considered the title of the window. Another alternate route to this is to leave the constructor empty and then use the method:

```
frame.setTitle("Title");
```

Let's move on to the next line:

```
frame.setSize(250,250);
```

This line should be self-explanatory. There is a method within the JFrame object that is called "setSize" that takes in two values. The first value is the width of the JFrame and the second value is the height.

```
frame.setDefaultCloseOperation(frame. EXIT_ON_CLOSE );
```

This next line simply lets the window know that if the program is ended through a mouse click of the x button or through the console, it should exit the window instead of leaving it open.

```
frame.setVisible(true);
```

The most obvious line of them all, this line simply sets the visibility of the JFrame to true so the user is able to see the window.

JPanel

In the canvas of the interface (JFrame), we need a panel so that we can add components to it (buttons, labels, etc.). You can think of the JFrame as a clipboard and the JPanel as the paper, that extra layer that will allow you to draw and add different types of components to the interface. One thing to note is that components can be added to the raw canvas (JFrame). However, many complications can arise from doing this. That is why we should add a JPanel to the JFrame.

To declare a JPanel, you can do the following in conjunction with the previous sample code:

```
JFrame frame = new JFrame();
frame.setTitle("Title");
frame.setSize(250,250);           frame.setDefaultCloseOperation(frame.
EXIT_ON_CLOSE );
JPanel panel = new JPanel();// new line of code
frame.add(panel); // new line of code
frame.setVisible(true);
```

In every object within the Swing hierarchy of classes, there is a commonly used method called “add” that takes in any Swing component. A JPanel is considered a component and is thus added to the frame. Now the JPanel can add components to it by referring to the object name of the JPanel and by using the add method. The JFrame object and the JPanel object technically have identical methods, so you can use the same methods you have been using with JFrame on JPanel. Adding this piece of code won’t change anything in the program, because all a JPanel does is to act as a container for GUI components to be added.

JLabel

The JLabel object is simply a Swing component that can be added to the JPanel. This component simply acts as a text element in the JFrame window. In order to add a JLabel component to the main panel we created before, we simply do the following:

```
JFrame frame = new JFrame();
frame.setTitle("Title");
frame.setSize(250,250);
frame.setDefaultCloseOperation(frame.  
EXIT_ON_CLOSE );
JPanel panel = new JPanel();
frame.add(panel);
frame.setVisible(true);
JLabel lblText = new JLabel("This is Text");
panel.add(lblText);
```

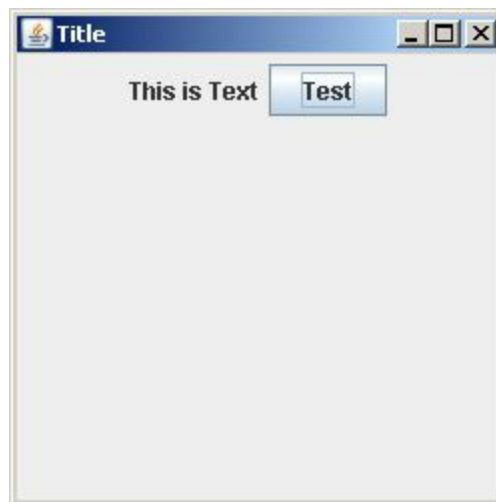


As shown above in the last two lines, you can see that it is simple to create a new JLabel object that takes in a string as its constructor and to use that to display the text. Another way to complete this task is to leave the constructor empty and use the setText method for the JLabel. The next line simply adds the JLabel object to the panel so it can be displayed. You must be wondering how you can have the program decide where and how to display the text. We will talk further about this in the section on LayoutManager.

JButton

The JButton object is also a Swing component that can be added to the JPanel. This component acts as a button element in the JFrame window. In order to add a JButton component to the panel, we simply do the following:

```
JFrame frame = new JFrame();
frame.setTitle("Title");
frame.setSize(250,250);
frame.setDefaultCloseOperation(frame.  
EXIT_ON_CLOSE );
JPanel panel = new JPanel();
frame.add(panel);
frame.setVisible(true);
JLabel lblText = new JLabel("This is Text");
panel.add(lblText);
JButton btnTest = new JButton("Test"); // new line of code
panel.add(btnTest); // new line of code
```



In the last two lines, we see how the JButton object takes in the value of a string as its constructor and is then used as the name of the button. Another way to do this is to leave the constructor empty and use the setText method for the JButton. Then, we add the JButton component to the JPanel. Are you seeing a trend here?

Layout Managers

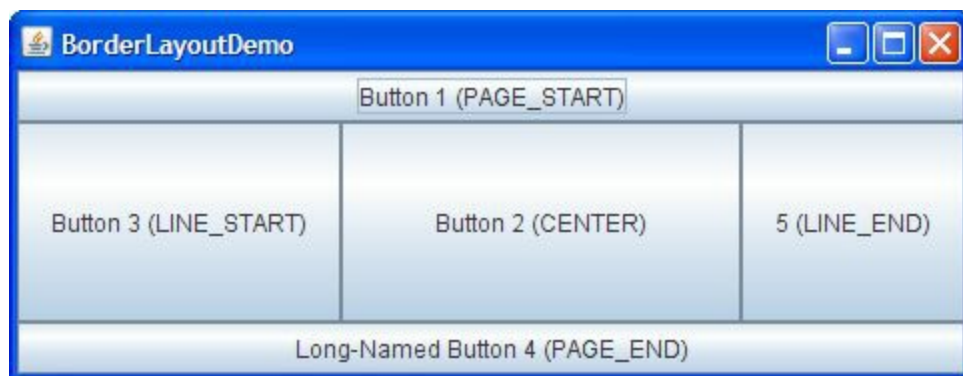
In this section, we will learn about layout managers and how to setup Swing components in the window to your satisfaction. There are many different types of layout managers in Java. The default layout is the `FlowLayout`, which we have been using. For the other layouts, you must specify the type of layout you want to use. If you do not specify a specific layout, then the `FlowLayout` will be used by default. This guide is going to get too long if we go over all the layouts individually, so instead we'll briefly show you some visuals of the different types of layouts so you can decide which ones you'd like to explore. Before we go over the list of layouts, let's go over how you can set one up. There are different ways to set up your layout depending on which type you choose, but there is one constant line you must always remember:

```
panel.setLayout(new GridLayout(10, 1));
```

In the line above, we have used the `setLayout` method, which will take a specific Layout object. For example purposes, we will input a Grid Layout into the layout parameter. This is how we can set a layout for different types of panels or frames within Swing.

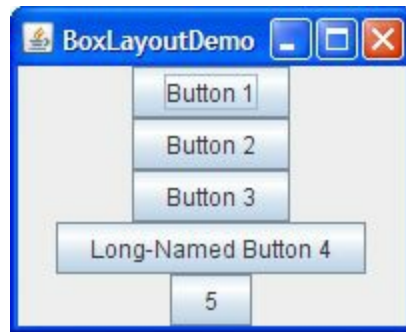
Now let's go over all the different types of layouts:

BorderLayout



The `BorderLayout` gives you the option to place components at the top, bottom, left, right, and center of the panel.

BoxLayout



The BoxLayout allows you to put components in a single row or column and allows the user to easily specify any custom maximum size. On top of that, it also allows you to align the components according to your needs.

CardLayout



The CardLayout allows you to implement an area that contains different components at different times.

FlowLayout



The FlowLayout is the default layout manager and does not need to be specified for use. All it does is lay out components in a single row and start a

new row depending on the width of the window.

GridBagLayout



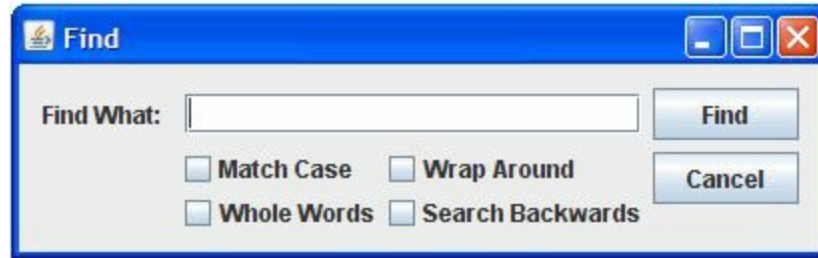
The GridBagLayout is the most flexible layout manager of all; it allows you to easily align and place components within a grid of cells.

GridLayout



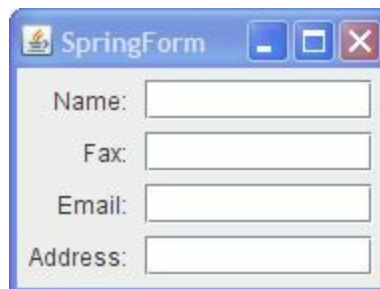
The GridLayout makes all components equal in size so they can be displayed sequentially within a grid, depending on the order of how the elements are being added—sort of like a FlowLayout except organized into rows and cells that you have specified.

GroupLayout



The GroupLayout allows you to work with separate horizontal and vertical layouts that are independently defined for each dimension.

SpringLayout



The SpringLayout is another flexible layout manager that allows you to specify the distance of elements from other elements, such as defining the left edge of a component relative to another component's right edge.

Chapter 18

Event Handling

The idea of Event Handling is that GUI elements will interact with each other depending on what they are asked to do in certain situations. In this part, we will look at handling the event of a user clicking a button in a Java UI. In order to set up a button, as with the JButton component, we add an ActionListener. A “Listener” is simply a function within a GUI element that will wait (or “Listen”) for something before reacting accordingly. When adding an action listener to a GUI component, it must reference a class that implements the ActionListener interface. Once the interface has been implemented, you will have to implement the methods part of that interface accordingly. Using all of our knowledge about GUIs, we are able to create a basic calculator using some Event Handling to listen for a mouse click on a JButton.

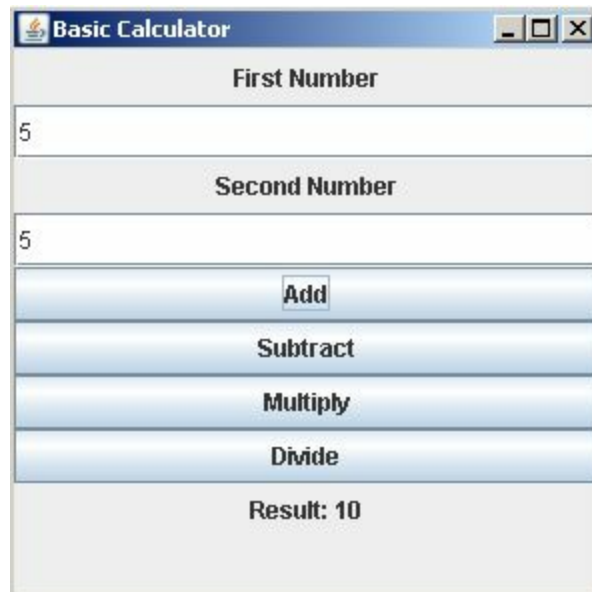
```
public class ThisIsAClass {
    public static void main (String[] args) {
        GUI g = new GUI();
    }
}
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class GUI extends JFrame implements ActionListener{
    JPanel mainPanel;
    JLabel lblfNum;
    JLabel lblsNum;
    JLabel lblResult;
    JButton btnAdd;
    JButton btnSubtract;
    JButton btnMultiply;
    JButton btnDivide;
    JTextField txt_fNum;
    JTextField txt_sNum;
    public GUI() {
        setDefaultCloseOperation( EXIT_ON_CLOSE );
```

```

setSize(300, 300);
setTitle("Basic Calculator");
setVisible(true);
Initialize();
}
private void Initialize() {
    mainPanel = new JPanel();
    mainPanel.setLayout(new GridLayout(10, 1));
    lblfNum = new JLabel("First Number", JLabel. CENTER );
    lblsNum = new JLabel("Second Number", JLabel. CENTER );
    lblResult = new JLabel("---", JLabel. CENTER );
    txt_fNum = new JTextField();
    txt_sNum = new JTextField();
    btnAdd = new JButton("Add");
    btnSubtract = new JButton("Subtract");
    btnMultiply = new JButton("Multiply");
    btnDivide = new JButton("Divide");
    SetListeners();
    mainPanel.add(lblfNum);
    mainPanel.add(txt_fNum);
    mainPanel.add(lblsNum);
    mainPanel.add(txt_sNum);
    mainPanel.add(btnAdd);
    mainPanel.add(btnSubtract);
    mainPanel.add(btnMultiply);
    mainPanel.add(btnDivide);
    mainPanel.add(lblResult);
    add(mainPanel);
    revalidate();
}
private void SetListeners() {
    btnAdd.addActionListener(this);
    btnSubtract.addActionListener(this);
    btnMultiply.addActionListener(this);
    btnDivide.addActionListener(this);
}
public void actionPerformed(ActionEvent e) {

```

```
String btnName = e.getActionCommand();
try{
int fNum = Integer.parseInt(txt_fNum.getText().trim());
int sNum = Integer.parseInt(txt_sNum.getText().trim());
if (btnName.equals("Add")) {
int result = fNum + sNum;
lblResult.setText(result + "");
}
else if (btnName.equals("Subtract")) {
int result = fNum - sNum;
lblResult.setText(result + "");
}
else if (btnName.equals("Multiply")) {
int result = fNum * sNum;
lblResult.setText(result + "");
}
else if (btnName.equals("Divide")) {
int result = fNum / sNum;
lblResult.setText(result + "");
}
}catch(Exception exc) {
lblResult.setText("Invalid input!");
}
}
}
```



In the main class, we have declared the GUI object that is automatically initializing the UI within the constructor. The GUI class is extending to JFrame, so we do not need to refer to the JFrame object when referring to its methods. We can go straight to referring to the methods, which is why the GUI class is organized that way.

```
setDefaultCloseOperation ( EXIT_ON_CLOSE );  
setSize(300, 300);  
setTitle("Basic Calculator");  
setVisible(true);  
Initialize();
```

The constructor calls the “Initialize” method, which contains the following:

```
mainPanel = new JPanel();  
mainPanel.setLayout(new GridLayout(10, 1));  
  
lblfNum = new JLabel("First Number", JLabel.CENTER);  
lblsNum = new JLabel("Second Number", JLabel.CENTER);  
lblResult = new JLabel("---", JLabel.CENTER);  
  
txt_fNum = new JTextField();  
txt_sNum = new JTextField();
```

```
btnAdd = new JButton("Add");  
btnSubtract = new JButton("Subtract");  
btnMultiply = new JButton("Multiply");  
btnDivide = new JButton("Divide");
```

```
SetListeners();
```

```
mainPanel.add(lblfNum);  
mainPanel.add(txt_fNum);  
mainPanel.add(lblsNum);  
mainPanel.add(txt_sNum);  
mainPanel.add(btnAdd);  
mainPanel.add(btnSubtract);  
mainPanel.add(btnMultiply);  
mainPanel.add(btnDivide);  
mainPanel.add(lblResult);
```

```
add(mainPanel);  
revalidate();
```

In the first line:

```
mainPanel = new JPanel();
```

We are declaring a JPanel object that then sets its layout as a GridLayout.

```
mainPanel.setLayout(new GridLayout(10, 1));
```

The two arguments are rows and columns. There are nine rows in the program, but to be safe, we input ten rows. The calculator is only one column and therefore only the number one has been inputted into the second argument.

```
lblfNum = new JLabel("First Number", JLabel.CENTER);  
lblsNum = new JLabel("Second Number", JLabel.CENTER);
```

```
lblResult = new JLabel("---", JLabel.CENTER);
```

```
txt_fNum = new JTextField();
```

```
txt_sNum = new JTextField();
```

```
btnAdd = new JButton("Add");
```

```
btnSubtract = new JButton("Subtract");
```

```
btnMultiply = new JButton("Multiply");
```

```
btnDivide = new JButton("Divide");
```

This is a conventional setup to initialize the Swing components being added to the panel, which is added to the JFrame.

The next line then calls upon a method:

```
SetListeners();
```

This method contains the following:

```
btnAdd.addActionListener(this);
```

```
btnSubtract.addActionListener(this);
```

```
btnMultiply.addActionListener(this);
```

```
btnDivide.addActionListener(this);
```

In this method, each JButton is having an ActionListener added to it. The argument within the parameter is the location of which ActionListener to refer to. Since the current class (GUI) implements the ActionListener interface, it is able to refer to the current class using the keyword `this`. The ActionListener interface implements the method “actionPerformed” and is then implemented. The method contains the following:

```
public void actionPerformed(ActionEvent e) {  
    String btnName = e.getActionCommand();  
    try{  
        int fNum = Integer.parseInt(txt_fNum.getText().trim());  
        int sNum = Integer.parseInt(txt_sNum.getText().trim());  
        if (btnName.equals("Add")) {
```

```

int result = fNum + sNum;
lblResult.setText("Result: " + result + "");
}else if (btnName.equals("Subtract")) {
int result = fNum - sNum;
lblResult.setText("Result: " + result + "");
}else if (btnName.equals("Multiply")) {
int result = fNum * sNum;
lblResult.setText("Result: " + result + "");
}else if (btnName.equals("Divide")) {
int result = fNum / sNum;
lblResult.setText("Result: " + result + "");
}
}catch(Exception exc) {
lblResult.setText("Invalid input!");
}
}

```

In this method, a try-catch statement is being used to detect whether or not the text field is going to contain a string, so that the Java application doesn't try to calculate text. The code above should be fairly self-explanatory. The variable `ActionEvent e` is given information when a Swing component is interacted with; once it has been interacted with, this method is called and `e` is given info. The `getActionCommand()` method returns the name of the button that has been interacted with. Next, a series of if-else statements are created to determine which button it was, and the appropriate operation is carried out.

Right after the `SetListeners()` method, all the Swing components that were initialized are added accordingly:

```

mainPanel.add(lblfNum);
mainPanel.add(txt_fNum);
mainPanel.add(lblsNum);
mainPanel.add(txt_sNum);
mainPanel.add(btnAdd);
mainPanel.add(btnSubtract);
mainPanel.add(btnMultiply);
mainPanel.add(btnDivide);

```



```
mainPanel.add(lblResult);
```

```
add(mainPanel);
```

```
revalidate();
```

Everything here should be self-explanatory except for the last line, which simply revalidates or updates all the Swing components if they haven't already been updated. It is essentially a safeguard that ensures all Swing components are visible and updated.

Chapter 19

JDBC Programming

Basically all software has two “ends”: Frontend and Backend. Frontend means the user interface of the Application and backend means the database. Frontends are created through the programming language. For example, if you create a program with the help of Java, VB, C#, etc. that program will be considered Frontend. Backend is used to store data. If you want to see those data in your program, you will need to connect your backend to the frontend. In Java, JDBC (Java Database Connectivity) plays a major role in connecting frontend and backend.

In the following section, we have described how JDBC connects with the program.

Java Program-----JDBC-----ODBC-----Database

A Java program starts its connecting process with the JDBC, then it connects with ODBC. The ODBC reads the data from database. Finally, the data are displayed in the user interface.

We have a set of classes to create the connection between frontend and backend, contained in the Java.sql package. In the following section, we will see the important classes and methods involved in connection creation. Before starting the JDBC, you need to know SQL concepts because it is necessary to know the SQL query to handle the database data. The following section will briefly explain the task.

Important Queries in SQL

Database Creation

```
create database database_name;
```

This query will create a new database.

```
Create table table_name(field_name1 datatype, field_name2 datatype.....);
```

This query will create a new table with some fields.

```
Insert into table_name values('data1',data2,data3.....);
```

This query will add the new record to the table.

```
Alter table table_name add/drop column_name datatype;
```

This query will alter the field or field data type, or will delete the field.

```
Update table_name set column_name="value" where condition;
```

This query will update a particular column.

```
Select * from table_name;
```

This query will show all records inserted in the table.

Sample Query

```
create database db1;  
create table student(Sname varchar(25),Sno int,age int);  
insert into student values('Mathew',6001,23);  
insert into student values('Antony',6002,23);  
insert into student values('William',6003,23);  
select * from student;
```

Important Classes in java.sql

Connection Contains some important methods that are used to establish the connection.

Statement Contains some methods that are used to run some SQL queries.

ResultSet Contains some methods and holds the records retrieved from the table.

Steps Involved in Connection Creation

Step 1: Initialize the driver file

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Step 2: Establish a connection with the database.

Step 3: Query the database.

Methods Involved in Database

Connection createStatement()____Initializes the statement object.

Statement executeUpdate()____Runs queries like alter, insert, create, drop, update, and delete.

executeQuery()____Runs the selected statement.

Sample Program 1

```
import java.sql.*;
class SampleTable
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection("jdbc:odbc:dsn1","","");
            String qry = "Create Table Student(sno int,sname varchar(20))";
            Statement st = con.createStatement();
            st.executeUpdate(qry);
            System.out.println("Table has been created");
        }
        catch(Exception e)
        {
            System.out.println("Exception : " + e);
        }
    }
}
```

In this program, we have created a new table. First we have created a connection object by using the `getConnection()` method. There are three different available parameters. The first one is the jdbc and odbc driver along with the data source name. The second one is the user name of the database application, and the third one is the password of the database application. On the next line, we have stored a table creation query in a string variable. Then we have created a statement object in order to run the SQL query. Finally, we have passed the string query as a parameter. It will execute the query and create the table. Once the table has been created, statement confirms the creation of the table. You can check the created table in the database you have selected.

Inserting the Data

```
import java.io.*;
import java.sql.*;

class InsertionSample
{
    public static void main(String args[])
    {
        int sno;
        String sname;
        String wish="y";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection("jdbc:odbc:dsn1","","");
            BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
            do{
                System.out.print("Enter sno and sname : ");
                sno = Integer.parseInt(br.readLine());
                sname = br.readLine();
                String qry = "Insert into Student values(" + sno + "," + sname + ")";
                Statement st = con.createStatement();
                int r = st.executeUpdate(qry);
                System.out.println(r + " row(s) Inserted");
            }
```

```

System.out.print("Add Another Record (y/n)?");
wish = br.readLine();
} while(wish.equals("y"));
}
catch(Exception e)
{
System.out.println("Exception : " + e);
}
}
}

```

This program will insert the data into the table we have created in the last sample program. First we have created a connection object using the getConnection() method. On the next line we have stored a table creation query in a string variable. Then we have created a statement object in order to run the SQL query. Finally, we have passed the string query as a parameter. It will execute the query and a command prompt system will wait for the user to enter the data. First you will need to enter the student number, followed by the name of the student. Finally, data will be inserted and display the message “1 row(s) inserted.” This statement confirms that the process has been successfully completed. Then the program asks the user to either enter another datum or to leave. If you enter “y,” this process starts again; if you press “n,” the program will terminate.

Displaying Table Data

```

import java.sql.*;
class SelectionSample
{
public static void main(String args[])
{
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con =
DriverManager.getConnection("jdbc:odbc:dsn1","", "");
Statement st = con.createStatement();
String qry = "Select * from student";

```

```

ResultSet rs = st.executeQuery(qry);
while(rs.next())
{
System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
} catch(Exception e)
{
System.out.println("Exception : " + e);
}
}
}

```

This program will show the data from the table we have created in the first program. First we have created a connection object by using the `getConnection()` method. On the next line we have stored a table creation query in a string variable. Then we have created a Statement object in order to run the SQL query. Now we needed to create an object for Resultset to store the data. Finally, we have passed the string query as a parameter. It will execute the query and store the data into the result set object. Then we can read those data by result set methods. The `getInt()` method will show the integer data and the `getString()` method will show the string data from the result object. The `rs.next()` method will move the record to next record. This process continues until the result set pointer reaches the end of the file.

Sample Program Using Frame

```

import java.sql.*;
import java.awt.*;
import java.awt.event.*;
public class DBstudent extends Frame implements ActionListener
{
Label sno,sname;
TextField sn,sna;
Button ins,can;
Connection con;
String msg = new String();
public DBstudent(String title)

```

```

{
super(title);
setLayout(new FlowLayout());
sno = new Label("Sno");
sname = new Label("Sname");
sn = new TextField(5);
sna = new TextField(20);
ins = new Button("Add Record");
can = new Button("Cancel");
add(sno);
add(sn);
add(sname);
add(sna);
add(ins);
add(can);
ins.addActionListener(this);
can.addActionListener(this);
addWindowListener(new MWA(this));
}
public void actionPerformed(ActionEvent ae)
{
int no;
String nam;
if(ae.getSource() == ins)
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con = DriverManager.getConnection("jdbc:odbc:mydsn","sa","");
no = Integer.parseInt(sn.getText());
nam = sna.getText();
String qry = "Insert into Student Values(" + no + "," + nam + ")";
Statement st = con.createStatement();
int r = st.executeUpdate(qry);
msg = r + " row(s) Inserted";
MessageBox mb = new MessageBox(msg);
mb.setVisible(true);

```



```

    } catch(Exception e)
    {
        System.out.println("Exception : " + e);
    }
}
else if(ae.getSource() == can)
{
    sn.setText("");
    sna.setText("");
}
}
public static void main(String args[])
{
    DBstudent s = new DBstudent("Student Data");
    s.setVisible(true);
    s.setSize(400,400);
}
}
class MWA extends WindowAdapter
{
    DBstudent s;
    public MWA(Student s)
    {
        this.s = s;
    }
    public void windowClosing(WindowEvent we)
    {
        try
        {
            s.dispose();
            System.exit(0);
        } catch(Exception e){}
    }
}
class MessageBox extends Frame implements ActionListener
{
    Label ms;

```

```

Button b1;
public MessageBox(String lab)
{
setLayout(new FlowLayout());
ms = new Label(lab);
b1 = new Button("ok");
add(ms);
add(b1);
setSize(200,200);
setLocation(100,100);
b1.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
try{
dispose();
}catch(Exception e){}
}
}

```

This program is similar to our InsertionSample Program. Here we are also inserting data into the same Student table through our GUI application. In this program you need to enter the data in the corresponding text field, then click the Add Record button to insert the record. It will show “1 row(s) inserted” in a message box. First we have created two text fields and two buttons that will help us to enter the input data and submit the entered data. You will need to enter the student name and number in the corresponding fields. After entering your data, you can click the “add record” button to insert the data into the database you have created and stored in your system. In the action performed method, we establish the connection and get the data from the text fields. Finally, all data will be inserted into the database table you have created. After inserting the data, an acknowledgment will display in the message box. We also have another button that is used to remove the entered data from the text field.

Chapter 20

Exception Handling

Every programming language will throw an error if you make any mistakes. We have two different types of errors in Java programming: syntax and runtime. Basically, syntax errors arise when we break the rules of the language, such as missing the semicolons or brackets, mismatching variable names, and so on. This type of error can easily be corrected, because these are all reported when you compile your program—hence they are also called “compile time errors.” On the other hand, some types of errors only arise after you start running your program. These are called “runtime errors,” also known as “exceptions.” There are two different types of exceptions: predefined and user defined. All predefined exceptions are defined with the help of the exception class.

When coding, you may land in situations where your code returns an exception when attempting to do something. For example, if an object requires an integer in its constructor but you feed it null instead, the compiler will return an error indicating that the object didn't receive the appropriate parameter. Exception class is the super class for all types of Exception class. Because each exception handled by specific Exception class. For example, a divide by zero exception is only handled by the ArithmeticException class. In the following section, we have listed some predefined exception classes and the exceptions they handle.

Predefined Exception Classes

ArithmeticException___Handles exceptions caused by math errors, e.g., divide by zero.

ArrayIndexOutOfBoundsException___Handles exceptions from bad array indices.

ArrayStoreException___Arises when a program tries to store the wrong type of data in an exception.

FileNotFoundException___Arises when we attempt to access a nonexistent

file.

`IOException`____Arises with general I/O failures such as inability to read from a file.

`NullPointerException`____Arises when we reference a null object.

`NumberFormatException`____Arises when a conversion between strings and numbers fails.

`OutOfMemoryException`____Arises when there's not enough memory to allocate a new object.

`SecurityException`____Arises when an applet tries to perform an action not allowed by the browser's security setting.

`StringIndexOutOfBoundsException`____Arises when a program attempts to access a nonexistent character position in a string.

`NegativeArraySizeException`____Arises when we declare the array size as negative.

Another example would be when you're working with file handling (which will be taught later) and you're trying to read a file in a directory that doesn't exist. The compiler would then return an exception such as "`FileNotFoundException`"—there are others, but this type of exception is found regularly in file handling.

In order to regulate exceptions in your code, you can use try-catch statements. What these statements do is check if a block of code returns an exception; if it does, instead of halting the whole operation of the program, it will catch the exception and try to either fix it or work around it, depending on what the programmer has coded.

We can handle this type of exception with help of following statements or blocks:

- Try catch finally block
- Nested Try block
- Multiple catch block
- Throw statements
- Throws statements

1. Try Catch Finally Block

```

Try
{
//Exception rising statements kept under this block . . . . .
// If any exceptions arise, they will be thrown to the catch block
. . . . .
. . . . .
}
catch(Exception_class object)
{
//This block handles the raised exceptions and smoothly terminates the
program.
}
finally
{
// this finally block always runs and doesn't care about the exception.
}

```

In the following example, you can see how the try catch finally block handled the raised exception.

The file “text.txt” does not exist.

```

Import java.io.*;
Import java.lang.*;
Class ExceptionDemo
{
Public static void main(String args[])
{
try {

```

```

ReadFile("text.txt");
}catch(Exception e) {
System.out.println("The file does not exist.");
}
Finally
{
System.out.println ("Check before you read the file if it exists or not");
}
}

```

The try statement does not require parameters, but the catch exception requires the parameter of an Exception object. When an exception does occur, the variable “e” of type Exception is given data, and the “e” variable can be manipulated to either print out the error or do various other things. A commonly used method for the e variable is e.printStackTrace();

2. Nested Try Block

This block contains a one try block inside another one try block. Following the syntax should help you to understand the nested try block. In a nested try block, each try block has its own catch block.

```

try {
// exception raising program statements . . . .
try
{
// exception raising program statements . . . . .
}
catch(Exception_class object)
{
// Exception handling statements. . . . . .
}
}
Catch(Exception_class object)
{
}

```

3. Multiple Catch Block

In a multiple catch block, one try block may contain more than one catch block so you can easily handle more than one exception. If an exception is raised, it is first sent to the first catch block; if it is not handled there, that exception will be sent to next catch block. This process continues until the last catch block. The following syntax should help you to understand the multiple catch block. Simply put, it works like a ladder if statement.

```
try
{
// exception arising statements goes here. . . . .
}
catch (exception_class obj)
{
// Exception handling statements . . . . .
}
catch (Exception_class obj) {
// Exception handling statements . . . . .
}
```

Example Program

```
class Exp1
{
public static void main(String args[])
{
int z;
try
{
DataInputStream br=new DataInputStream(System.in);
String a[]=new String[2];
System.out.println("Enter your data");
for(int i=0;i<=2;i++)
a[i]=br.readLine();
int x = Integer.parseInt(a[0]);
int y = Integer.parseInt(a[1]);
```

```

z = x/y;
System.out.println("c = " + c);
}catch(ArithmeticException e)
{
System.out.println("Arithmetic Exception Raised");
z=0;
} catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Invalid Arguments");
}
catch(NumberFormatException e)
{
System.out.println("String cannot be changed to Integer");
} finally
{
System.out.println("Reached Finally Block");
}
}
}

```

In this example, we have declared a string array size of 2, so we can store a maximum of two string data inside that string array variable. First of all, we are reading two string data at runtime with the help of the `readLine()` method, which is defined inside the `DataInputStream` class. So, the `br.readLine()` method will read the string data and then store it to the string array, where we can convert those data to integer data with the help of the Wrapper class method `Integer.parseInt(String data)`. If you enter zero, it will be stored into the `x` or `y`. Inside the try block, we are dividing `x` by `y`. If the `y` value is zero, it will throw an `ArithmeticException` that can be handled by first catch block. If you enter only one datum, an `ArrayIndexOutOfBoundsException` Exception will arise. If your data are characters, they cannot be converted by the Wrapper class method and so it will throw a `NumberFormatException`. If everything is fine, the mean dividend result will be displayed. Here we have used multiple catch blocks to handle more than one exception at a time; the process starts from the first catch block.

4. Throw Statement

This is used in cases of manual exceptions. It will throw an exception when we need it to. Basically, it helps when user defined exceptions arise because it doesn't arise automatically, so you will need to use a throw statement. Following is the syntax of a Throw statement.

```
Throw new Exception_class(".....");
```

Example

```
Throw new MyException();
```

Throws statements

These are particularly used to apply to whole methods. In your method, any part may send an exception, meaning you can throw those exceptions by using this exception-handling statement.

```
public void sample(int a) throws Exception
{
// Statements of the method . . . . .
}
```

Chapter 21

File Handling

File Handling is the idea of reading and writing external files in the machine on which the program is being coded. In Java, file handling isn't that complicated once a few basic ideas have been understood. In this guide, we will be going over some basic file handling fundamentals for reading text files.

Reading Files

In order to read files, you must use the “`FileReader`” class/object, which reads ordinary text files.

`FileReader` is a class or object that is used to read files, and should be utilized with the `BufferedReader` object, which is going to help save data when reading the text file because it will allow you to read the text file one line at a time. If you don't use `BufferedReader`, you will feel the pain of reading each character of the text file, which is less easy to do than reading it line by line.

The first thing you must do is import everything that has to do with Input and Output. The import directory for this is:

```
import java.io.*;
```

We will use the text file “test.txt” as an example.

The first thing you must do is declare your file name, which you can do either within the declaration of the object or within a string variable that can then later on be used within the constructor.

Example:

```
String fileName = “test.txt”; // this line is the name of the file to read
```

The next thing you must do is declare the `FileReader` object—but remember

to do all of this within a try-catch statement, because if you're trying to read a file that does not exist, it will throw an exception.

Example:

```
FileReader fileReader = new FileReader(fileName);  
OR  
FileReader fileReader = new FileReader("test.txt");
```

Once this object has been declared, you must wrap `FileReader` within `BufferedReader` like the following:

```
BufferedReader buffRead = new BufferedReader(fileReader);
```

The `BufferedReader` constructor takes in the value of the `FileReader` object, so you must feed the `FileReader` object into the constructor.

The next thing you must do is have a while loop that checks whether or not the current line that the `BufferedReader` is reading is a valid line that isn't null (doesn't exist). Every time `.readLine()` is called from the `BufferedReader` object, it reads the line and moves on to the next line for the next time that `.readLine()` is called. Therefore, the line must be stored in a string to conserve the line. While the "while" loop is going on, you can declare the line to the `buffRead.readLine()`.

Example:

```
String line = "";  
while ((line = buffRead.readLine()) != null) {  
    System.out.println(line);  
}
```

The above code shows a while loop going through the text file "test.txt" and checking if each line exists; if it does, it will print out accordingly. Once this has finished, close the file like such:

```
buffReader.close();
```

If any exception is thrown from reading a file and gets dropped down to the catch, then you can display an error.

Writing to Files

Writing to a text file in Java is similar to reading a text file, except this time you're outputting/writing to a text file. You must use the "FileWriter" class/object, which writes to ordinary text files.

FileWriter is a class or object that can be used to write to files and should be utilized with the BufferedOuputWriter object, which will help write text line by line.

A lot of the steps taken to write to a file are similar to reading a file, but we'll go over them again. The first thing you must do is import everything that has to do with Input and Output—the import directory for this is:

```
import java.io.*;
```

We will use the text file "test.txt" as an example.

The first thing you must do is declare your file name, which you can do either within the declaration of the object or within a string variable that can later be used within the constructor. Remember, when writing to a file, it will either overwrite a file with the same file name OR create a file with the file name specified.

Example:

```
String fileName = "test.txt"; // this line is the name of the file to write to or create
```

The next thing you must do is declare the FileWriter object, but remember to do all of this within a try-catch statement for any potential exceptions that can be thrown when trying to write to a file.

```
FileWriter fileWriter = new FileWriter(fileName);  
OR  
FileWriter fileWriter = new FileWriter("test.txt");
```

Once this object has been declared, you must wrap `FileWriter` within `BufferedOutputStream` like so:

```
BufferedOutputStream buffWrite = new BufferedOutputStream (fileWriter);
```

The `BufferedOutputStream` constructor takes in the value of the `FileWriter` object, so you must feed the `FileWriter` object as in the above sample code.

Now, in order to write to a file, you must do the following:

```
buffWrite.write("Hi, my name is ");  
buffWrite.write(" Tom.");
```

In the above example, we are writing to a text file, but only to the same line. In order to move on to the next line, you must use:

```
buffWrite.newLine();
```

Once you have done this, you are able to move on to the next line by simply typing:

```
buffWrite.write("This is the next line");
```

Finally, you must close your file by typing out the following line:

```
buffWrite.close();
```

Chapter 22

Example Programs

How about we test our knowledge by coding some example programs? For these, we will be using the 2014 Canadian Computing Competition. The problems are linked here:

<https://cemc.math.uwaterloo.ca/contests/computing/2014/stage%201/juniorEn>

Problem N°1

Problem J1: Triangle Times

Problem Description

You have trouble remembering which type of triangle is which.

You write a program to help.

Your program reads in three angles (in degrees).

- If all three angles are 60, output Equilateral.
- If the three angles add up to 180 and exactly two of the angles are the same, output Isosceles.
- If the three angles add up to 180 and no two angles are the same, output Scalene.
- If the three angles do not add up to 180, output Error.

Input Specification The input consists of three integers, each on a separate line.

Each integer will be greater than 0 and less than 180.

Output Specification Exactly one of Equilateral, Isosceles, Scalene or Error will be printed on one line.

Sample Input 1

60

70

50

Output for Sample Input 1
Scalene

Sample Input 2
60
75
55

Output for Sample Input 2
Error

Solution N°1

```
import java.util.Scanner;

public class J1 {
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        int a1 = Integer.parseInt(scan.nextLine());
        int a2 = Integer.parseInt(scan.nextLine());
        int a3 = Integer.parseInt(scan.nextLine());
        if(a1 > 0 && a2 > 0 && a3 > 0 && a1 < 180 && a2 < 180 && a3 < 180)
        {
            if(a1 == 60 && a2 == 60 && a3 == 60)
            {
                System.out.println("Equilateral");
            }
            else if((a1 + a2 + a3 == 180) && ((a1 == a2) || (a2 == a3) || (a3 == a1)))
            {
                System.out.println("Isosceles");
            }
            else if((a1 + a2 + a3 == 180) && ((a1 != a2) && (a2 != a3) && (a3 != a1)))
            {
                System.out.println("Scalene");
            }
        }
    }
}
```

```
}  
else  
{  
System.out.println("Error");  
}  
}  
}  
}
```

So what did we do there? First, we created a scanner. Then we got the three angles given to us. We then checked to see if the angle was valid. Next, we checked if the triangle had all angles equal to 60 degrees, meaning it was equilateral. If so, we printed that out. Otherwise, we checked if the angles added up to 180 to check if the triangle was valid, and we checked if any two angles were equal. This would prove that it was an isosceles triangle. If so, we printed it out. Otherwise, we checked if the triangle was valid and if all the sides were dissimilar, meaning it was scalene. If so, we printed it out. If none of those situations were true, we printed out “Error.”

Problem N°2

Problem J2:Vote Count

Problem Description

A vote is held after singer A and singer B compete in the final round of a singing competition.

Your job is to count the votes and determine the outcome.

Input Specification The input will be two lines. The first line will contain V ($1 \leq V \leq 15$), the total number of votes. The second line of input will be a sequence of V characters, each of which will be A or B, representing the votes for a particular singer.

Output Specification

The output will be one of three possibilities:

- A, if there are more A votes than B votes;
- B, if there are more B votes than A votes;
- Tie, if there are an equal number of A votes and B votes.

Sample Input 1

6

ABBABB

Output for Sample Input 1

B

Sample Input 2

6

ABBABA

Output for Sample Input 2

Tie

Solution N°2

```
import java.util.Scanner;
public class J2
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System. in );
        int chars = Integer. parseInt ((scan.nextLine()));
        String votes = scan.nextLine();
        if(chars > 0 && chars < 16)
        {
            int voteA = 0;
            int voteB = 0;
            for(int i = 0; i < chars; i++)
            {
                if(votes.charAt(i)=='A')
                {
                    voteA++;
                }
            }
        }
    }
}
```

```

}
else if(votes.charAt(i)=='B')
{
voteB++;
}
}
if(voteA == voteB)
{
System.out.println("Tie");
}
else if(voteA > voteB)
{
System.out.println("A");
}
else
{
System.out.println("B");
}
}
}
}

```

First, we created a scanner. We got the first line of input, which is the number of votes. We got the second line of input, which is the actual votes. Then we looped through each letter of the second line and if the letter was a capital A we added one to the voteA variable. If it was a capital B, we added one to the voteB variable. Finally, we checked whether voteA and voteB were equal; if so, it was a tie. If voteA is greater than voteB, A wins; otherwise, B wins.

Problem N°3

Problem J3: Double Dice

Problem Description

Antonia and David are playing a game. Each player starts with 100 points. The game uses standard six-sided dice and is played in rounds. During one

round, each player rolls one die. The player with the lower roll loses the number of points shown on the higher die. If both players roll the same number, no points are lost by either player. Write a program to determine the final scores.

Input Specification

The first line of input contains the integer n ($1 \leq n \leq 15$), which is the number of rounds that will be played. On each of the next n lines, will be two integers: the roll of Antonia for that round, followed by a space, followed by the roll of David for that round. Each roll will be an integer between 1 and 6 (inclusive).

Output Specification The output will consist of two lines. On the first line, output the number of points that Antonia has after all rounds have been played. On the second line, output the number of points that David has after all rounds have been played.

Sample Input

```
4
5 6
6 6
4 3
5 2
```

Output for Sample Input

```
94
91
```

Explanation of Output for Sample Input

After the first round, David wins, so Antonia loses 6 points. After the second round, there is a tie and no points are lost. After the third round, Antonia wins, so David loses 4 points. After the fourth round, Antonia wins, so David loses 5 points. In total, Antonia has lost 6 points and David has lost 9 points.

Solution N°3

```
import java.util.Scanner;
```

```

public class J3 {
public static void main(String[] args)
{
Scanner scan = new Scanner(System. in );
byte rounds = Byte. parseByte (scan.nextLine());
if(rounds > 0 && rounds < 16)
{
byte scoreA = 100;
byte scoreD = 100;
for(byte i = 0; i < rounds; i++)
{
byte rollA = scan.nextByte();
byte rollD = scan.nextByte();
if(rollA > 0 && rollA < 7 && rollD > 0 && rollD < 7)
{
if(rollA > rollD)
{
scoreD -= rollA;
}
else if(rollA < rollD)
{
scoreA -= rollD;
}
}
}
System. out .println(scoreA);
System. out .println(scoreD);
}
}
}

```

As usual, we have created a scanner. We made a variable to store the number of rounds and we stored the input. We made two variables: one to store Antonia's score, and one to store David's score. We then made a loop that runs for every round. We got Antonia's roll and David's roll and stored them in respective variables. We checked to see if each roll was within range. Then we checked if Antonia rolled higher; if so, we removed that roll from David's

score. Otherwise, we checked if David rolled higher; if so, we removed that roll from Antonia's score. We excluded a final else statement, because if neither is higher than the other, then the rolls are equal, meaning neither player should lose points.

Problem N°4

Problem J4: Party Invitation

Problem Description

You are hosting a party and do not have room to invite all of your friends. You use the following unemotional mathematical method to determine which friends to invite. Number your friends $1, 2, \dots, K$ and place them in a list in this order. Then perform m rounds. In each round, use a number to determine which friends to remove from the ordered list. The rounds will use numbers r_1, r_2, \dots, r_m . In round i remove all the remaining people in positions that are multiples of r_i (that is, $r_i, 2r_i, 3r_i, \dots$) The beginning of the list is position 1. Output the numbers of the friends that remain after this removal process.

Input Specification The first line of input contains the integer K ($1 \leq K \leq 100$). The second line of input contains the integer m ($1 \leq m \leq 10$), which is the number of rounds of removal. The next m lines each contain one integer. The i th of these lines ($1 \leq i \leq m$) contains r_i ($2 \leq r_i \leq 100$) indicating that every person at a position which is multiple of r_i should be removed.

Output Specification The output is the integers assigned to friends who were not removed. One integer is printed per line in increasing sorted order.

Sample Input

```
10
2
2
3
```

Output for Sample Input 1

3
7
9

Explanation of Output for Sample Input

Initially, our list of invitees is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. There will be two rounds of removals. After the first round of removals, we remove the even positions (i.e., every second position), which causes our list of invitees to be 1, 3, 5, 7, 9. After the second round of removals, we remove every 3rd remaining invitee: thus, we keep 1 and 3, remove 5 and keep 7 and 9, which leaves us with an invitee list of 1, 3, 7, 9.

Solution N°4

```
import java.util.ArrayList;
import java.util.Scanner;
public class J4 {
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in );
        ArrayList<Byte> friends = new ArrayList<Byte>();
        byte friendCount = Byte.parseByte (scan.nextLine());
        byte m = Byte.parseByte (scan.nextLine());
        if(friendCount > 0 && friendCount < 101 && m > 0 && m < 11)
        {
            for(byte i = 1; i < friendCount+1; i++)
            {
                friends.add(new Byte(i));
            }
            for(byte ind = 0; ind < m; ind++)
            {
                byte i = Byte.parseByte (scan.nextLine());
                if(i > 1 && i < 101)
                {
                    for(byte r = (byte)(i-1); r < friends.size(); r+=i-1)
                    {
```

```

friends.remove(r);
}
}
}
for(byte i = 0; i < friends.size(); i++)
{
System. out .println(friends.get(i).intValue());
}
}
}
}
}

```

We have created another scanner. We created an ArrayList of friends so that we can add and remove friends. Then we got the friend count, and the number of rounds of removal. Then we checked if these numbers were within range. Then, we looped through the number of friends, one-based as opposed to zero-based, and each time we added a friend's number to the ArrayList. Then we looped through the rounds of removal. In each iteration, we got the multiple to remove, checked if the multiple was within range, and if so, looped through the array (but only through the multiples), and removed the friend at that index. Finally, once all the rounds were finished, we printed out the remaining friend numbers from the ArrayList.

Problem N°5

Problem J5: Assigning Partners

Problem Description The CEMC is organizing a workshop with an activity involving pairs of students. They decided to assign partners ahead of time. You need to determine if they did this consistently. That is, whenever A is a partner of B, then B is also a partner of A, and no one is a partner of themselves.

Input Specification

The input consists of three lines. The first line consists of an integer N ($1 < N \leq 30$), which is the number of students in the class. The second line contains

the first names of the N students separated by single spaces. (Names contain only uppercase or lowercase letters, and no two students have the same first name). The third line contains the same N names in some order, separated by single spaces. The positions of the names in the last two lines indicate the assignment of partners: the ith name on the second line is the assigned partner of the ith name on the third line.

Output Specification

The output will be good if the two lists of names are arranged consistently, and bad if the arrangement of partners is not consistent.

Sample Input 1

4

Ada Alan Grace

John John Grace Alan Ada

Output for Sample Input 1

good

Explanation for Output for Sample Input 1

Ada and John are partners, and Alan and Grace are partners. This arrangement is consistent.

Sample Input 2

7

Rich Graeme Michelle Sandy Vlado Ron Jacob

Ron Vlado Sandy Michelle Rich Graeme Jacob

Output for Sample Input 2

bad

Explanation for Output for Sample Input 2

Graeme is partnered with Vlado, but Vlado is partnered with Rich. This is not consistent. It is also inconsistent because Jacob is partnered with himself.

Solution N°5

```
import java.util.Scanner;
import java.util.StringTokenizer;
public class J5
{
    Scanner scan = new Scanner(System. in );
    byte n;
    public J5()
    {
        n = Byte. parseByte (scan.nextLine());
        String[] partner1 = new String[n];
        String[] partner2 = new String[n];
        assignPartners(partner1, partner2);
        if(checkForSelfPartner(partner1, partner2) || checkForMultiUsage(partner1,
        partner2) || checkPartnersMatch(partner1, partner2))
        {
            System. out .println("bad");
        }
        else
        {
            System. out .println("good");
        }
    }
    public void assignPartners(String[] partner1, String[] partner2)
    {
        StringTokenizer partner1Names = new StringTokenizer(scan.nextLine());
        StringTokenizer partner2Names = new StringTokenizer(scan.nextLine());
        for(byte i = 0; i < n; i++)
        {
            partner1[i] = partner1Names.nextToken();
            partner2[i] = partner2Names.nextToken();
        }
    }

    public boolean checkForSelfPartner(String[] partner1, String[] partner2)
```

```
{  
for(byte i = 0; i < partner1.length; i++)  
{  
if(partner1[i].equals(partner2[i]))  
{  
return true;  
}  
}  
return false;  
}
```

```
public boolean checkForMultiUsage(String[] partner1, String[] partner2)  
{  
for(byte name = 0; name < partner1.length; name++)  
{  
byte usages = 0;  
for(byte i = 0; i < partner1.length; i++)  
{  
if(partner1[name].equals(partner1[i]))  
{  
usages++;  
}  
if(usages > 1)  
{  
return true;  
}  
}  
}  
for(byte name = 0; name < partner2.length; name++)  
{  
byte usages = 0;  
for(byte i = 0; i < partner2.length; i++)  
{  
if(partner2[name].equals(partner2[i]))  
{  
usages++;  
}  
}  
}
```

```

if(usages > 1)
{
return true;
}
}
}
return false;
}
public boolean checkPartnersMatch(String[] partners1, String[] partners2)
{
for(byte pair = 0; pair < partners1.length; pair++)
{
String p1 = partners1[pair];
String p2 = partners2[pair];
boolean foundPair = false;
for(byte i = 0; i < partners1.length && !foundPair; i++)
{
if(p2.equals(partners1[i]))
{
if(p1.equals(partners2[i]))
{
foundPair = true;
}
else
{
return true;
}
}
}
}
return false;
}
public static void main(String[] args)
{
new J5();
}
}

```

OK, there's a lot to do here! First, we got out of the static method by instantiating J5. This is simply so that our global variables and methods do not have to be static. Next, we defined a global scanner and set it up, and then we defined a global variable n, which is the number of students. This is so we can access them from any part of our code inside J5. Inside our J5 constructor, we read in the number of students and created two arrays, which contain the student names. Next, we called a method called assignPartners that uses a special class called "StringTokenizer." Essentially what this class does is allow us to split a string like a scanner. We can call nextToken on the tokenizer to get the next section of the string, just as when you call next for a scanner to get the next section of an input. So, our method has filled the two student arrays with names, using a loop that runs for n times where n is the global variable that was set to the number of students. After we called the assignPartners method, we have done three checks. We checked if any two partners were the same person, we checked if any student was used more than once, and we checked if partners matched (if two partners work out if you flip it).

So, first we have a method called checkForSelfPartners that returns a boolean; the method will return true if there is a case of self-partnership. The way we check for this is by looping through one array and checking if both arrays are equal at that position. If partners1[0] is the same as partners2[0], then that means someone is partnered with themselves, which makes the whole test case "bad." If they are equal, return true (this means it will ignore the rest of the for loop, and the method for that matter, and just give back the value). If the loop finishes without having returned a value of true, that means that at no point were the partners equal, and so there is no self-partnership, meaning you should return false.

Next, we have the checkForMultiUsage method, which again returns a boolean, so if a student name is used more than once, it will return true. We have two main loops that are simply duplicates of each other, except one is for the partner1 array and the other is for the partner2 array. These loops loop through the entire array. We have another loop inside that will loop through the array that is past our selected name. This is to decrease the number of iterations because if there is a duplicate, our name loop will find the first

occurrence of the duplicate, and then this inner loop will look through the rest of the array for any other occurrences of that element. So in this inner loop, we have simply checked to see if the name we have selected was equal to the element we were looking at (based on our “i” for loop), and if so return true, since that means a name has multiple usages. We have done exactly the same thing with the partner2 array. If the method goes through both loops without returning true, this means that at no point were the names used multiple times, so we return false at the end.

Finally, we have the checkIfPartnersDontMatch method, which returns a boolean on whether partners are invalid. This means that if student A was partnered with student B, and student B was partnered with student C, then the partners did not match and the method should return true. So how did we do this? First, we looped through the arrays (technically we looped through partners1, but we will be using the same index to access both arrays inside the same iteration) and in each iteration we set two variables, p1 and p2, each to the respective partner, based on the loop. Then we created a new for loop that will loop through the entire array again. In this loop, we checked if the reverse relation existed, meaning we looked for whether p1 was the same as the partner2 at the loop position, and if p2 was the same as partner1 at the loop position. If one is true but not the other, then we return true, as the partners don’t match. If both partners are true, then we break the inner loop and move on to the next set of partners. Finally, if the code runs through without returning true, we return false. So back to our constructor, we check to see if any of these methods return true, and if so, we output “bad” to the user, as any of these scenarios (a student is his own partner, a student is partners with multiple people, or a partnership does not match) will make the response be bad. Otherwise, output “good.” If we look at our constructor, it is rather simple. It’s almost English! That is why making methods is a good idea: It helps to make your ideas clearer and your code easier to read and understand.

Bonus Chapter

Algorithms

Let's take a look at a few low difficulty problems on ProjectEuler (<http://projecteuler.net>) that we can solve using Java. We will explain how we can approach each algorithm to help you start thinking in a programmer's mindset, and then you can try to solve it yourself. It is recommended that you sign up for this site and practice your newfound Java skills there.

Multiples of 3 and 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Now let's think about this. What exactly is this problem asking from us? It wants us to find all the multiples of 3 and 5 below 1000. Now mathematically, without programming, let's think about how we would approach that. Anything that is divisible by 3 or 5 that is within the sequence from 1 to 999 would be added to a list of numbers.

After finding all the numbers that are divisible by 3 or 5, we would then take a look at our list of numbers and add all those numbers up to provide the sum to ProjectEuler to see if we got the answer correct. This seems easy enough, but it would take a long time to complete if we were simply to use pen and paper. But now, with our knowledge of Java, we are able to figure out how to get a program to find our answer.

In order to actually cycle through from 1 to 1000, we should use a loop. It doesn't matter what type of loop we use, but we need to use a loop that affects a variable to increment from 1 to 1000, not including 1000 because the question is asking us to find the sum of all the multiples of 3 or 5 below 1000, not below or equal to.

Then, through each iteration of the loop, we must check if the variable being affected by the loop is divisible by 3 or 5. There are multiple ways to do this, which can vary from checking the remainder (using the % symbol) or dividing and checking whether or not it is a whole number. This should be pretty easy to figure out on your own. The next thing we must do, if the number is divisible by 3 or 5, is add it to a list. We can add it to a list and then loop through the list (or array), or we can add all the numbers together, or we can create a variable before the loop and simply add on to it any number found to be divisible by 3 or 5. The most efficient method is the lattermost.

The reason we have pointed out the array method is essentially to let you know that there are multiple ways to solve problems in programming, but it is important to find out which way is the most efficient way—using your knowledge and a spice of logic. Now try programming this using what you know in Java and the logic I have provided.

We first check if our solution works with their example:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string args[]) {
int limit = 10;
int sum = 0;
for (int i = 3; i < limit; i++) {
if ((i % 3) == 0) {
sum += i;
} else if ((i % 5) == 0) {
sum += i;
}
}
System.out.println("The sum below " + limit + " is: " + sum);
}
}
```

We then get the output: The sum below 10 is 23.

The output is correct, so we are going to input the number they ask to solve the problem with the following code:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string args[]) {
int limit = 1000;
int sum = 0;
for (int i = 3; i < limit; i++) {
if ((i % 3) == 0) {
sum += i;
} else if ((i % 5) == 0) {
sum += i;
}
}
System.out.println("The sum below " + limit + " is: " + sum);
}
}
```

This time the limit is 1000. The output is now: The sum below 1000 is 233168.

Even Fibonacci Numbers

Problem 2

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

In this problem, we are told the pattern by which the Fibonacci sequence is generated and then we are told what we must solve. This problem asks us to find the sum of the even-valued term values in the Fibonacci sequence. Don't get confused and assume they are talking about the term indices!

There are technically two problems here, one of which we have already completed. You saw, in the last problem, how we were checking whether or not a number was divisible by x or y in a range from 1 to 1000 and then adding it to an accumulation variable. Well, the same idea applies here, where as we generate the Fibonacci sequence we are checking whether or not the term value that was generated in that iteration is even or odd. If the term is even, then we simply add it to an accumulation variable; if not, we do nothing with that iteration.

So truly, we're only really solving the Fibonacci problem here. This is the great thing about practicing algorithms: You start to gain a "muscle memory" over them so you become more and more efficient.

So how exactly do you generate the Fibonacci sequence? Well, we know that you start with 1 and 2. Then you add the two terms together to get the third. Then you get the last 2 terms and add those together to get the fourth, and so on.

1, 2, 3, 5, 8, 13...

$$1 + 2 = 3$$

$$3 + 2 = 5$$

$$5 + 3 = 8$$

$$8 + 5 = 13$$

Now that we have a better understanding of how the Fibonacci sequence works, let's see how we can recreate that pattern in Java. First, we must create a loop that has the number of iterations identical to the term value that we want to reach. So if we want to get the third term value, then we have three iterations, and if we have three iterations, we're going to end up getting the value of 3, because in the sequence the third term value is 3.

However, before creating the loop, we should declare two variables: one that is initialized at one, and another that is initialized at two. Why? Because we should simulate exactly how the Fibonacci sequence works by having these two variables dynamically change as the iterations of the loop go on. Let's go back to our loop and look at how we would do this. First, let's create a quick variable that is the sum of the two variables we initialized before the loop. Next, let's check if the second variable that we declared as 2 is even, and if it is, let's add it to an accumulation variable. Next, we assign our variable that was assigned 1 to the second variable that was assigned 2, and then assign our second variable that was assigned 2 to the sum of the two numbers.

Now if you read that carefully, you would understand that we are simply cycling through the Fibonacci sequence in the exact algorithm that I have just specified. Now you can try it out!

Solution for the first ten terms by specifying a limit of 89:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string args[]) {
int limit = 89; // declaring limit
int sum = 0; // declaring sum
int left = 1; // first term of fib seq
int right = 2; // second term of fib seq
while (true) { // loop continuously
int sumLR = left + right; // adding two prev terms
left = right; // swapping values
if (right > limit) { // if goes over limit, break
break;
} else if ((right % 2) == 0) { // if even, add to sum
System.out.print(right + ", ");
sum += right;
}
right = sumLR; // swap values for next iteration
}
```

```
System.out.println("\nThe sum is: " + sum); // display sum
}
}
```

Solution for problem with limit changed to four million:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string[] args) {
int limit = 4000000; // declaring limit
int sum = 0; // declaring sum
int left = 1; // first term of fib seq
int right = 2; // second term of fib seq
while (true) { // loop continuously
int sumLR = left + right; // adding two prev terms
left = right; // swapping values
if (right > limit) { // if goes over limit, break
break;
} else if ((right % 2) == 0) { // if even, add to sum
System.out.print(right + ", ");
sum += right;
}
right = sumLR; // swap values for next iteration
}
System.out.println("\nThe sum is: " + sum); // display sum

}
}
```

Now let's check out another Project Euler problem.

Largest Prime Factor

Problem 3

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143?

Now in order to solve this problem efficiently in under a minute, we must carefully consider how we want to approach solving this problem. Since we are asked to find the largest prime factor for a really high digit number, we can face long processing times while the program figures out if our algorithm is properly configured. That is why we must think about this carefully.

We know that if we take a number like 100 and want to figure out the largest prime factor, we would do the following:

$$100 / 2 = 50$$

$$50 / 2 = 25$$

$$25 / 2 = \text{Remainder}$$

$$25 / 3 = \text{Remainder}$$

$$25 / 4 = \text{Remainder}$$

$$25 / 5 = 5$$

$$5 / 2 = \text{Remainder}$$

$$5 / 3 = \text{Remainder}$$

$$5 / 4 = \text{Remainder}$$

$$5 / 5 = 1$$

Therefore, the largest prime factor of 100 is 5.

Once our answer has reached one, we know that we have reached our greatest prime factor. This is an efficient way of completing this algorithm because it means that we are not constantly processing the large number that we want to find the prime factor of, but rather are reducing its size as a number as we divide it until we find the largest prime factor of that number. This makes sense because all we are doing is simply slowing the reduction of the number as we change its ratio through division until we reach a certain point where we can't divide by anything other than 1 or itself, which is the definition of a prime number.

So, how would we approach this problem programmatically in JAVA? Well, the first thing we must do is again create a loop, but right before it, create a

boolean that declares the MaxPrimeFound variable as false. Once we have completed this, we are able to process a loop from 1 to the number's variable itself. While we are processing, we try dividing the variable being affected in the loop's process, by a factor of 2 for example, and checking whether or not it is a whole number. If it is a whole number, we continue on. If it is not a whole number, we increment the dividing factor until we reach a point where it divides evenly. If the number that it divides evenly is in to itself, then we know that we have indeed found the largest prime factor of the large digit number. We simply repeat this process until it divides into itself and becomes 1.

Solution using their example:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string[] args) {
int number = 13195;
int factor = 2;
int largestPrimeFactor = 0;
while (true) {
if ((number % factor) == 0) {
number /= factor;
}
else {
factor++;
if (factor == number) {
largestPrimeFactor = number;
break;
}
}
}
System.out.println("The largest prime factor is: " + largestPrimeFactor);
}
}
}
```

Output: The largest prime factor is 29.

This is correct when we are using the number 13915. Let's figure out the problem's solution, though:

```
Import java.io.*;
Import java.lang.*;
class MainClass {
public static void Main(string[] args) {
long number = 600851475143L;
int factor = 2;
long largestPrimeFactor = 0;
while (true) {
if ((number % factor) == 0) {
number /= factor;
}
else {
factor++;
if (factor == number) {
largestPrimeFactor = number;
break;
}
}
}
System.out.println("The largest prime factor is: " + largestPrimeFactor);
}
}
```

Output: The largest prime factor is 6857.

Final Words

Where to Go From Here

This is the start of your journey as a Java programmer. You have just barely scratched the surface with this guide, as learning the syntax and conventions of a language is just the beginning. The most important part of programming is the logical aspect of it. Sure, you may know how to loop through an array of variables like a list of shopping items, but if someone asks you to process an image using your knowledge of programming, with the help of an API and some thinking you can figure out how to invert the colors of an image, flip it, rotate it, scale it, etc.

The real programming comes in the logical portion of the mind. It's similar to learning any other language, like English for example. You may understand the grammar rules and the conventions, like adding periods to the end of sentences—but the true skill lies in being able to write clean and logically thought-out structured essays. The same concept applies to programming, where the person writing the code must know how to apply her or his knowledge of the rules of the language (like Java) and use it to his or her advantage to come up with neat programs.

The knowledge and understanding of programming is truly great because it's the closest thing we have to having magical powers. You can literally create something from scratch out of an empty notepad and have it function to do things you want it to do. Whether that be a bot to analyze and predict the stock market or a video game, that choice is yours.

In this guide you have learned the fundamentals of Java. You haven't learned all the possible methods that can be used in the language, but that isn't the point. This guide was designed to set you on a journey toward discovering objects and methods that you need in order to help you to create programs that you desire. You have been given the optimum knowledge to understand how to read an API and to be able to understand what it adds to your code.

We hope to see you soon again in the upcoming editions of our programming

language learning books.

Until next time,

Alphy Books