

JAVA

**A Beginner's Guide to
Learning the Basics of Java Programming**



JAMES PATTERSON

JAVA

A Beginner's Guide to Learning the Basics of Java Programming

By James Patrick

© Copyright 2015 by James Patterson – All rights reserved.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective author own all copyrights not held by the publisher.

Legal Notice:

This book is copyright protected. This is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part of the content within this book without the consent of the author or copyright owner. Legal action will be pursued if this is breached.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. Every attempt has been made to provide accurate, up to date and reliable complete information. No warranties of any kind are expressed or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice.

By reading this document, the reader agrees that under no circumstances are we responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, - errors, omissions, or inaccuracies.

TABLE OF CONTENTS

Chapter 1 – Introduction to Java Programming

Chapter 2 – Getting Started

Chapter 3 – Java Variables

Chapter 4 – Control Flow

Chapter 5 – Java Arrays

Chapter 6 – Java String Methods

Conclusion

Chapter 1 – Introduction to Java Programming

With Object Oriented Programming such as Java, it is possible today to organize complex and large programs through encapsulation, polymorphism, inheritance, objects, and classes.

For many years, C++ used OOP language. With the rise of the World Wide Web, Java programming became more popular, especially in the development of consumer electronics such as television, microwaves, and more.

Computer experts devoted a lot of their time in trying to find software that is safe, reliable, compact, and processor independent. Java programming gradually progressed to become a full-pledged programming language, changing its focus from consumer electronics to a wide range of platforms to develop more powerful applications.

JAVA PROGRAMMING – A BRIEF HISTORY

The Java Programming Language was developed in 1991 by five computer experts – Ed Frank, Mike Sheridan, Chris Warth, Patrick Naughton, and James Gosling who all worked for Sun Microsystems Inc. It took 18 months for them to develop the program, which was initially named “Oak.” It was renamed Java in 1995 because of copyright concerns.

The concept was to create a programming language that can be used across platforms and that could build embedded software for consumer electronics. The popular programming languages at the time, C and C++ were not efficient for this purpose, because they are dependent on platform as the programs written on them should be compiled first for specific hardware before launching. In addition, the compiled code was not efficient for other processors and it should be re-compiled.

Hence, the team of five, also known as the Green Team, started to work on building an easier programming language. They tinkered for a year and a half in creating a compact, platform-independent programming language, which can allow a programmer to build a code that could run on different processors under various environments.

This led to the development of Java. Simultaneously, the World Wide Web and the Internet were becoming popular. The web programs were still dependent on platforms, and required the programs that could operate on any OS regardless of the software and hardware configuration.

It required for compact and small programs, which could be easily carried over the network. Java was the language that complied with such requirements. Web developers soon realized that a language that is architectural neutral such as Java can be ideal for writing programs for the web.

Hence, Java became more popular as a programming language for the World Wide Web, from its humble beginnings as a language for consumer electronics. Today, Java is far from a basic programming language. This is a well-developed technology that is simple, secure, portable, platform independent, multi-threaded, object oriented, distributed, and robust.

PRIMARY FEATURES OF JAVA

Simple

Java is regarded as a simple language, because it doesn't have complicated features such as Explicit memory allocation, pointers, Multiple inheritance, and Operator overloading.

Secure

Java comes with a virtual firewall between the computer and the application. Java codes are restricted inside the Java Runtime Environment (JRE), which doesn't approve unauthorized access for the system resources.

Portable

A code written in Java on one platform could run on another platform on a different machine. The Java byte code could be transported to any platform for operation, which makes java code very portable.

Platform Independent

A platform refers to a pre-established set-up to run a program, conform to its restrictions, and use its features. During the compilation phase, the java program is converted into a byte code, which could be used to any platform such as Mac/OS, Linux, or Windows. Therefore, a program that has been compiled on Linux can still be used on Windows and vice versa. That is why Java is a platform independent programming language.

Multi-threading

Java supports multi-threading because it allows a program to perform several tasks all at the same time.

Object Oriented

Java is an object oriented programming language, because it can organize programs as a group of objects that each represent an instance of a class. The four primary concepts of OOP are: polymorphism, inheritance, encapsulation, and abstraction. You will understand each concept as you learn java programming.

Distributed

Through java programming, you can develop distributed applications. Enterprise Java Beans (EJB) and Remote Method Invocation (RMI) are employed for developing distributed apps using Java. To put this simply, you can distribute java programs on several systems that are linked to one another through the Internet. Objects within a Java Virtual Machine (JVM) could run protocols using a remote JVM.

Robust

Mishandled runtime errors and memory management mistakes are the two main problems which cause program failures. Java can handle these issues with high efficiency. Mishandled runtime errors could be resolved through Exception Handling protocol, while memory management mistakes could be resolved by garbage collection, which is an automatic de-allocation of objects that are already unnecessary.

Chapter 2 – Getting Started

Before you can start with Java, you need to install everything that you need. The struggle is real: you can experience headache even before you write a single code. This chapter is intended to make your experience a bit easier. I recommend writing your code using NetBeans, which is a free software and one of the most popular Interface Development Environments (IDEs). But before you can just do that, you need to install the important Java files and components. The first thing that you need to install is the Java Virtual Machine.

THE JAVA VIRTUAL MACHINE

As we have already discussed, Java is platform independent, so it can be used on any operating system. Hence, regardless if you are using Mac OS, Linux, or Windows, it will be all the same with Java. The Java Virtual Machine (JVM) is the main reason why the program could run on any OS. The Virtual Machine is a program, which could accurately process all your codes. Hence, you need to install this program before you can operate any form of Java code.

Because Java is owned by Oracle, you first need to visit the company's website to download the JVM or also known as the JRE (Java Runtime Environment). But as a shortcut, you can click on the link below:

<http://java.com/en/download/index.jsp>

To check if JRE is already installed in your computer, just click the link "Do I have Java?" that you can find beneath the large Download button at the upper portion of the page. Once you click the link, your PC will be scanned to find if you have the JRE. A notification will be sent to you if you have the JRE or not. If you don't have the JRE, you will be prompted to download and install it. But as a shortcut, you can click on the link below:

<http://java.com/en/download/manual.jsp>

The link above will direct you to a page where you can manually download the JRE. The page will provide you with the instructions and download links for different OS.

After you download and install the JRE, you might need to re-start your PC, before you could use the program.

THE JAVA SOFTWARE DEVELOPMENT KIT

Even if you have the JRE, you still can't write any program. The JRE will only allow the Java programs to run on your PC. In order to write a code and test it out, you need to download and install the Java Software Development Kit. You can download it by clicking the link below:

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

It is recommend that you use the Java Standard Edition or Java SE. Click the Top Downloads located on the right of the page, and you will be redirected on a separate page containing a list of options for download. Since we will use NetBeans, you need to find **JDK 8 with NetBeans**.

Click on the Download link, which will direct you to another page. Find the link for your OS. But take note that this download requires about 290 MB for Windows (64 bit). After downloading the JDK and NetBeans, you can install it on your PC. In this book, we will use NetBeans to write codes.

But before using the software, you first need to understand how things work in the Java world.

THE JAVA SYSTEM

You can use a text editor to write the actual code for your program. NetBeans provide a special space for writing a code. This code is known as a source code, and is stored in a file extension - .java. A special program known as Javac will be used to convert the source code into a byte code. This process is known as compiling. Once Javac has completed compiling the byte code, it will make a new file with the .class extension. When the class file has been generated, it could be used on the JVM.

As a summary:

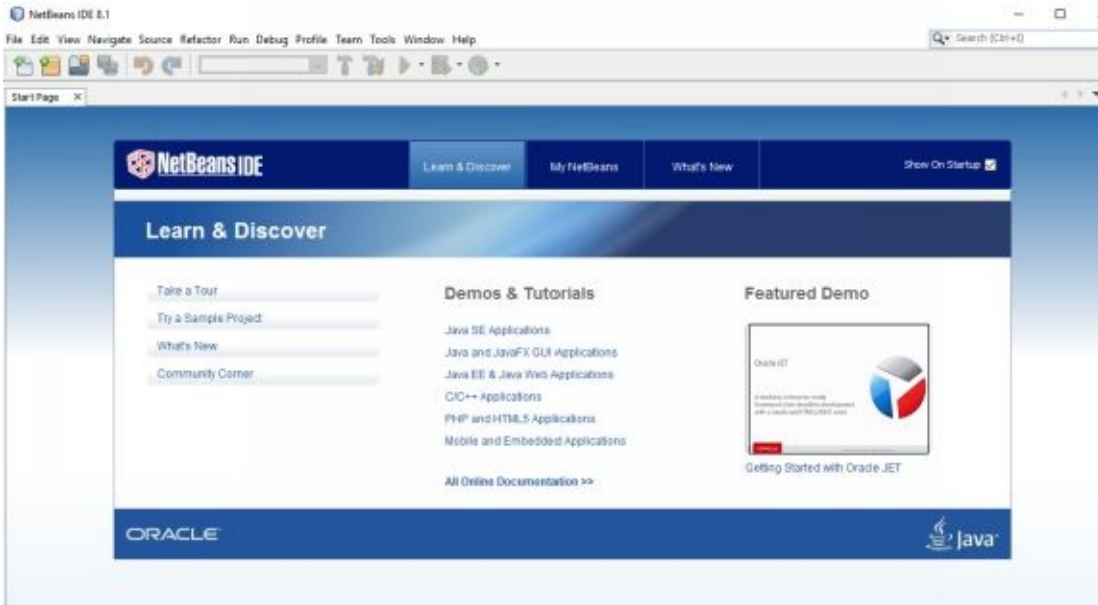
- The source code is created using the .java extension.
- Compiling is processed through Javac and will create a file with .class extension.
- Initiate the compiled class.

Using NetBeans, you can easily compile and create files. At the background though, it will take the source codes and create the java file. This will run Javac and compile the class file. NetBeans will then run the program inside the software, which will save you the trouble of initiating a terminal window as well as encoding a lengthy series of commands.

After understanding how Java works, you can now run the NetBeans software.

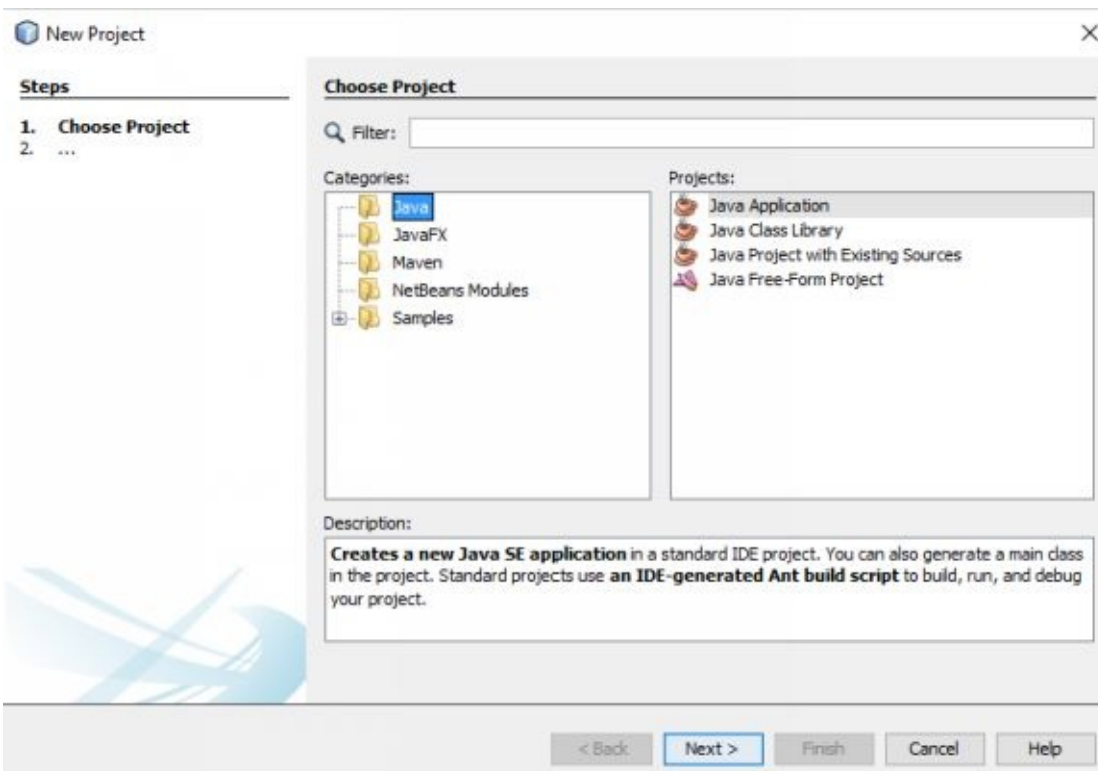
THE NETBEANS SOFTWARE

In first running the NetBeans Software, you will see a screen like this:

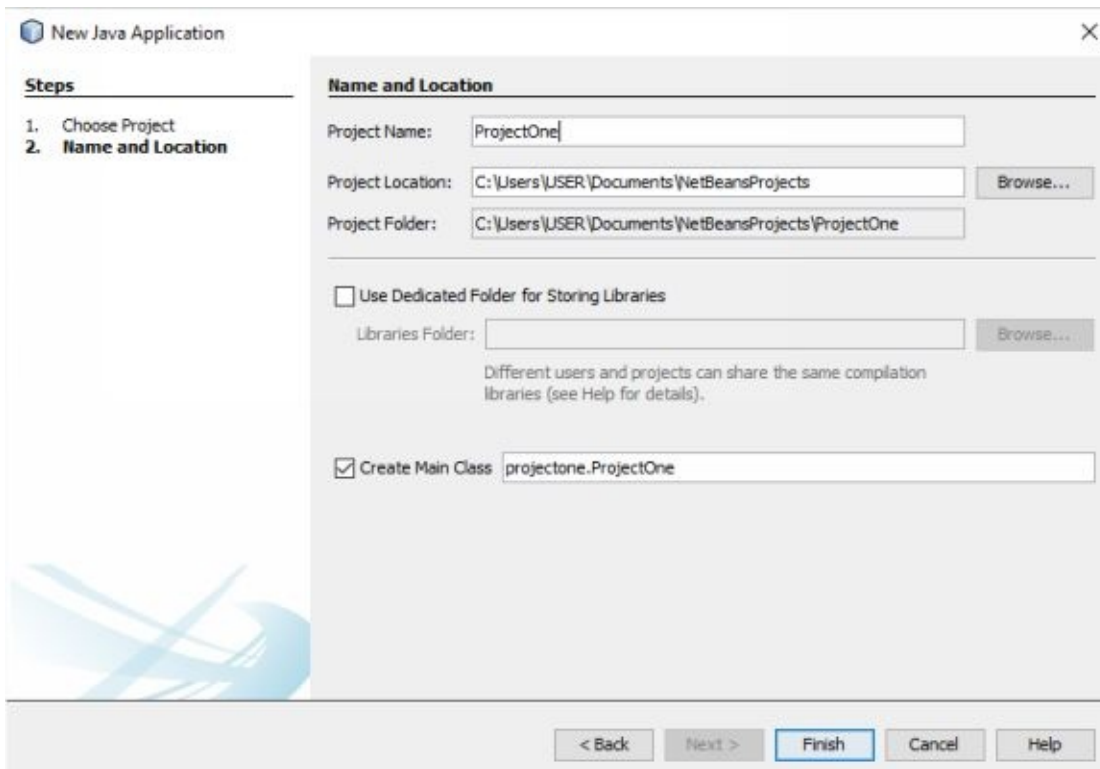


Go ahead and have a cup of coffee as this is not known for its speed.

At the top of the NetBeans menu, you can click on **File>New Project** to initiate a new project. The dialogue box below will appear:



You need to make a Java Application, so choose Java under the Categories menu, then **Java Application** under the **Projects** category. To go to step two, click the Next button.



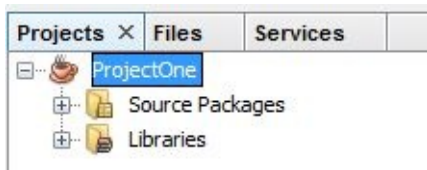
The first area shows the Project Name. Click on the blank space and choose a title for your Project. You will notice that the text at the bottom also changes to match the name of the project.

Our Class will be known as ProjectOne with capitalized P and O. The package is also known as projectone, but with small caps p and o. Notice that the default location to save your projects also appears in the text box for Project Location. You have the freedom to change this, if you like. NetBeans can also generate a folder with your chosen project name and will be saved in the same location. Just click the Finish button and NetBeans will run to create all the needed files.

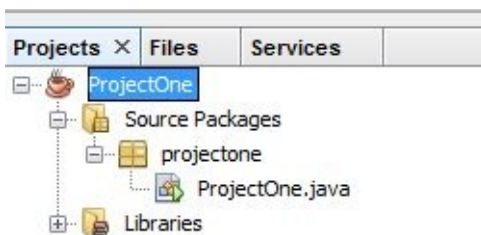
Once NetBeans redirects you to the IDE, observe the Projects area located in the upper left of the screen. If you cannot see this, just click **Windows > Projects** from the menu bar at the upper part of the software.



When you click the plus symbol, you can expand the project, and you will see the screen below:

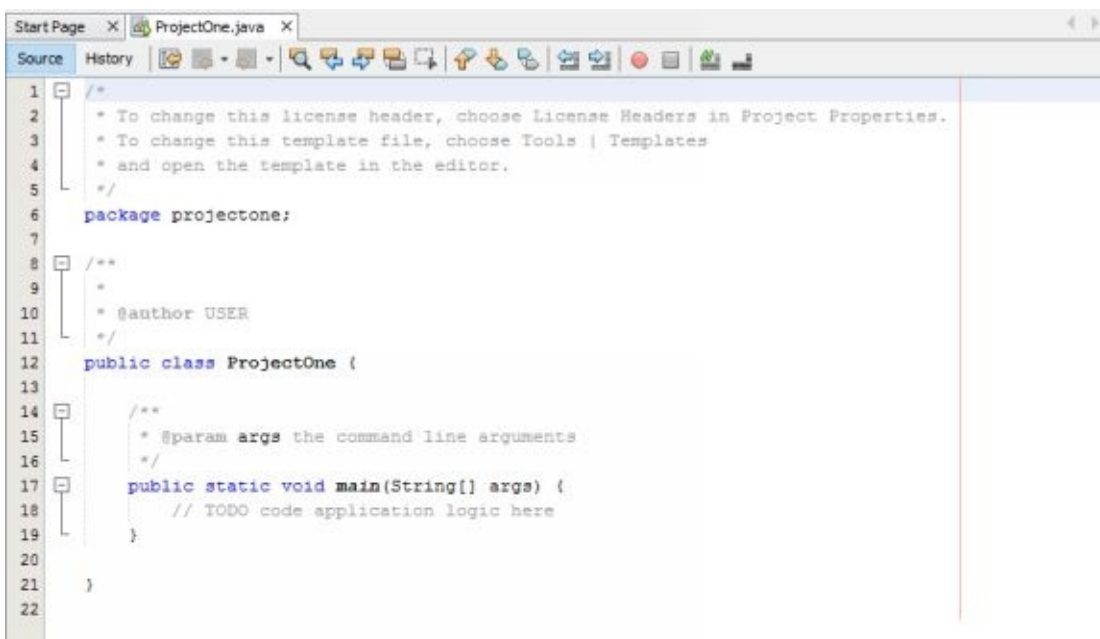


Next, you can expand the Source Packages to see the project name. You can expand this and you will see the Java file, which is actually your source code.



The same source code will be shown to the right, in the bigger text space. This will be called as ProjectOne.java. If there is no code window, you just need to double click the ProjectOne.java in your Projects screen as show above. The code will be shown so you can start working on it.

The coding screen is shown below:



Take note that the class here is known as ProjectOne:

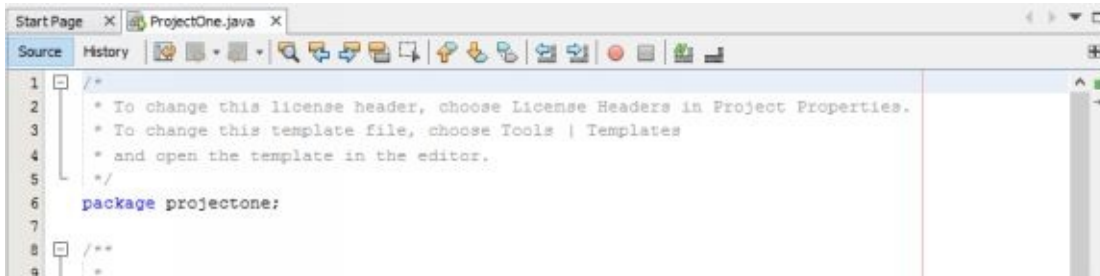
```
public class ProjectOne {
```

This is similar to the name of the java source file in the project window: ProjectOne.java. Once you run the programs, the compiler will require the source file as well as the class name. Hence, if the .java file is known as projectOne but the class is known as ProjectOne then you will have an error on compiation. This is all because of the lower case “p” and the second one is capitalized.

Take note that even though we have also called the package ProjectOne, this is not necessary. You can use a different name for the package, because the package name doesn’t need to be the same as the java source file or the source file class. It is only the name of the class and the name of the java source file that should be the same.

JAVA COMMENTS

Once you create a New Project in NetBeans, you will notice that there are text in grey color, with asterisks and slashes.



The text in grey are comments. Once you run the program, these will be ignored. Hence, you have the freedom to type anything you like as comments. It is typical for the comments to describe what you are trying to do. You can enter a single line comment by typing two slashes then your comment.

```
//This is an example of a single line comment in Java.
```

If you like to enter comments in several lines, you can perform this:

```
/*  
    This is an example of a comment  
    That takes two lines or more.  
*/
```

Or this:

```
//This is another example of a comment  
// that takes two lines or more.
```

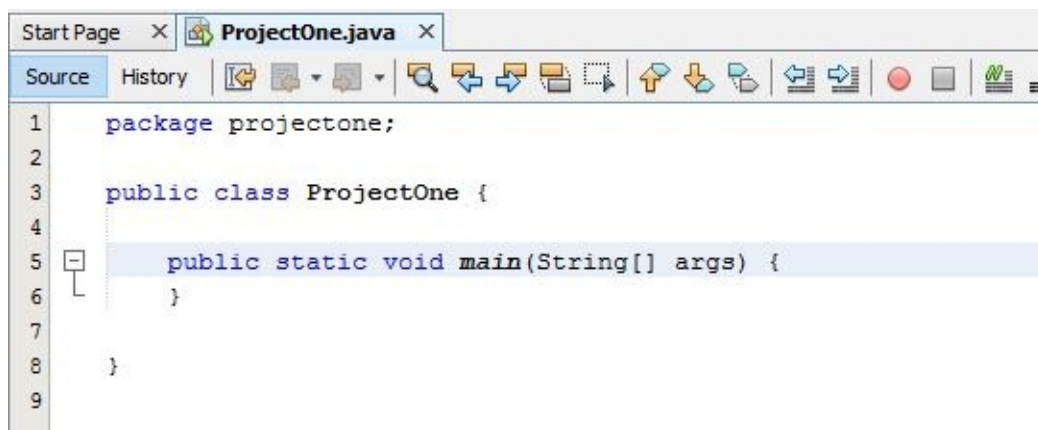
In the first option, notice how the comment begins with `/*` and ends with `*/`.

Then there's the Javadoc comment, which begins with one forward slash followed by two asterisks (/**). It also ends with an asterisk with one forward slash (*). Take note that every line of the comment begins with an asterisk:

```
/**  
 *This is an example of a Javadoc type comment  
 */
```

Javadoc comments are used to record code. The recorded code could be converted into an HTML page, which will help other programmers to make sense of the code. Click Run from the NetBeans menu to see what this will look like. From the Run menu, choose Generate Javadoc. Don't expect to see much as we haven't started writing any code yet.

At this point, it's okay to do away with the comments first, which the NetBeans have generated for you. Below is the code area without the comments:



In the above screenshot, you can see the name of the package first. Take note that the line ends using a semicolon. Remember, without the semicolon, the program will not initiate the compilation process.

package projectone;

Next is the class name:

```
Public class ProjectOne {  
}
```

The class is considered as a code segment. However, you need to specify where the segments begin and end by adding curly brackets. The beginning of a code segment is indicated with a left curly bracket { and ends with a right curly bracket }. Any code confined in these brackets are included in that code segment.

Meanwhile, anything that is inside of the right and left curly brackets for the class is also a considered a code segment. Take a look at this:

What's inside of the left and right curly brackets for the class is another code segment. This one:

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

Take note that the text “main” is the most essential word here. Once you initiate the program, it will search for a method named main. A method is just a segment of code, and you will learn more about the other segments later on. It will then launch any code inside the curly brackets for the main. There will be error without the main method in the program. But as the name suggests, it refers to the main entry point for the programs.

For now, don't worry about the blue text before the text “main”. But in case you like a glimpse, public refers to the method, which could be seen external of the class, while static signifies that there is no need to build a new object, and void means it will not yield a value. The sections between the rounded brackets of main are known as the command line arguments. Confused? Well, you can learn more about them later, so don't worry.

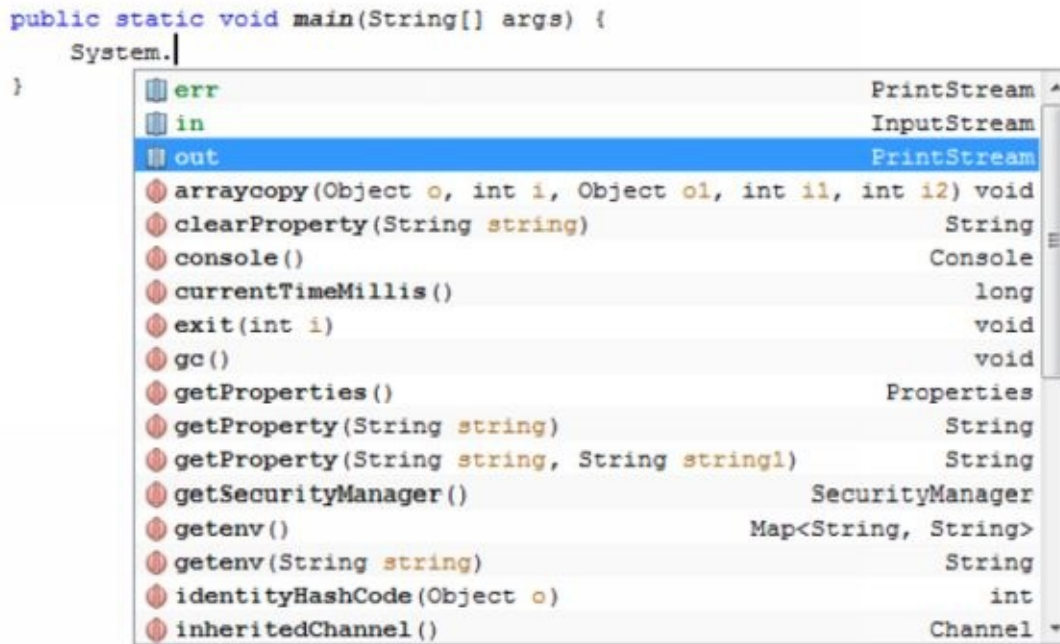
The important pointer to take note is that you now have a class known as ProjectOne. This class involves a method known as main. These segments have their own curly brackets. However, the main chunk of the code refers to the ProjectOne class.

Now, let's learn how we can print to the output screen or window.

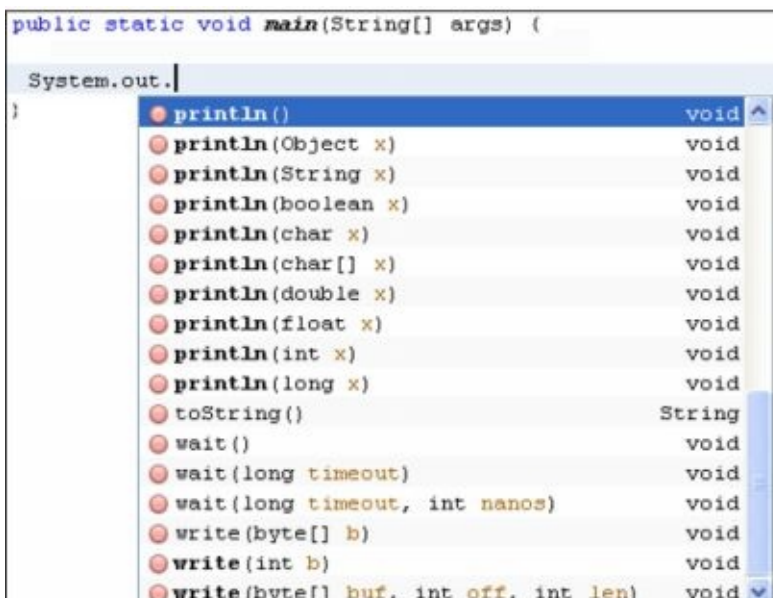
At this point, you can run the code and convert it into a real program. It will not do anything, but you can initiate the compilation process. Now, add another code line to see how it will work. You can add some words to a console window. You can insert the following line to the main method:

```
public static void main( String[ ] args ) {
    System.out.println( "My Project One" );
}
```

Once you end the word System with a full stop, the program will help you by showing a list of options:



Click the out option twice so you can add it to your code. Then, add another full stop, and the list of options will appear again:



Choose the **println()** option, which will allow you to print a single line of text to the output window. However, you should place the text between the rounded brackets for **println**. The text must be enclosed in double quotes.

```
public static void main(String[] args) {  
    System.out.println("");  
}
```

```
public static void main(String[] args) {  
    System.out.println( "My Project One" );  
}
```

When you enter your double quotes, you can add the text of your choice:

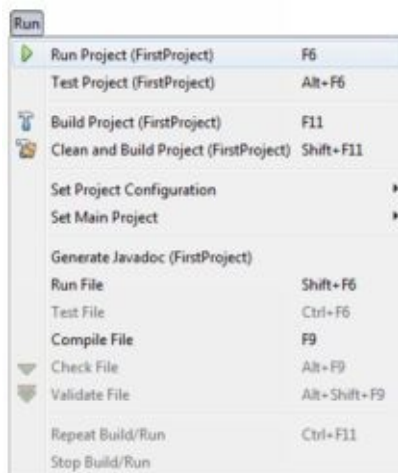
Take note that the line ends using a semicolon. Every completed code line in Java should end using a semicolon. Without the semicolon, the program will not initiate the compilation process.

Now, we could test our first program. But before you can do that, be sure to save your code. You can do this by clicking the Save icon in the NetBeans toolbar or following this command line: **File > Save** or **File > Save All**.

RUNNING JAVA PROGRAMS

Once you run a program in NetBeans, the software will run it in the Output screen under the screen at the bottom of the code. This is easier because there is no need to initiate a console window or a terminal. The Output screen IS your console.

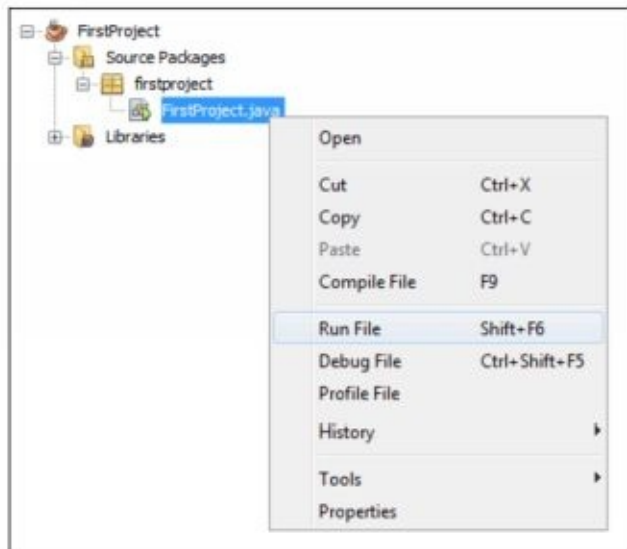
There are several methods of running a program in NetBeans. The simplest way is to hit F6 on Keyboard. Also, you can run a program using the menu on the NetBeans toolbar: Run > Run Project (Name of your project).



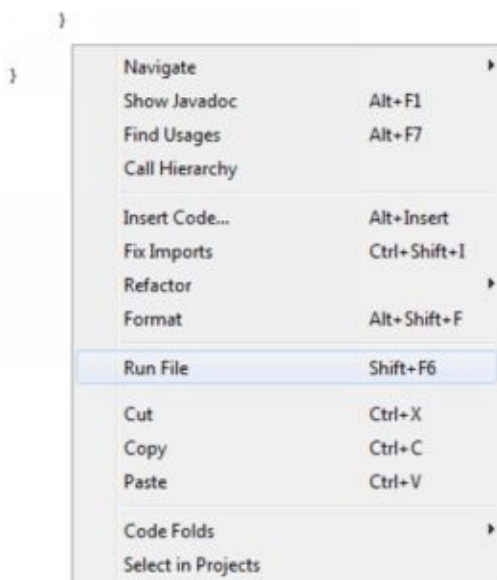
Another option is to hit the green arrow icon on the toolbar.



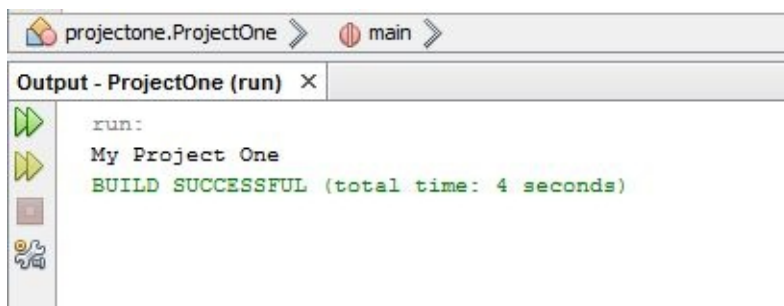
Another method in running a program is through the Projects screen. This will make certain that you are running the right source code. Just right click the java source file in the projects screen and the menu will appear. Choose Run File.



You can also run the program by right clicking within the code screen. As shown in the screenshot below, we have right clicked before the final curly bracket.



Choose your preferred method and run the program. In the Output screen, you will see something like this:

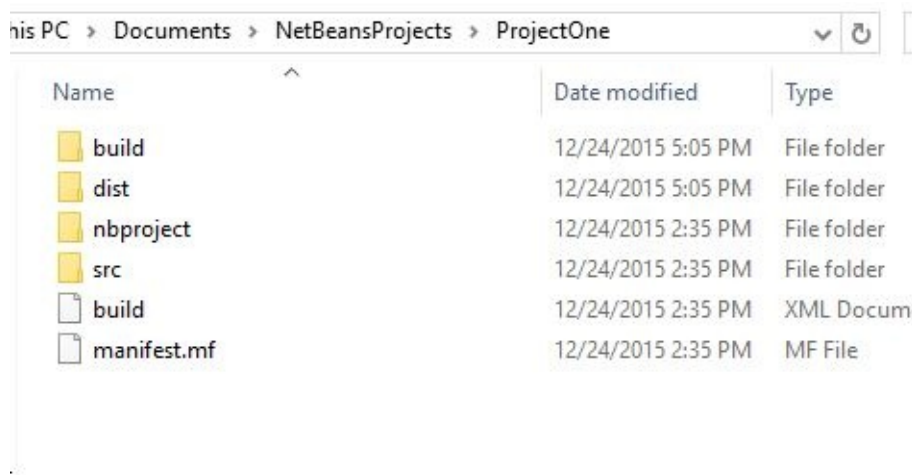


The second line in the Output screen in the image above is our code: My Project One. It's easy to initiate a re-run by clicking the two green arrows in the left side toolbar.

SHARING JAVA PROGRAMS

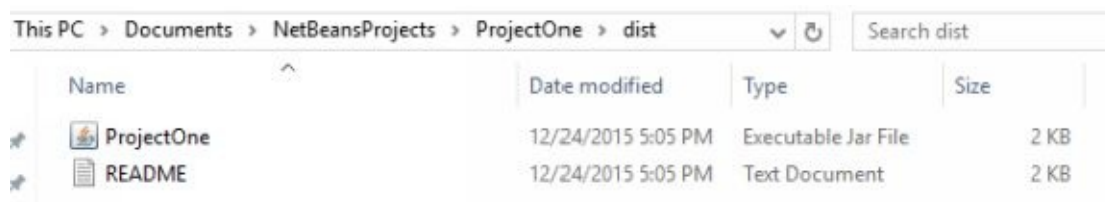
Java allows you to share your programs to other people, so they can also try running them. In order to do this, you should first create a Java Archive file or JAR file. NetBeans will help you do this. On the Run menu, choose **Clean and Build Main project**.

NetBeans will then save your code and will make all the needed files. It will also make a new folder named **dist** where it will save all the files. Take a look at the location where the projects are and you will also see the folder named **dist**.



Name	Date modified	Type
build	12/24/2015 5:05 PM	File folder
dist	12/24/2015 5:05 PM	File folder
nbproject	12/24/2015 2:35 PM	File folder
src	12/24/2015 2:35 PM	File folder
build	12/24/2015 2:35 PM	XML Document
manifest.mf	12/24/2015 2:35 PM	MF File

Open the dist folder and take a look inside:



Name	Date modified	Type	Size
ProjectOne	12/24/2015 5:05 PM	Executable Jar File	2 KB
README	12/24/2015 5:05 PM	Text Document	2 KB

There must be a JAR file and a README txt file, which stores the instructions on how you can run the program from the console or terminal screen.

At this point, you already know how you can run java source files. The next chapter will help you learn more about actual programming.

Chapter 3 – Java Variables

In programming, always remember that the language works through the use of data stored in memory. This data could be objects, text, or pointer numbers to other data areas. This data is provided in a name. Hence, it will be retrieved once you need it. Variable refers to the name as well as its value. We will begin learning about number values.

In java, there are different options in storing numbers. Whole numbers like 2, 5, 10 and so on can be held using the variable `int`, which means integer. Point numbers such as 2.4, 5.8, 10.2, and so on are stored through the use of the variable `double`. We can store it using the equals symbol (`=`). In this section, we will take a work on examples, using your ProjectOne code.

To establish an integer, just enter the main line code method of the ProjectOne.

```
public static void main(String[ ] args) {  
    int first_number;  
    System.out.println("My Project One");  
}
```

In order to command the Java program to store an integer, you first need to start type `int` and space. It is important to think of a title for the whole number variable. Generally, you have the freedom to choose a name, as long as you follow the rules below:

1. You can't start a variable name using a number. Hence, `first_number` is fine, but `1st_number` is not. It is acceptable to include numbers anywhere on the name variable except at the beginning.
2. You can't use Java keywords as name variable. These keywords will instantly turn blue when you type them such as `int`.
3. Spaces are not allowed in naming variables. The expression variable **`int first number`** would result to an error. Instead, you can use underscore in place of space. It is an industry practice that the main word begins with a small text and the next word capitalized such as `myFirstnumber` or `firstNumber`.

4. Case sensitivity is crucial for name variables. Hence, FirstNumber and firstNumber are not the same.

To keep anything within variable known as **first_number**, just include an equals symbol followed by the value that you like to keep.

```
public static void main(String[ ] args) {  
    int first_number;  
    first_number = 5;  
    System.out.println("My Project One");  
}
```

Java will interpret this that you like to keep the value 5 in the int variable, which refers to the first_number.

You can also write this all in a single linecode:

```
public static void main(String[ ] args) {  
    int first_number = 5;  
    System.out.println("My Project One");  
}
```

In order to see how this code works, make a slight change in the println method:

```
System.out.println( "First number = " + first_number );
```

Inside the println rounded brackets, you now have a direct text confined in double quotations:

```
("First number = "
```

Followed by a plus symbol and the variable name:

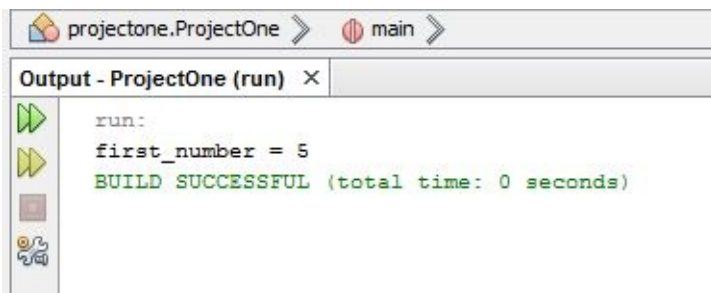
+ first_number);

Adding the plus symbol will be interpreted by Java that you like to combine the variable name with the direct text. This process is called **concatenation**.

The coding screen will now appear like the screenshot below. Notice how every code line is ended using a semicolon.

```
public static void main(String[] args) {  
    int first_number;  
    first_number = 5;  
  
    System.out.println( "first_number = " + first_number );  
}
```

When you try to run the program, you'll see the Output screen under:



Hence, the number which you keep in variable that we call as the **first_number** refers to the text at the right of the equals symbol.

Now, try a basic addition. Type two additional int variables into your code. Another to define a 2nd number while the other to define the “answer”:

int first_number, second_number, answer;

Take note that there are three names of variables in a single line. It's possible to perform this using Java as long as variables could be of similar type. In this case, they are all integers. Every name of the variable is then divided using comma.

Then, you can hold another text in the added variables:

```
first_number = 5;
second_number = 10;
answer = first_number + second_number;
```

To define the variable `answer`, you need to get the sum of the 1st number and the 2nd number. You can perform addition by including the addition symbol (+). This will command the Java program to add values in the `first_number` and the `second_Number`. Once it is done, it will also keep the total variable located on the left part of the equals symbol. Therefore, instead of defining 5 or 10 to the name of the variable, it will sum up and will perform the assigning.

But Java has already interpreted what is the value of the double variables, so it is fine to use their names.

Now, turn the method of the `println` to the line below:

```
System.out.println("Addition Total = " + answer );
```

Remember, we are adding direct text within the double quotations with the variable name. The coding screen will appear like the screenshot below:

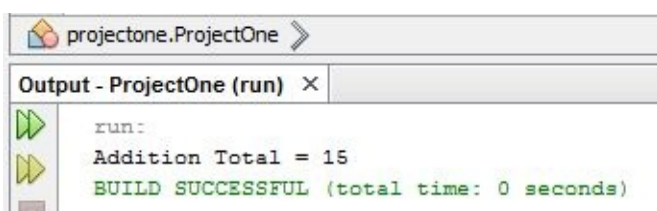
```
public class ProjectOne {
    |

    public static void main(String[] args) {
        int first_number, second_number, answer;

        first_number = 5;
        second_number = 10;
        answer = first_number + second_number;

        System.out.println( "Addition Total = " + answer );
    }
}
```

Run the program, and the result should look like what is shown in the screenshot below:



So far, we have performed these things in our first program:

- Stored an integer (first number)
- Stored another integer (second number)
- Combined these integers
- Held the sum of the integers in the third variable
- Print out the output
-

Alternatively, it is also possible to use direct number. Just turn the line answer to this:

```
answer = first_number + second_number + 10;
```

Click the run program, and see the results.

It is possible to store large integers using the **int** type, but the max value is 2147483647. To store a negative integer, the minimum value we can keep is -2147483648. To store higher or lower number, it is recommended to use the double variable type.

THE DOUBLE VARIABLE

The double variable could store very small or very large numbers. The max value is 1.7×10^{308} , and the minimum value is -1.7×10^{308} .

Floating point values such as 7.8, 11.6, or 14.5 can also be stored in a double variable. When you try to store a floating point value using an int variable, NetBeans will interpret it as an error.

Let's try practicing with double variables, using your ProjectOne code.

First, change the int variable to a double variable. Hence, this code:

```
int first_number, second_number, answer;
```

should be changed to this:

```
double first_number, second_number, answer;
```

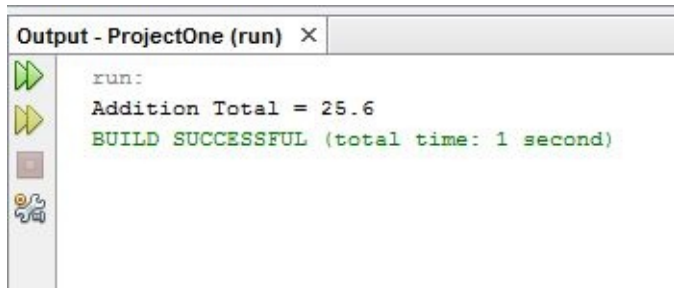
Then change the corresponding values of the first_number and the second_number:

```
first_number = 5.2;  
second_number = 10.4;
```

The coding area should look like the screenshot below:

```
public static void main(String[] args) {  
    double first_number, second_number, answer;  
  
    first_number = 5.2;  
    second_number = 10.4;  
    answer = first_number + second_number + 10;  
  
    System.out.println( "Addition Total = " + answer );  
}
```

When you run the program, the Output screen should look like this:



Change values stored in the `first_number` and `second_number`. Choose any value you like. Run the program and see the results.

SHORT AND FLOAT VARIABLES

Short and **float** are another types of variables that you can use. The variable short can be used to hold smaller number, ranging from -32768 to 32767. But rather than utilizing **int** in the code like in the previous codes, you can use short. Take note that you must only use the variable short if you are certain that values, which you like to store will not exceed 32767 or lower than -32768.

The double variable can hold very large numbers for the floating range point. But rather than using double, you can use float. In holding a value in a variable float, you should add f at the end of the number, such as:

```
float first_number, second_number, answer;
```

```
first_number = 5.2f;  
second_number = 10.4f;
```

Take note that f should be added after the actual value but not after the semicolon.

BASIC ARITHMETIC

Using the variables we have learned so far, we can use these symbols to perform computations:

- The plus symbol (+) is for addition.
- The minus symbol (-) is for subtraction.
- The asterisk (*) is for multiplication.
- The forward dash (/) is for division.

Let's do some exercise.

Get rid of the plus sign, which we have used to combine the `first_number` and the `second_number`. In its place, enter the minus symbol, the asterisk, and the forward dash. The result for the division must provide a large number (15.600000000000001), because the variable type that you have used is the double. But you need to turn the double variable to float and insert the letter f next to values. Hence, the code must appear similar to this screenshot:

```
public static void main(String[] args) {  
    float first_number, second_number, answer;  
  
    first_number = 5.2f;  
    second_number = 10.4f;  
    answer = first_number / second_number;  
  
    System.out.println( "Addition Total = " + answer );  
}
```

If you run this code, you will get 0.5. The program has rounded this up. Hence, the variable double type can store more numbers compared to the float. (Float has the capacity of 32 bits while double has 64 bits.

OPERATOR PRECEDENCE

Of course, it is possible to compute more numbers. However you must define specifically what should be computed. Let's add another number in our code.

```
first_number = 50;
second_number=25;
third_number=15;
    answer = first_number - second_number + third_number;
```

In performing the computation starting from left to right, the result will be 50-25, and the final answer is 25. Then include the third number (15). The total will be 40. But what if this is not your intention? Let's say you like to get the sum of the second_number and the third_number, and then subtract the result from the first_number. Hence, that will be 25 + 15 = 40. Then, subtract this from the first_number, which is 50. The answer would be 10.

To make certain that the program is performing what you intend, you must use rounded brackets. Hence, the first computation will look like this:

```
answer = (first_number - second_number) + third_number;
```

Your coding area should look like this:

```
public static void main(String[] args) {
    int first_number, second_number, third_number, answer;

    first_number = 50;
    second_number = 25;
    third_number = 15;
    answer = (first_number - second_number) + third_number;

    System.out.println( " Total = " + answer );
}
```

This is the second computation:

```
answer = first_number - (second_number + third_number);
```

Meanwhile, the code area is this:

```

public static void main(String[] args) {
    int first_number, second_number, third_number, answer;

    first_number = 50;
    second_number = 25;
    third_number = 15;
    answer = first_number - (second_number + third_number);

    System.out.println( " Total = " + answer );
}

```

Next, we'll do some exercises for addition and multiplication.

Change the operators into addition symbol and asterisk sign:

answer = first_number + second_number * third_number;

Get rid of all the rounded brackets before running the program. Without the brackets, it is common to guess that Java will do the calculation starting at left to right. Hence, you would think that it will add the first_number to the second_number to get 75. Then it will multiply the answer to the third_number, which is 15. Hence, the answer will be 1125. Run the program, and the answer you will get is only 425. The answer is different from what we expect.

Operator Precedence is the reason why Java yielded a different result. Java prioritizes some operators than other operators. It prioritizes multiplication as than addition, hence it is performing multiplication first. It will then perform the addition. Hence, Java is performing the code below:

answer = first_number + (second_number * third_number);

Within the added rounded brackets, the second_number is multiplied by the third_number. This total will then be added on top of the first_number. Hence, 25 multiplied by 15 is 375. Add 50, and you will get 425.

If you prefer it the alternative way, be sure to instruct the Java by adding the rounded brackets:

answer = (first_number + second_number) * third_number;

Aside from multiplication, division is regarded as a more important operator for Java. The program will do the division first before doing the subtraction or addition. Try changing

the answer line:

```
answer = first_number + second_number / third_number;
```

The result that we will get is 51. Then, add rounded brackets:

```
answer = (first_number + second_number) / third_number;
```

This time, the answer is 5. Hence, if you get rid of the rounded brackets, Java will first do the division and add 50 to the total. Java will not perform calculations from left to right.

Here is the Operator Precedence:

- Multiply and Divide are considered equally, but they are treated as more important compared to Subtraction and Addition.
- Add and Subtract are considered equally, but they are less prioritized compared to multiplication and division.

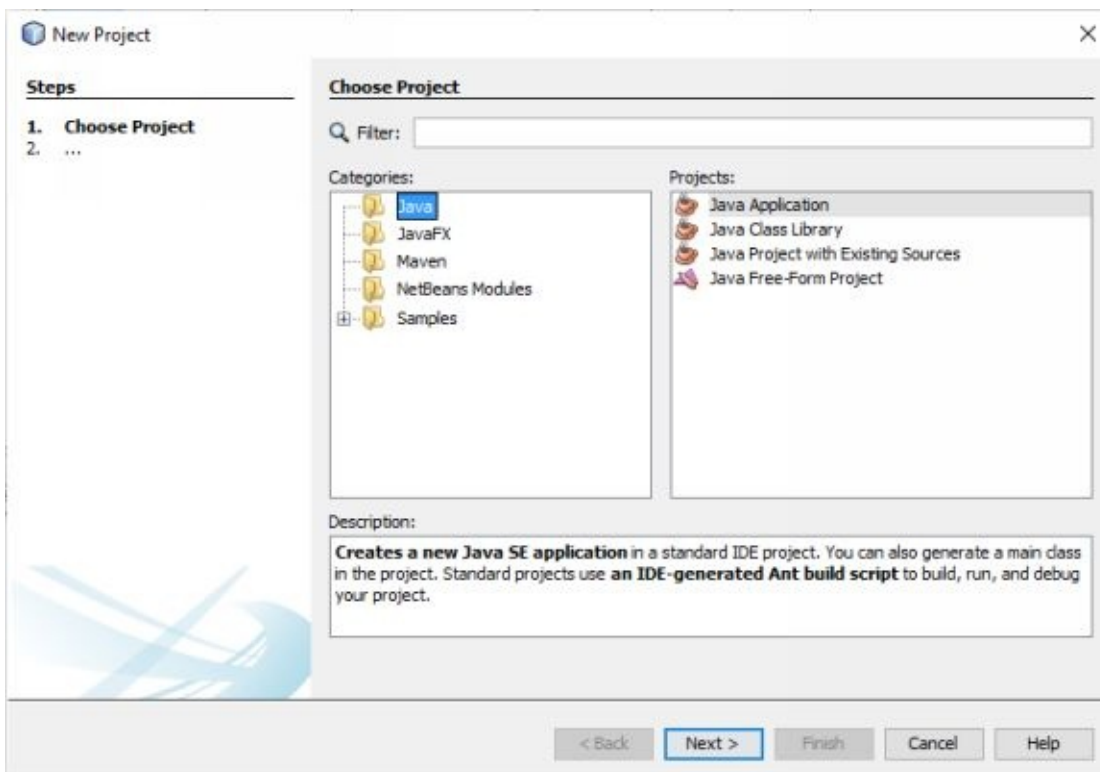
Hence, if Java is providing you an incorrect result, take note that Precedence is essential, and include rounded brackets.

String Variables

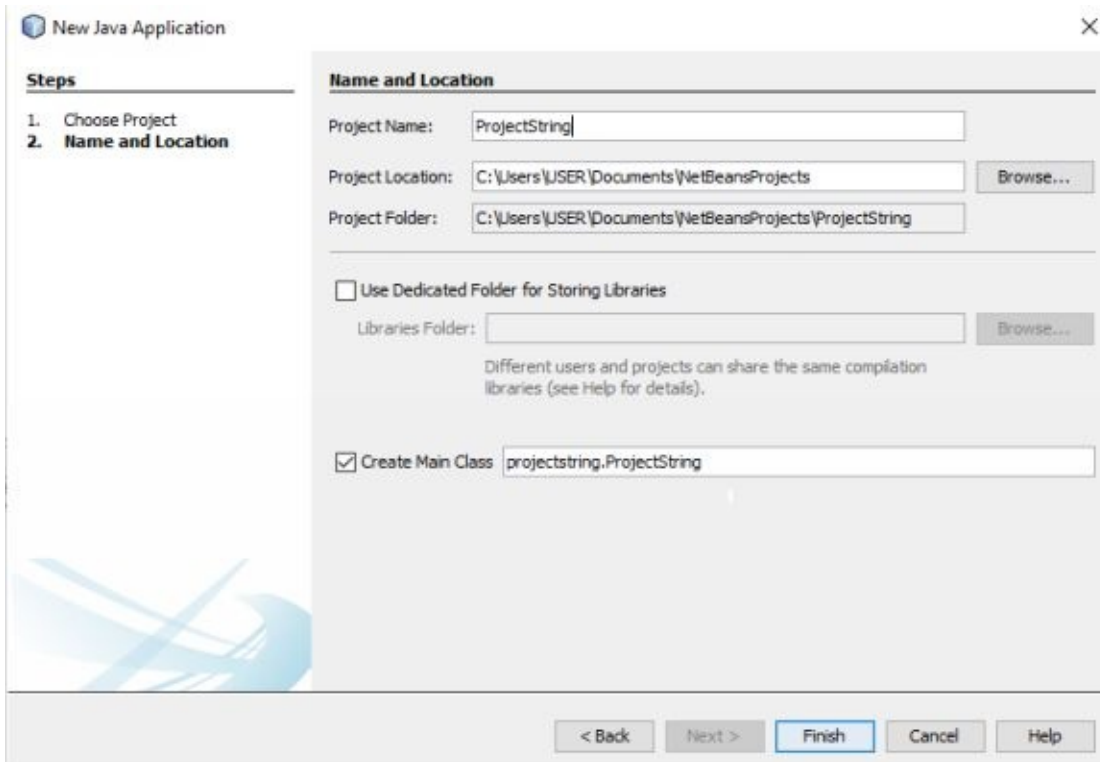
Aside from holding number values, variables could also store text. You can hold one character or many characters. You could use the char variable if you like to hold one character. But commonly, you need to hold several characters.

To store just one character, the char variable is used. Usually, though, you'll like to store more than one character. To do so, you need the string variable type.

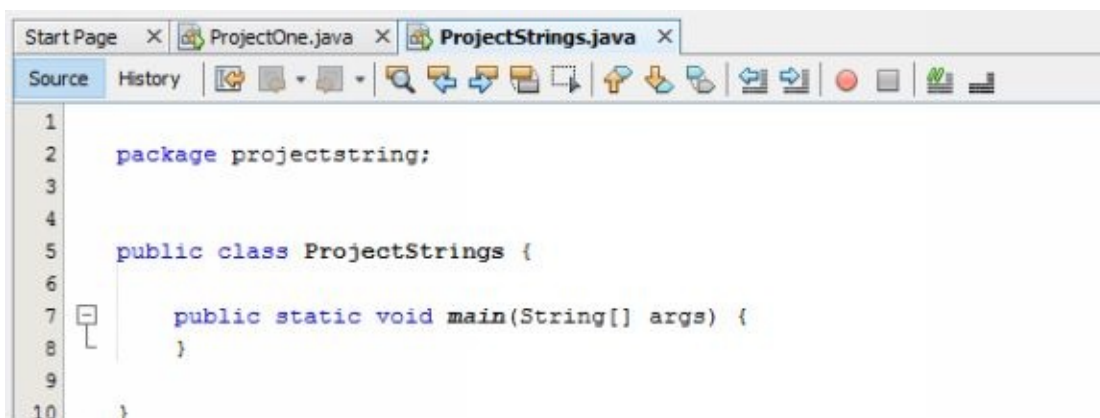
You need to begin a new project for this in NetBeans. **Click File > New Project** at the menu. Once the **NewProject** dialogue box will appear, ensure that you select **Java** and **Java Application**.



Click **Next** and enter **ProjectString** as the name of the project. Be sure that the box for Create Main Class has been selected. Then, delete the Main after the projectstring, and type ProjectStrings instead as you can see below:



Hence the name of the project is **ProjectString**, and the class name is **ProjectStrings**. Hit the Finish button and the coding area should look like this, after deleting all the comments. Take note that the package name is in lower case (projectstring), but the name of the project is **ProjectString**.



In order to create a string variable, you need to use the term String and then the name of the variable. Remember, it should be started with a capital S. Of course, it should be ended using a semicolon.

String first_name;

Choose a value to be stored in the new string variable by adding an equals sign. Following the equals symbol, the text that you like to be held will go between the two sets of double quotes.

```
first_name = "Harry";
```

If you like, you can type them all in a single line:

```
String first_name = "Harry";
```

Add another string variable to store a family name:

```
String family_name = "Potter";
```

In order to print these names, follow it with the `println` method ()

```
System.out.println( first_name + " " + family_name );
```

Within the rounded brackets of `println`, add this:

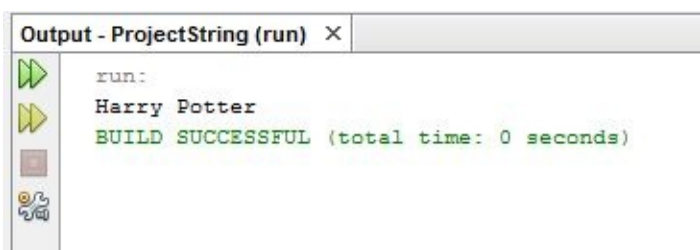
```
first_name + " " + family_name
```

Basically, we are telling Java to print out the contents of the variable known as `first_name`. Then, we have a plus sign and a space. The space is restricted in double quotes. This is important to make certain that the Java will understand that we like to add a space. After this space, be sure to add another plus sign then the `family_name` variable.

Even though this could be a bit complicated, we are only telling Java to print out a first name, a space, then the family name. The code must look like what is shown below:

```
public class ProjectStrings {  
  
    public static void main(String[] args) {  
        String first_name = "Harry";  
        String family_name = "Potter";  
  
        System.out.println( first_name + " " + family_name);  
    }  
}
```

When you run this program, you must see this in the Output screen:



If you just need to hold one character, then the variable that you should use is **char**. Take note that it is in lowercase c. To hold one character, you should use single quotes rather

than double quotes. Take a look at the program below using the **char** variable.

```
public class ProjectStrings {  
    public static void main(String[] args) {  
        char first_name = 'H';  
        char family_name = 'S';  
  
        System.out.println( first_name + " " + family_name);  
    }  
}
```

Try to confine a char variable within double quotes, and NetBeans will flag it with red underline, and will result to incompatible error. However, you can still have a string variable using only one character. But you should use double quotes. Hence, this line is okay:

String first_name = “H”;

But this is not okay:

String first_name = ‘S’;

The first example has double quotes, while the second has single quotes.

There are more to learn about string variables, and we will discuss them in the succeeding chapters. At this point, we need to proceed and get some input.

RECEIVING INPUT

With Java, you can take advantage of the wide range of code libraries that you can use. These codes have been established to perform particular tasks. You just need to determine the specific code that you like to use, then pinpoint a method. A useful class, which could handle user input is known as a **Scanner** class. This can be referenced in the library: **java.util**. You should refer it before you can utilize the class Scanner. This can be performed using the **import** keyword.

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

The line import should be placed on the top of the statement Class:

```
import java.util.Scanner;  
public class ProjectStrings {  
  
}
```

This will instruct the Java program to use a certain class within a specific library (Scanner class) that could be found in the library: java.util.

Next, you need to make the object from the class Scanner. Take note that a class is just a group of codes. It cannot perform something unless you make another object.

To set up a new object Scanner, you need to type this code:

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

Hence, you need to set up a **Scanner** variable instead of a variable **String** or an **int** variable. The name of our variable is input_user. Next to the equals symbol, we have entered the **new** keyword, which can be used to make fresh objects in the class. Take note that this object that we are setting up is sourced out from the class Scanner. Within the rounded brackets, we need to instruct Java that we intend this to be for System.in or System Input.

In order to receive the input from the user, we need to specify an action from the different methods that you can use to your Scanner object. A method is known as next, which could obtain the following text string, which a user could type.


```
String first_name;  
first_name = input_user.next( );
```

After the `input_user`, we follow it with a period. A popup list will appear containing a list of methods that you can use. Choose the method **`next`** and follow it using semicolon at the line end. Also, it is possible to print the text as a guide:

```
String first_name;  
System.out.print("Type your first name here: ");  
first_name = input_user.next( );
```

Take note that we have used **`print`** instead of **`println`** that we have used in our previous code. Take note that the **`print`** will stay on one line, while **`println`** will shift the cursor for another line next to the output.

Next, add another prompt for the surname or family name:

```
String family_name;  
System.out.print("Type your family name here: ");  
family_name = input_user.next( );
```

Take note that this is a similar code, but the program will now hold any information that the user provides into the variable `family_name` rather than the variable `first_name`.

In order to display the input, just add the code below:

```
String full_name;  
full_name = first_name + " " + family_name;  
  
System.out.println("Hello ");
```

Now, we have established `full_name` as another variable `String`. We are holding any information in variables `family_name` and `first_name`. Take note that there is a space in between these variables, and the last line will print it all from the Output screen.

Your code should look like the screenshot below:

```

package projectstring;

import java.util.Scanner;

public class ProjectStrings {

    public static void main(String[] args) {

        Scanner input_user = new Scanner(System.in);

        String first_name;
        System.out.print("Type your first name here:");
        first_name = input_user.next();

        String family_name;
        System.out.print("Type your family name here:");
        family_name = input_user.next();

        String full_name;
        full_name = first_name + " " + family_name;

        System.out.print( "Hello, ");

        System.out.println( first_name + " " + family_name);
    }
}

```

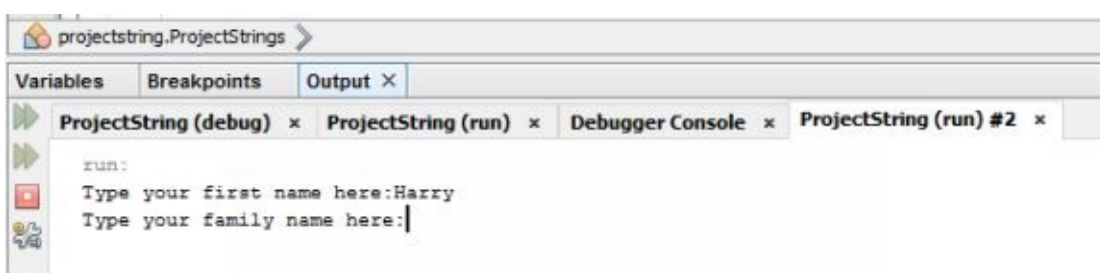
When you run the program, the Output screen should look like this:



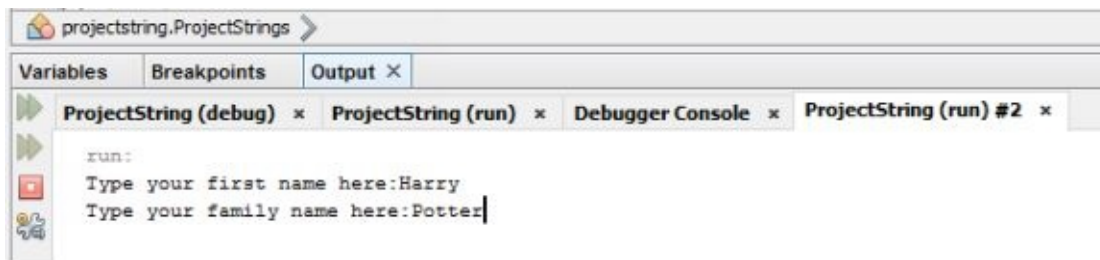
At this point, Java is waiting for you to type in something. It will not proceed until you press the Enter button in your keyboard. Just click the left mouse button next to the “Type your first name here:” and a cursor will blink. Enter your first name, and press the Enter button.

After pressing the Enter button, the program will store the information you provide and keep it within the name variable after the equals symbol. In our code, this refers to the `first_name` variable.

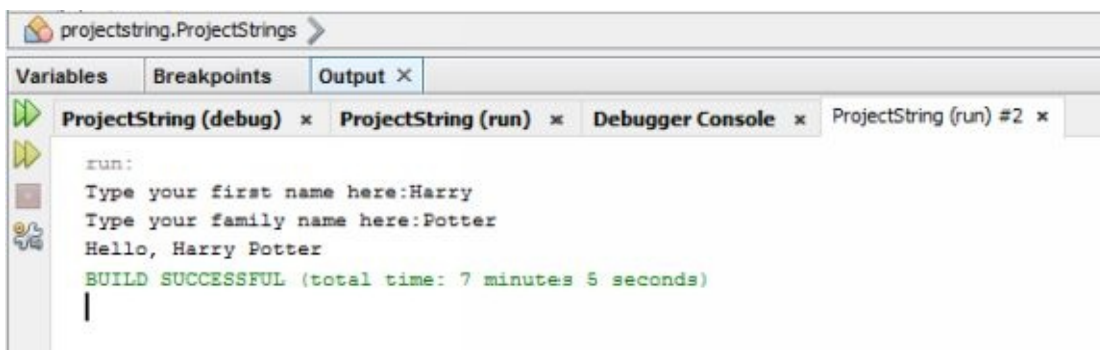
After providing your first name, the program will proceed to the following code line:



Provide a surname, and press again the Enter button:



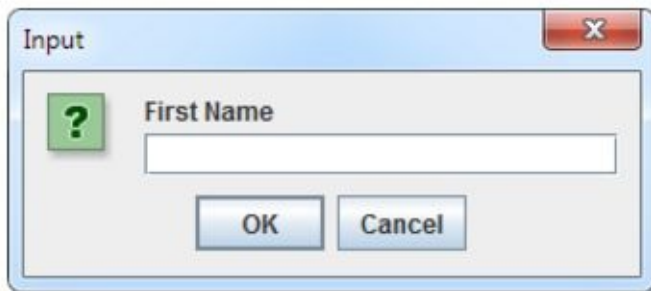
We have now completed the input from the user, and the program will start processing the code, which is the combination of the two names. The final output must look like this:



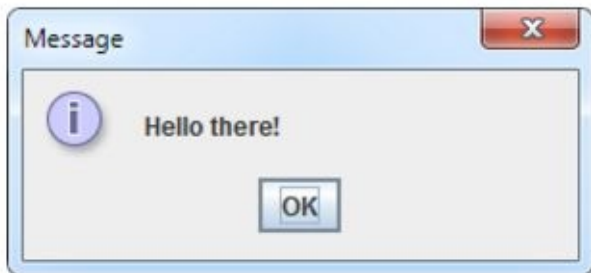
Now, you have learned how to use the class Scanner to obtain information from the program user. Any information provided by the user will be held in variables, and the result will be printed in the Output screen.

THE OPTION PANES

The JOptionPane class is another valuable class for obtaining input from the user and showing results. You can find this in the library: javax.swing. This class will let you display attractive input boxes such as this one:



Or a message box such as this:



We can use our ProjectStrings code and add options type panes. It is important first to refer the library that we need. Enter this into the code:

```
import javax.swing.JOptionPane;
```

This will instruct program that you need the class JOptionPane, situated in the library javax.swing.

If you like, we could begin another project. I trust, by now you know already how to start a project. Be sure to turn the class name to the name of your own choice. We will use the name class DisplayPanels for this one, and the name of the package is inputuser.

Type in new import line in the new project. The coding screen must appear like this:

```
package inputuser;
import javax.swing.JOptionPane;

public class DisplayPanes {

    public static void main(String[] args) {

    }

}
```

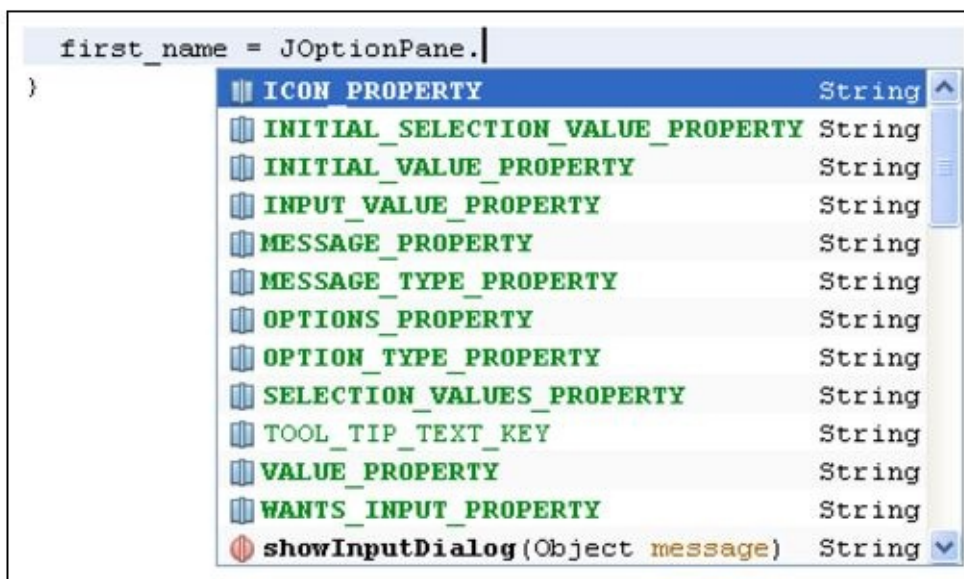
Notice that there is a wavy yellow line in the import code because we are not still using any class yet. Once we do, it will just go away, so don't worry about it.

In order to create a box for input, which the user could provide information, you can use the method **showInputDialog** of the JOptionPane. Just like in the previous sections, we will handle the input directly into the variable first name. Add the following line below the main method so you can see how this one works:

String first_name;

first_name = JOptionPane.showInputDialog("First Name");

Once you enter a period after the JOptionPane, a popup list will appear like this:



Choose the showInputDialog. Within the rounded brackets of the showInputDialog type, enter the message, which you like to be displayed on top of the text box for input. We have chosen "First Name". Similar to strings, this should be confined inside the double quotes.

Type the code below, so we could obtain the family name of the user:

```
String family_name;  
family_name = JOptionPane.showInputDialog("Family Name");
```

Combine these together, and include a message:

```
String full_name;  
full_name = "Hello" + first_name + " " + family_name;
```

In order to show the message in a display box, include this line:

```
JOptionPane.showMessageDialog( null, full_name );
```

At this point, you need to choose the showMessageDialog. Within the rounded brackets, you need to enter the null word. This keyword signifies that the box for message should not be connected to any aspect of the program. The comma should be followed by a text that you like to be shown from the message prompt. Your code must appear like:

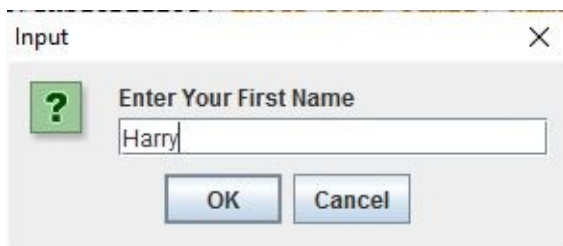
```
public class DisplayPanels {  
  
    public static void main(String[] args) {  
  
        String first_name;  
        first_name=JOptionPane.showInputDialog("Enter Your First Name");  
  
        String family_name;  
        family_name = JOptionPane.showInputDialog("Enter Your Family Name");  
  
        String full_name;  
        full_name = "Hello, " + first_name + " " + family_name;  
  
        JOptionPane.showMessageDialog(null, full_name);  
        System.exit(0);  
    }  
}
```

Take note of the end code below:

```
System.exit(0);
```

This codeline will make certain that the box will exit. It will also tidy things up as it could get rid of the objects from the memory.

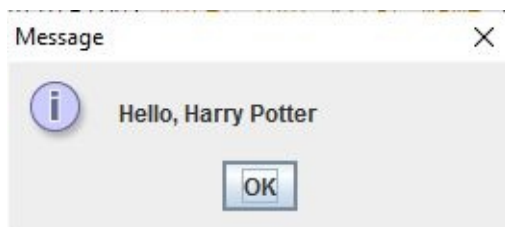
Once you run the code, you will see the input box for First Name. Enter a text and click OK.



Once the input box for Family Name pops up, enter a name and hit OK:



This box will appear after clicking OK.



Now you have learned how to create display boxes. But this is just the tip of the iceberg as you can do far greater things with Java. The next chapter will discuss all about Control Flow.

Chapter 4 – Control Flow

The programming that you have learned at this point is known as sequential programming, which signifies that the code is initiated from up to down. This is quite linear, as each code line will be interpreted, beginning on the first line that you have entered to the end of the last line.

But that is very primitive programming. You need a code to be initiated once specific conditions are met. For instance, you may need one message to be shown if a user is a male and a different message if the user is a female. You need to control the program flow. This could be done through conditional logic.

Conditional logic is primarily about the word “IF.” IF user is a male, then show this message. IF user is female, then show this message. Luckily, it is quite easy to use conditional logic in Java programming. The first conditional logic that we will learn is IF statements.

IF STATEMENTS

In programming, it is quite common to run a code if certain conditions are met. That is why IF Statements are developed. In Java, the structure of the IF statement is this:

```
if ( Statement ) {  
}
```

You begin with the lowercase word if followed by a pair of rounded brackets. Curvy brackets are then used to group a bunch of code. This code is the only code you like to run once the IF condition has been fulfilled. The condition itself must be confined between the rounded brackets:

```
if ( user < 21 ) {  
}
```

This condition states that IF user is less than 21. But rather than using less than, we could use the less than sign (<). If the user is less than 21 years old, then you need to perform a certain action such as to show a message:

```
if ( user < 21 ) {  
    //DISPLAY MESSAGE  
}
```

When the user is not less than 21, then the code within the curvy brackets should proceed, and the program will continue to run until the last code line. Anything you include the curvy brackets will only take effect IF the condition has been fulfilled, and this condition is confined within the rounded brackets.

Another symbol that we can use is the greater than symbol (>). The IF statement above could be changed a bit to check if the user is greater than 21.

```
if ( user > 21 ) {  
    //DISPLAY MESSAGE  
}
```

The only change we have done so far in the code is to change the less than symbol (<) to greater than symbol (>). The IF statement will now check if the user is greater than 21.

However, this code will not check for users who are specifically 21 and not those who are greater than 21. If you need to check for users who are 21 or greater, you can include the equals symbol. Thus:

```
if ( user >= 21 ) {  
    //DISPLAY MESSAGE  
}
```

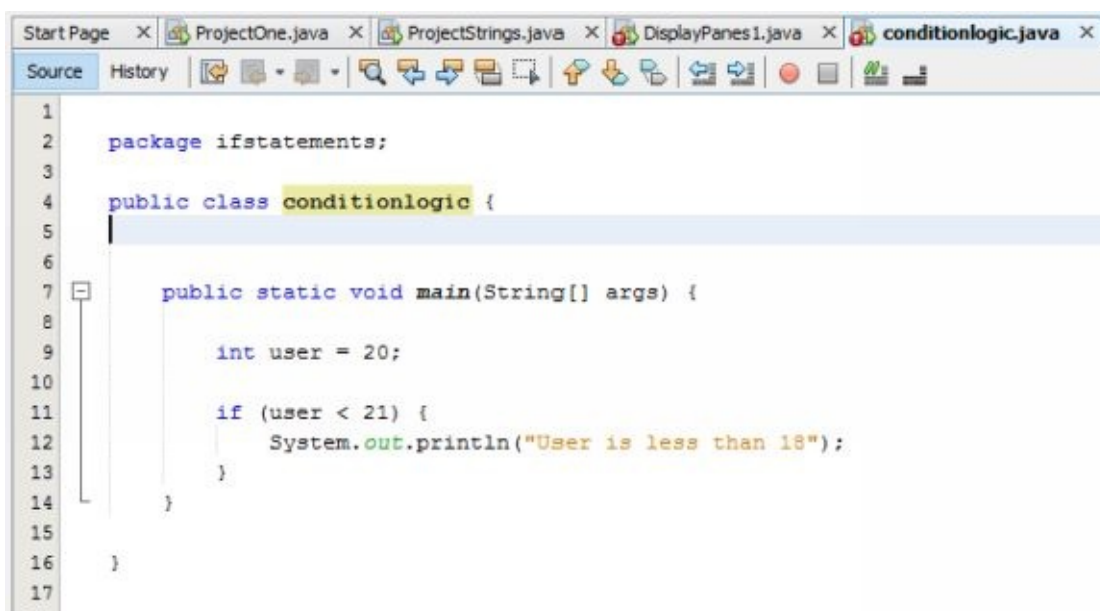
Similarly, you can check for the less than or equal to sign in a similar manner:

```
if ( user <= 21 ) {  
    //DISPLAY MESSAGE  
}
```

The code above contains the less than sign (<) and the equals symbol (=).

Now, we can try this code in a basic program.

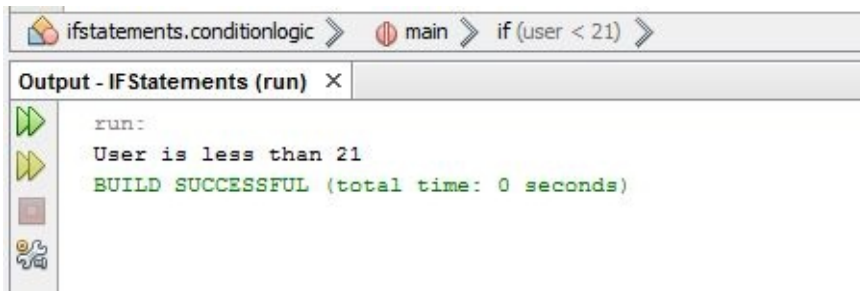
Begin a new project. You can name the package and class name anything you like. We have named the package **IFStatements** while the class is **conditionlogic**.

A screenshot of an IDE window showing a Java file named 'conditionlogic.java'. The code defines a package 'ifstatements' and a public class 'conditionlogic'. Inside the class, there is a 'main' method that initializes a variable 'user' to 20 and checks if it is less than 21. If true, it prints 'User is less than 18'.

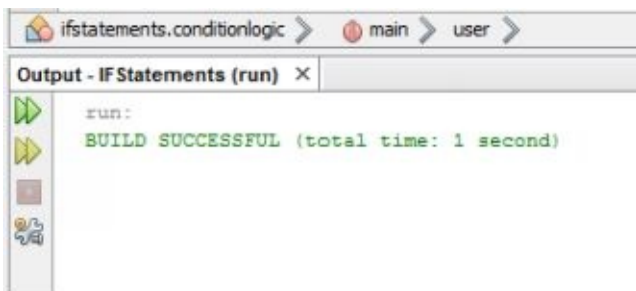
```
1 package ifstatements;  
2  
3  
4 public class conditionlogic {  
5  
6  
7     public static void main(String[] args) {  
8  
9         int user = 20;  
10  
11         if (user < 21) {  
12             System.out.println("User is less than 18");  
13         }  
14     }  
15  
16 }  
17
```

We have established an integer variable, and defined the value 20. The IF statement will check for “less than 21”. Hence, the message within the curly brackets will be displayed.

You can now run the program. If you will notice, NetBeans runs the program in box text in the Projects screen and not the code you have shown. In order to run the code in the coding screen, just right click in the code area. A popup list will appear. Choose Run File. You can see this in the Output screen:



Next, change the user variable value from 20 to 21. Run it, and you must see this:



The program is running fine, and it has no error messages. However, there 's no print out, because the message within the curly brackets of the IF Statement, which is checking values less than 21. The condition is not fulfilled so Java is ignoring the curly brackets and proceeds with the rest of the code.

Now, try replacing < with < = symbols. Also change the message that is suitable such as “ user is less than or equal to 21 ” . Run the program. What message do you see?

Next, you can change the user value to 22, and run the program. Is the message still there?

You can also have several IF Statements in the code. Try this one:

```

public class conditionlogic {

    public static void main(String[] args) {

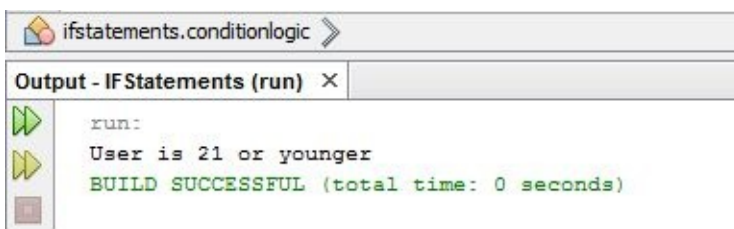
        int user = 21;

        if (user <= 21) {
            System.out.println("User is 21 or younger");
        }

        if (user > 21) {
            System.out.println("User is greater than 21");
        }
    }
}

```

At this point, we have two IF Statements. The first checks for values that are less than or equal to 21. The second IF statement checks for values that are greater than 21. Once you run the code with the value 21 or less for the variable user, the Output will be:



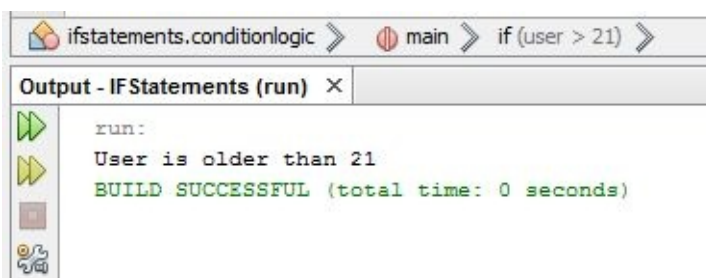
The screenshot shows an IDE window titled 'ifstatements.conditionlogic'. Below the title bar is a tab labeled 'Output - IFStatements (run) X'. The output area contains the following text:

```

run:
User is 21 or younger
BUILD SUCCESSFUL (total time: 0 seconds)

```

Change the value of the variable user to 22 and the Output will be like this:



The screenshot shows an IDE window titled 'ifstatements.conditionlogic'. Below the title bar is a tab labeled 'Output - IFStatements (run) X'. The breadcrumb navigation shows 'main > if (user > 21) >'. The output area contains the following text:

```

run:
User is older than 21
BUILD SUCCESSFUL (total time: 0 seconds)

```

Hence, only one of the IF statements will print out a message in the display. And the message will depend on the value provided in the user variable.

IF...ELSE STATEMENTS

Rather than using two IF Statements, it is easier to use an IF...Else Statement. Below is the format of an IF...Else statement:

```
    if ( condition ) {  
}  
else {  
}
```

It begins with the “if” code and then rounded brackets containing the condition that you like to check. Remember to use curvy brackets to chunk off the several options. The second option goes after “else” and confined with its own curvy brackets. Again, here is the code, which tests the age of the user:

```
public static void main(String[] args) {  
  
    int user = 22;  
  
    if (user <= 21) {  
        System.out.println("User is 21 or younger");  
    }  
    else {  
        System.out.println("User is older than 21");  
    }  
}
```

Now we only have two options: either the user is 21 or less than or the user is greater than that. Change your code so it will resemble the screenshot above, and run it. You will find that the first message will be displayed. You can change the value of the user variable to 22 and run it again. The text between the curvy brackets after the ELSE will be shown in the Output Screen.

IF...ELSE IF

It is possible to check for several options. For instance, let's say that we like to check for more age ranges such as 20 to 40, and 41 and beyond? For more than two options, we can use the If...Else If statement. The format of the If...Else If statement is this:

```
    if ( condition_1 ) {  
    }  
    else if ( condition_2 ) {  
    }  
    else {  
    }
```

This is the new part:

```
    else if ( condition_2 ) {  
    }
```

Hence, the first IF checks for the condition_1 (21 or under, for instance). Then the else if followed by rounded brackets. Condition_2 is confined inside the new rounded brackets. Any information not confined by the first 2 conditions will be confined in the else if. Remember, your code should be chunked off through the curvy brackets, with every if, else if, or else, with its own set of curvy brackets. Delete one bracket, and an error message will show.

Before running this code, it is important first to learn other operators for conditional logic. So far, we have used the following:

Less Than (<)

Greater Than (>)

Less Than or Equal To (<=)

Greater Than or Equal To (>=)

Here are four more operators that we can use:

And (&&)

Not (!)

Or (||)

Has the value (==)

The And operator is composed of two ampersands (&&), and can be used to check for more than one condition simultaneously. We could use this to check for two ranges of ages:

```
else if ( user > 21 && user < 60)
```

At this point, we intent to test if the user is greater than 21 but less than 60. Take note that we are trying to test what is contained in the user variable. The first condition is “greater than 21” (user > 21), and the second condition is “less than 60” (user < 60). Between these two, we have the && operator. Hence, the whole code line means “else if user is greater than 21 AND user is less than 60.”

We will try the other conditional operators in the later sections. First, let’s try a new code:

```
public static void main(String[] args) {  
  
    int user = 21;  
  
    if (user <= 21) {  
        System.out.println("User is 21 or younger");  
    }  
    else if (user > 21 && user < 60) {  
        System.out.println("User is between 22 and 59");  
    }  
    else {  
        System.out.println("User is older than 60");  
    }  
}
```

Run the program and test it. By now, you should have developed the skill to guess exactly what will the print out will be before you run the code. Since we have a value 21 for the variable user the message confined between the curvy brackets of else will show the Output screen.

Change the value of the user variable from 21 to 50. The display for the else section of the code must now show.

You have the freedom to add as many else if sections. Let’s say that we like to test if the user was either 50 or 55. Now we can use the other operators. We can test if the variable for the user has value 50 or has value 55.

else if (user == 50 || user == 55)

Use two equal signs to check if the user variable has value something. Take note that there should be no space between the equal signs. Java will check for this value only. Since we like to check for the user being 55 also, we could include another condition within the same rounded brackets: user==55. This will instruct Java to check if the user variable has value 55. Notice that between these two conditions, we have added the operator OR. Remember, there should be no space between the two characters. The entire line code means “Else if the user has value 55 OR the user has value 55”.

Below is the code with the added new else part.

```
public static void main(String[] args) {  
  
    int user = 50;  
  
    if (user <= 21) {  
        System.out.println("User is 21 or younger");  
    }  
    else if (user > 21 && user < 60) {  
        System.out.println("User is between 22 and 59");  
    }  
    else if (user == 50 || user == 55) {  
        System.out.println("User is either 50 or 55");  
    }  
    else {  
        System.out.println("User is older than 60");  
    }  
}
```

Now, change the user variable value to 55 and run the code. Next, change it to 60 and run the code. In either case, the new message should be displayed.

Try it out for yourself. Change the value of the user variable to 45 and run your code. Then change it to 50 and re-run the code. In both cases the new message should display.

It can be tricky to use the different conditional operators. However, we are just checking a variable for a certain condition. This is just basically about choosing the suitable conditional operator for the function you like.

NESTED STATEMENTS

It is possible to nest conditional logic statements. Nesting means confining one statement between another. For instance, if we like to check if a user is less than 21 years old, but greater than 18 years old, we can display a different message for the over 18. We can start with the first IF statement:

```
if ( user < 21 ) {  
System.out.println( "21 or younger");  
}
```

In order to test for over 18, you could add a second statement within the first one. The structure is still the same:

```
if ( user < 22 ) {  
if ( user > 18 && user < 22 ) {  
System.out.println( "You are 18 or 22");  
}  
}
```

Hence, the first statement covers the variable user if it is lower than 22. The second statement will confine the user variable down for value more than 18 and lower than 22. To print out separate messages, you can use the If...else statement rather than the If statement as shown above.

```
if ( user < 22 ) {  
if ( user > 18 && user < 22 ) {  
System.out.println( "You are 18 or 22");  
}  
else {  
System.out.println( "18 or younger");  
}  
}
```

Take note of the location of the curvy brackets: misplace one bracket and your program will not run.

The nested IF statements could be tricky. However, we are trying to narrow down the

options.

BOOLEAN VALUES IN JAVA PROGRAMMING

A Boolean value is one of the two options: 1 or 0, yes or no, true or false. In Java programming, there's a variable type for Boolean:

```
boolean user = true;
```

Hence, rather than entering string or double or int, you can use boolean (with a small b). After the variable name, you can define value true or false. Take note that the assignment operator refers to one equals sign (=). If you need to test if a variable “has value” something, you can use double equals sign (==).

You can try this basic code:

```
boolean user = true;

if ( user == true) {
    System.out.println(“it’s true”);
}
else {
    System.out.println(“it’s false”);
}
```

Hence, the first statement tests if the value of user variable is true. The else part tests if the value is false. There is no need to express “else if (user==false)”, because if it is not true, then it is false. Therefore, we can just use else. There are only two options when it comes to boolean values.

The other operator on our option is the NOT operator. We could use this with boolean value. Observe this code:

```
boolean user = true;

if ( !user ) {
    System.out.println(“it’s flase”);
}
else {
    System.out.println(“it’s true”);
}
```

This is almost similar to the other boolean value, but for this one:

```
if ( !user ) {
```

But at this point, we can use the operator NOT prior to the user variable. The operator NOT is one exclamation point (!) and it is placed prior to the variable that you are trying to check. This is checking for negation that means that it is checking for the opposite of the actual value. Since the variable user has been established to be true, then the operator ! will check for false values. If the user has been established to false, then the operator ! will check for true values. To put it simply, if a value is NOT true, then what will it be? Or if a value is NOT false, then what it is?

JAVA SWITCH STATEMENTS

Switch statements are also used to control the flow of programs. These statements provides you the choice to check for a range of values for the variables. You can use them rather than the lengthy, sophisticated **if...else if** statements. This is the format of the switch statement:

```
switch ( variable_2_check) {  
case value:  
code_here;  
break;  
case value:  
code_here;  
break;  
default:  
values_not_caught_above;  
}
```

Begin with the word switch and follow it with a pair of rounded brackets. The variable that you like to test should be confined between the rounded brackets of the switch. Then, add a pair of curvy brackets. The other sections of the switch statement should all go within the two curvy brackets. You should use the word case for each value that you like to test. You can then test this value:

case value:

The case value should be followed by a colon. Then, you can place what you need to see if they can match values. This is the code that you like to run. The **break** keyword should be used to break out every case of the statement.

It is optional to include the standard value. You can place it if there are other values, which could be stored in variable, but that you have not tested anywhere in the statement.

If you are confused by this, you can try this code. You can begin a new project for this, or you can add a new comment. A fast way to comment out the code within NetBeans is to click the comment icon found at the toolbar. The first step is to highlight the code that you like to comment out. The next step is to click the icon for comment.



Now here is the code:

```
public static void main(String[] args) {  
  
    int user = 21;  
  
    switch ( user ) {  
        case 21:  
            System.out.println("You are 21");  
            break;  
        case 22:  
            System.out.println("You are 22");  
            break;  
        case 23:  
            System.out.println("You are 23");  
            break;  
        default:  
            System.out.println("You are not 21, 22 or 23");  
    }  
}
```

This code will define a value that you need to check. We have now established an integer variable and referred to it as **user**. We have defined the value to 18. The statement will test the variable user and check the value. This will then pass through every statement. Once it finds one that agrees with, the code will pause and run for that case and will then break the statement.

You can try running this program. Type in the different values for the variable user and see what will happen.

Unfortunately, there is no way to check for a values range after the case. Hence, you can check for a single value only. Therefore, it is not possible to do this code:

case (user <= 21):

Meanwhile, it is possible to perform this:

case 1: case 2: case 3: case 4:

The test above will check for the range of values from one to four. However, you need to

specify every value. Be sure to take note of the placement of all the case and colons.

JAVA LOOPS

So far, the programming that we have been doing now is sequential. It runs from top to bottom with each code line being run, except when you include a section that tells Java not to read that code.

In the previous section, using IF statements to chunk off code areas is a method to instruct Java not to read each line.

Another method to disrupt the flow from top to bottom is through loops. In programming, a loop is one that forces the program to run code lines again and again.

For instance, let's say that we intend to add the numbers 11 to 20. You can easily compute that in Java such as this:

```
int addition = 11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20;
```

However, using this method to add up a long string of numbers such as 1 to 10,000 is time consuming. An easier method is to use a loop to return the line code again and again until you have reached 10,000. You can then exit the loop and move on.

A common type of loops is called For Loops. Below is the format of this LOOP:

```
for ( start_value; end_value; increment_number ) {  
    //INSERT_CODE_HERE  
}
```

The keyword for is followed by rounded brackets containing three elements: the start value, the end value, and the method to obtain from one number to another. This is known as the increment number, and often the value of one. You have the freedom to choose the increment.

Next to the rounded brackets are the curvy brackets, which are employed to chunk off the code that you need to run again and again. Confused? Here is an example code.

Start a new project, and name the project and class anything you like. For this, I have named the project ForLoops and the class JavaLoops. Next, add the code below:

```
package forloops;

public class javaloops {

    public static void main(String[] args) {

        int loopVal;
        int end_value = 21;

        for (loopVal = 0; loopVal < end_value; loopVal++) {

            System.out.println("Loop Value = " + loopVal);

        }

    }

}
```

We can begin by defining an integer variable that we have named as loopVal. The second line defines another int variable, which we can use for the loop's end value, which is set to 21. What we intend to do is to loop around to print out the numbers from zero to 21.

We have the following code within the rounded brackets of the for loop:

loopVal =0; loopVal < end_value; loopVal++

The first chunk instructs Java to start at this value in looping. In this, we are defining zero value to the variable loopVal. This will be used as the first number in the loop. The next chunk employs a conditional logic.

loopVal < end_value

The above line means that “loopVal is less than the end_value”. The for loop will then run and run if the value of the variable loopVal is still lower compared to the value defined in the end_value variable. Java will keep on running over the code confined inside the curly brackets as long as the value of the loopVal is still less than the value of the variable end_value.

Below is the last chunk inside the rounded brackets of the for loop

loopVal++

At this point, we are instructing Java the method we like it to follow to go from the start value in the loopVal to the next number in the series. We intend to count from zero to 20. The next number is 1. The code loopVal++ simply means “add an increment of 1 to the variable value.

Rather than expressing loopkVal++, we could have written this:

loopVal = loopVal + 1

After the loopVal+1, Java will add an increment of 1 to anything that is presently stored in variable loopVal. After it has added one to this value, it will hold the result within variable before the equals symbol. Again, this is the variable loopVal. The outcome is that 1 will keep getting added to the loopVal. This is referred to as incrementing the variable. This is so common in programming that the variable notation signified by two plus symbols (++) was developed for this:

```
int set_number = 0;  
set_number++;
```

The value of set_number is 1 once you run the code. This is the short hand of expressing this:

```
int set_number = 0;  
set_number = set_number + 1;
```

In summary, the for loop refers to this:

Loop Start value: 0

Keep Looping until: Start value is less than 21

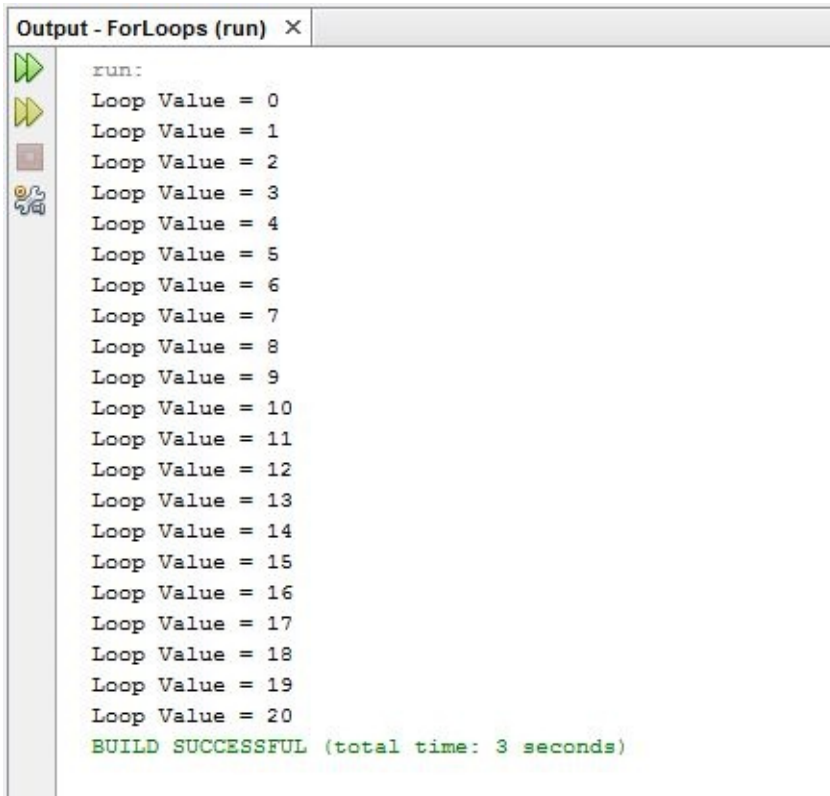
Method to progress the end value: Keep adding 1 to the start value

We have this curvy brackets of the for loop:

System.out.println(“Loop Value = ” + loopVal);

Anything that is presently confined the loopVal variable could be printed out with any message.

Now, run the program and this message should appear in the Output screen:



```
Output - ForLoops (run) X
run:
Loop Value = 0
Loop Value = 1
Loop Value = 2
Loop Value = 3
Loop Value = 4
Loop Value = 5
Loop Value = 6
Loop Value = 7
Loop Value = 8
Loop Value = 9
Loop Value = 10
Loop Value = 11
Loop Value = 12
Loop Value = 13
Loop Value = 14
Loop Value = 15
Loop Value = 16
Loop Value = 17
Loop Value = 18
Loop Value = 19
Loop Value = 20
BUILD SUCCESSFUL (total time: 3 seconds)
```

Hence, we have confined the program in a loop, and instructed it to run round and round. Every loop, an increment of 1 will be added to the variable loopVal. This loop will keep running again and again while the value of the loopVal is lower than the value defined in the end_value. Anything within the curly brackets of the loops refers to the code that will run again and again. This is the whole idea of the loop: to run the curly bracket code over and over again.

Here is a code that will allow you to add the numbers 0 to 20. Have a go:

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

    int loopVal;
    int end_value = 21;
    int addition = 0;

    for (loopVal = 0; loopVal <end_value; loopVal++) {

        addition = addition + loopVal;

    }

    System.out.println("Total = " + addition);

}
```

The result that must be shown in the Output screen is 210. More or less, the code itself is similar to the past for loop. You have the same set of variables written at the top – loopVal and end_value. Then, we have added another int variable that we have referred to as addition. This will store the value of the sum of 0 to 20.

Within the rounded brackets of the loop, it is also similar for the previous code. We have looped while the value of the loopVal variable is lower than the value defined in the end_value. Meanwhile, we were adding the increment of 1 to the variable loopVal every time the loop runs (loopVal++).

Inside the curly brackets, we now have a single code line:

addition = addition + loopVal;

This one code line adds the numbers from 0 to 20. If you are still not clear on how this one works, you can start next to the equals symbol:

addition + loopVal;

The first round of the loop defined the variable addition, which is storing 0 as its value. Meanwhile, the loopVal variable, is still storing value 1 (its beginning value). The program will add 0 to 1, and will keep the answer to the variable before the equals symbol. Remember, this is the variable addition. Any value that was stored in the previous codes will be eliminated and will be replaced with the new value.

The second round of the loop, the variable values will be new agains.

addition (1) + loopVal (2);

Of course, the answer is 3. Hence, this new value will be held to the variable before the equals symbol.

These are now then values of the loop for the third round.

addition (3) + loopVal (3);

The program will add values and will store the result to the variable before the equals symbol. This will keep running again and again until the loop ends. The final answer is 210.

Take note that the line for print is located outside the for loop, following the final curly bracket for the loop.

WHILE LOOPS

Another kind of loop, which you can use in Java is known as the while loop. You will find that while loops are easier to learn compared to for loops. Here is the format of a while loop:

```
while ( condition ) {  
    }  

```

We will begin with the keyword while in small caps. The condition that you like to check for will be confined inside the rounded brackets. This will be followed by a pair of curly brackets as well as the code that you like to run. Here is a while loop, which prints out a message. Try this one:

```
int loopVal = 0;  
  
while ( loopVal < 10) {  
    System.out.println("Display Some Message");  
    loopVal++;  
}
```

The condition to check is confined between the rounded brackets. We intend to keep looping until the value stored in the loopVal variable is lower than 10. Within the curly brackets, our program will display a line of message. We can then increment the value of the variable loopVal. Without defining this, we will have an infinite loop, because there's no way for the loopVal variable to obtain a value beyond its start value of zero.

Even though we have used a counter to proceed to the end condition, while loops are ideal to use if you really don't need a counting value, but instead just a value to check. For instance, it is fine to keep the loop while the user is still not pressing any key. This is helpful in programming games. A specific key could be pressed to exit the while loop. This is better known as the game loop, and therefore could be the game itself. Another good example is looping around the text file while the program has not yet reached the end of the file.

DO...WHILE

Do...While is related to the while loop. The format is this:


```
int loopVal = 0;  
do {  
System.out.println(“Display Some Message”);  
loopVal++;  
}  
while ( loopVal < 10 );
```

Take note that the program will loop again and again until you meet the end condition. At this point, the while part is located at the bottom. However, the condition is similar: just keep on looping while the value of the loopVal variable is less than 10.

The main distinction between the two is that the code inside the curly brackets for do...while could be run at least once. Using the while loop, the condition can easily be fulfilled. The program will just exit the loop, and not even run the code inside the curly bracket. To check this, you can try first the while loop. You can change the value of the variable loopVal to 10, and then run. You will find that the message doesn't get printed out. Now, you can try the do loop with the defined value of 5 for loopVal. The message will be displayed once, and then Java will exit out the loop.

In the next chapter, we will learn more about Java Arrays.

Chapter 5 – Java Arrays

A programming concept that you need to learn in order to effectively code is the array. It is a crucial concept, so it deserves its own chapter.

WHAT IS A JAVA ARRAY?

At this point, we have been coding with variables that store single values. The string variables we have set up only holds a long string of text, while the int variables have stored only a single number. In order to hold more than one value at the same time, we need to use an array. This is similar to the list of items. An array is similar to the columns in your spreadsheet software. A spreadsheet can be composed of a single column or several columns. The data stored in a single-list array may look like this:

	Array_Values
0	10
1	14
2	36
3	27
4	43
5	18

Similar to a spreadsheet, arrays include a position number for every row. The positions in an array begin at zero and will sequentially increase. Every position in the array could then store a value. In the screenshot above the position array 0 is storing value 10, position array 1 is holding 14, and array position 2 is holding value 36, and so on and so forth.

In order to establish an array of number similar to this screenshot, we need to instruct Java the type of data will be in the array (boolean values, strings, integers, and more). Then, you need to say how many array positions. The format is like this:

```
int[ ] numArray;
```

The main distinction between defining a normal integer variable and an array is a pair of square brackets right after the type of data. The square brackets are sufficient enough to command Java that you like to establish an array. The name of our array above is numArray. Similar to regular variables, you have the freedom to name the variable anyway you like.

However, this will instruct Java that you intend to establish an array integer. It will not say how many positions the array must store. In order to do this, we need to define a new object array:

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

We will begin with the name of the array and then equals sign. Next to the equals symbol, we add the keyword new, and then the type of data. Next to the type of data is a pair of square brackets. Within these brackets, we need to indicate the array size. The size is how many positions the array must store.

We can write them all in a single line:

```
int[ ] numArray = new int[6];
```

We are commanding Java to establish an array with five positions in it. Once this line is run, Java can specify default array values. Since we have already specified an array integer, the default values for the five positions will be 0.

In order to define values to the different positions in an array, we can do it in the regular method:

```
numArray[0] = 5;
```

In this line, we are assigning the value of 5 to position 0 in the array named numArray. Remember, we are using the square brackets to refer to every position. If you intend to define value 25 to position 1, the code should be like this:

```
numArray[1] = 25;
```

Meanwhile, we can assign value 50 to array position 2 through this:

```
numArray[2] = 50;
```

Take note, since arrays begin at 0, the third position in the array includes the number 2 index.

If you know values that you like to include in the array, you can write them up like this:

```
int[ ] numArray = { 1, 2, 3, 4 };
```

This manner of establishing an array uses curly brackets next to the equal symbol. Within the curly brackets, we can specify values that each array can store. The first value could be position 0 while the second value position 1, etc. Remember, you should add square brackets next to the int, but not on the keyword new or the cycle of the type of data and square brackets. However, this is just for data types of char values, string, and int. Otherwise, you will need to use the keyword new. Hence, you can write this:

```
String[ ] stringsArray = {"Harry", "Ron", "Hermione", "Snape" };
```

But not this:

```
boolean[ ] boolsArray = {true, false, true, false};
```

In order to establish an array boolean, we still need to use the keyword new:

```
boolean[ ] boolsArray= new boolean[ ] {true, false, true, false};
```

To move at values stored in the array, we can write the array name and include the position array in square brackets such as this:

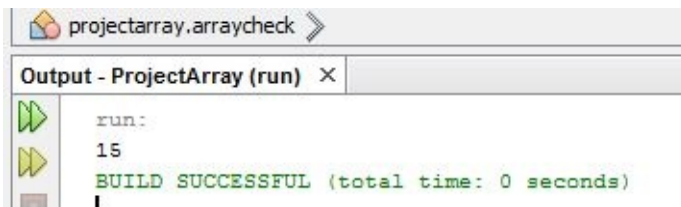
```
System.out.println( numArray[3] );
```

This code will display whatever value is stored at array position 3 in the array named as numArray. Let's do some exercise.

Begin a new project and name it anything you like. Be sure to choose a new name for the Class. Enter the code below into the new Main method:

```
public static void main(String[] args) {  
  
    int[] numArray;  
  
    numArray = new int[5];  
  
    numArray[0] = 5;  
    numArray[1] = 10;  
    numArray[2] = 15;  
    numArray[3] = 20;  
    numArray[4] = 25;  
  
    System.out.println( numArray[2] );  
}
```

Once we run the program, the Output screen should look like this:



Try changing the array position in the number within the line print from 2 to 4, and 25 should be displayed instead.

LOOPS AND ARRAYS

Arrays naturally have their own loops. In the previous section, the code below is used to assign values to array positions:

```
numArray[0] = 5;
```

However, this is not recommended if you have a long list of numbers that you need to assign to an array. For example, let's say we need to develop a lottery program and we need to assign the numbers 1 to 60 to positions in an array. Rather than encoding an exhaustive list of positions with their values, we can instead use a loop. Below is a code to work it out:

```
package projectarray;

public class arraycheck {

    public static void main(String[] args) {

        int[] lotto_nums = new int[60];
        int i;

        for (i=0; i < lotto_nums.length; i++) {
            lotto_nums[i] = i + 1;
            System.out.println( lotto_nums[i] );
        }

    }

}
```

In this code, we have defined an array to store 60 int values. Then, we added a loop code. Take note of the loop's end condition:

`i < lottery_numbers.length`

Length refers to the property of array objects, which you can use so you can obtain the array size or the number of positions. Hence, this loop will keep running while the variable value in i is lower compared to the array size.

The code below is used to define values for every array position:

```
lottery_numbers[i] = i + 1;
```

Rather than the hard-code value within the square brackets of name array, we use the `i` variable, which increases by value one every time the loop runs. We can then access every array position by utilizing value loop. The value that we are assigning to every position is `i` plus 1. Hence, this is an example of incremental value in a loop but with 1 added. Since the value of the loop starts at 0, this will provide you the numbers 1 to 60.

The added line in the loop will just display the value in every array position. If you like, you can even type a code to mix up the array numbers. When you have mixed up values, you can then get the first six so you can assign them as lotto numbers. Then type another block of code, which compares the number of the users with the assigned winning nums and you will now have a lotto code!

ARRAY SORTING

There are java methods that you can use for array sorting. In order to use this sorting method, it is important first to reference a library known as Arrays. You can do this using the import statement. You can use the numArry program and add the import statement below:

import java.util.Arrays;

The code must look like this screenshot:

```
package projectarray;

import java.util.Arrays;

public class arraycheck {

    public static void main(String[] args) {

        int[] numArray;
        numArray = new int[5];

        numArray[0] = 5;
        numArray[1] = 10;
        numArray[2] = 15;
        numArray[3] = 20;
        numArray[4] = 25;

    }

}
```

After importing the library Arrays, we can use now the sorting method. You should find this easy.

Arrays.sort(numArray);

The first step is to encode the word “Arrays followed by a period. Once you add the period, the program will show a list of options that you can do with arrays. Just enter the word: sort. Within the rounded brackets, place the name of the array that you like to sort. Take note that there is no need to use any square brackets after the name of the array.

And voila! You can now sort the array. Try the code below to see this work:

```
package projectarray;

import java.util.Arrays;

public class arraycheck {

    public static void main(String[] args) {

        int[] numArray;
        numArray = new int[5];

        numArray[0] = 5;
        numArray[1] = 10;
        numArray[2] = 15;
        numArray[3] = 20;
        numArray[4] = 25;

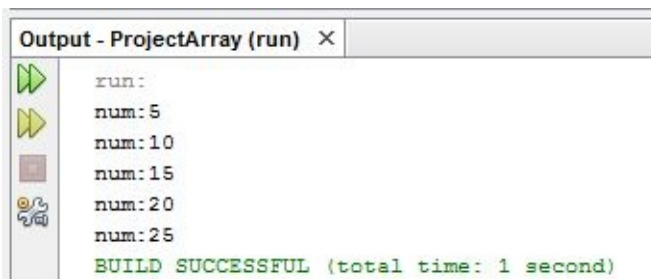
        Arrays.sort(numArray);

        int i;

        for (i=0; i < numArray.length; i++) {
            System.out.println("num:" + numArray[i]);
        }

    }
}
```

The for loop added the end of the line will run again and again to display values within every array position. Once the code has been run, the Output screen should look like this:

The screenshot shows the 'Output - ProjectArray (run)' window in a Java IDE. On the left, there is a vertical toolbar with icons for running (a green play button), stepping through (a yellow play button), debugging (a red square), and other IDE functions. The main area of the window displays the output of the program. It starts with 'run:' followed by five lines of output: 'num:5', 'num:10', 'num:15', 'num:20', and 'num:25'. At the bottom, a green status bar indicates 'BUILD SUCCESSFUL (total time: 1 second)'.

```
run:
num:5
num:10
num:15
num:20
num:25
BUILD SUCCESSFUL (total time: 1 second)
```

As you will see, the program sorted the array in ascending order.

If you like to sort out in descending order, we need to write down the sorting code, or transform the array to int objects then import from the library Collections. Below is the code if you like to sort in the descending order:

```
package projectarray;

import java.util.Arrays;
import java.util.Collections;

public class DescendingSort {

    public static void main(String[] args) {

        int[] numArray;
        numArray = new int[5];

        numArray[0] = 5; numArray[1] = 10; numArray[2] = 15; numArray[3] = 20;
        numArray[4] = 25;

        Integer[] integerArray = new Integer[numArray.length];

        for (int i = 0; i < numArray.length; i++) {
            integerArray[i] = new Integer(numArray[i]);
        }

        Arrays.sort(integerArray, Collections.reverseOrder());

        for (int i = 0; i < numArray.length; i++) {
            System.out.println("num: " + integerArray[i]);
        }
    }
}
```

The code above can be a bit messy, but it can do the job.

STRINGS AND ARRAYS

The next concept to learn is how to put text strings within arrays. You can also do this using the similar method of processing integers:

```
String[ ] stringArray = new String[5] ;  
stringArray[0] = "What";  
stringArray[1] = "does";  
stringArray[2] = "the";  
stringArray[3] = "fox";  
stringArray[4] = "say";
```

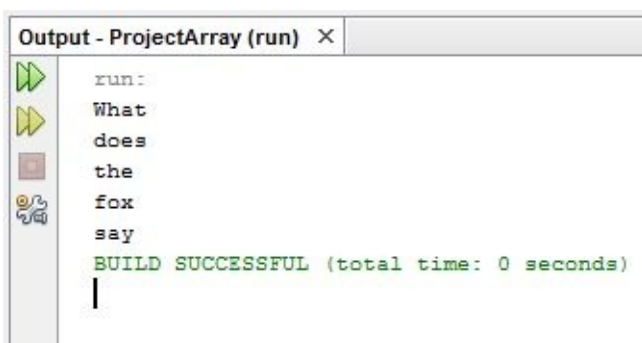
This code defines an array string with five positions. A specific text is then defined to every array position.

Below is a loop, which runs round all the array positions, which can display the message for every position:

```
int i;  
for ( i=0; i < stringArray.length; i++ ) {  
System.out.println( stringArray[i] );  
}
```

This loop will keep on running as long as the value in variable known as i is lower than the length of the array known as stringArray.

Once you run this program, the Output screen must look like this:



It is also possible to sort array strings, similar to what we have done with integers.

However, take note that the sorting will be alphabetical. Hence, a will come before b. Java is using Unicode texts to do a comparison of a letter to the string. Hence, the uppercase letters will come before the lowercase letters. Take a look at the code below:

```
package arraystrings;

import java.util.Arrays;

public class stringsarry {

    public static void main(String[] args) {

        String[] stringArray = new String [5] ;

        stringArray[0] = "What";
        stringArray[1] = "does";
        stringArray[2] = "the";
        stringArray[3] = "fox";
        stringArray[4] = "say";

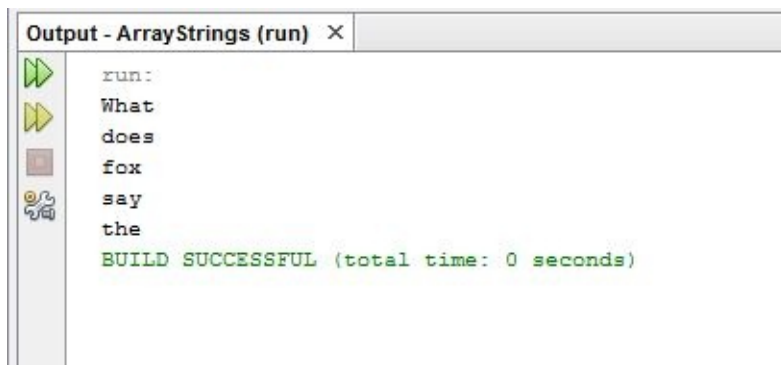
        Arrays.sort(stringArray);

        int i;
        for (i=0; i < stringArray.length; i++) {
            System.out.println( stringArray[i] );
        }

    }

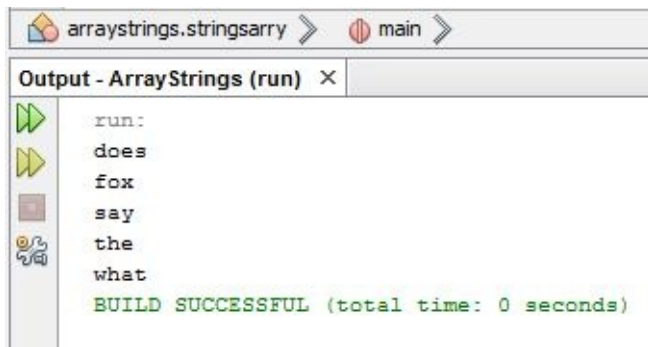
}
```

Once you run this code, the Output screen must look like this:



```
Output - ArrayStrings (run) X
run:
What
does
fox
say
the
BUILD SUCCESSFUL (total time: 0 seconds)
```

Even though we have already sorted the array, the word “What” is still placed first. If this is an alphabetical sort, we are expecting the word “does” to be printed first. This will be true if all the letters are in small caps. In the code, try changing the capital “W” of “What” to a small “w”. Run the code again. The output screen should now be like this:



```
run:
does
fox
say
the
what
BUILD SUCCESSFUL (total time: 0 seconds)
```

At this point, the word “what” is now placed last. We will learn more about strings in the next section.

JAVA MULTI-DIMENSIONAL ARRAYS

The arrays that we have been using so far have only stored one data column. However, we can still define an array to store several columns. These are known as multi-dimensional arrays. For example, consider an excel file with columns and rows. If you have five rows and six columns, then the excel file can store 30 values such as this:

	A	B	C	D	E	F
1	50	18	64	63	15	75
2	34	88	43	46	19	14
3	23	12	65	35	22	77
4	56	43	45	44	34	85
5	78	11	32	36	32	20

Multi-dimensional arrays could store all values above. In order to set up these arrays, use the code below:

```
int[ ][ ] numArrays = new int[5][6];
```

These are defined similarly as a regular array, except you have two pairs of square braces. The first pair of square braces are for the rows and the second pair of braces are for the columns. In the code line above, we are instructing Java to establish an array with five columns and six rows. In order to establish values within a multi-dimensional array, you need to track the columns and rows. Here is the code to assign the first rows from the excel screenshot.

```
numArrays[0][0] = 50;  
numArrays[0][1] = 18;  
numArrays[0][2] = 64;  
numArrays[0][3] = 63;  
numArrays[0][4] = 15;  
numArrays[0][5] = 75;
```

Hence, the first row is row 0. The columns will then go from zero to 5, which is 6 items. In order to fill out the next row, we need to write the following lines:

```
numArrays[1][0] = 34;
```

```
numArrays[1][1] = 88;  
numArrays[1][2] = 43;  
numArrays[1][3] = 46;  
numArrays[1][4] = 19;  
numArrays[1][5] = 14;
```

Take note that the number of the columns is still the same, but the number of rows is now assigned to 1.

In order to access the items within the multi-dimensional array, the strategy is to employ a single loop within another. Here is the code to access the number from above. It will use two for loops:

```

public static void main(String[] args) {

    int [][] numArrays = new int [5][6];

    numArrays[0][0] = 50;      numArrays[1][0] = 34;
    numArrays[0][1] = 18;      numArrays[1][1] = 88;
    numArrays[0][2] = 64;      numArrays[1][2] = 43;
    numArrays[0][3] = 63;      numArrays[1][3] = 46;
    numArrays[0][4] = 15;      numArrays[1][4] = 19;
    numArrays[0][5] = 75;      numArrays[1][5] = 14;

    numArrays[2][0] = 23;      numArrays[3][0] = 56;
    numArrays[2][1] = 12;      numArrays[3][1] = 43;
    numArrays[2][2] = 65;      numArrays[3][2] = 45;
    numArrays[2][3] = 35;      numArrays[3][3] = 44;
    numArrays[2][4] = 22;      numArrays[3][4] = 34;
    numArrays[2][5] = 77;      numArrays[3][5] = 85;

    numArrays[4][0] = 78;
    numArrays[4][1] = 11;
    numArrays[4][2] = 32;
    numArrays[4][3] = 36;
    numArrays[4][4] = 32;
    numArrays[4][5] = 20;

    int rows = 5;
    int columns = 6;
    int i, j = 0;

    for ( i = 0; i < rows; i++) {
        System.out.print(numArrays[i][j] + " ");
    }
    System.out.println( "" );

}

```

The first loop will be used for the rows, while the second loop will be used for the columns. On the first run of the first loop, the value of the *i* variable is 0. The code within the for loop is also a loop. The entire second loop will run as long as the value of the *i* variable is 0. The second for loop uses the *j* variable. The variables *j* and *i* could then be used to access the array.

numArrays[i][j]

Hence, the dual loop system is used to access all values within the multi-dimensional array, row after row.

JAVA ARRAY LISTING

If you are not certain about the numbers you need to store in the array, it is ideal to use an ArrayList. This data structure is dynamic, which means the items could be included and eliminated from list. A regular java array is static structure as you have to deal with fixed array size.

To define an ArrayList, it is important first to define an import package sourced out from the library **java.util**:

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

Next, we need to make a new object ArrayList using this code:

```
ArrayList listTest = new ArrayList( );
```

Take note, we don't need any squared braces at this point.

When we create ArrayList, we can then start adding elements using the method add:

```
listTest.add( "item one" );  
listTest.add( "item two" );  
listTest.add( "item three" );  
listTest.add( 8 );
```

Within the rounded brackets of the add method, we need to place the text or number that we like to include in the ArrayList. However, we can only include objects. Notice that the first 3 objects that we have included on the list are Strings, while the last object is a certain number. However, take note that this would be an object number of integer type and not primitive type data int.

These listings could be referenced using index number as well as through the utilization of the method **get**:

```
listTest.get( 3 )
```

This code could obtain the object at position 3 index on this list. Remember, index numbers begin at zero, hence this could be item four.

It is also possible to get rid of an item from the array listing. You can use this list value:

```
listTest.remove("item one");
```

Or this one:

```
listTest.remove(1);
```

Once you remove the item, the ArrayList will be resized. Hence, you must take extra care in getting an object from the list if you are using the number index. In this example, if you remove item 1, then the list will contain only three items. Getting the object using the number 2 index will then yield back an error.

In order to access every item in the ArrayList, we can add an Iterator, which is also located in the library: java.util:

```
import java.util.Iterator;
```

The next step is to attach the ArrayList to the new object Iterator:

```
Iterator it = listTest.iterator( );
```

This will add a new object Iterator, which can be used to access the objects in the listing named as listTest. It is ideal to use the Iterator object because it comes with the methods known as hasNext and next, which you can employ in a loop.

```
while ( it.hasNext( ) ) {  
System.out.println( it.next( ) );  
}
```

The hasNext method will yield a Boolean value. This value should be false once there are

no remaining objects within the ArrayList. We can use the method next to access all the objects in the listing.

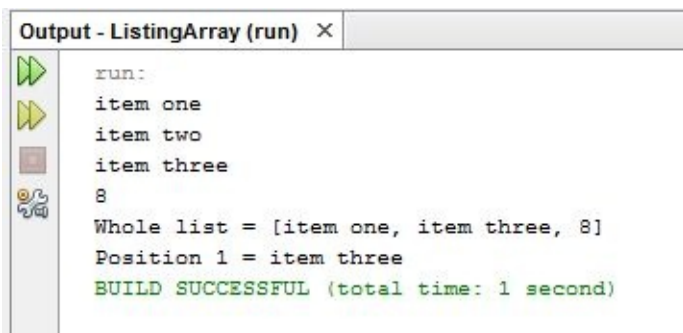
You can check this theory by using the code below:

```
public static void main(String[] args) {  
  
    ArrayList listTest = new ArrayList();  
  
    listTest.add("item one");  
    listTest.add("item two");  
    listTest.add("item three");  
    listTest.add (8);  
  
    Iterator it = listTest.iterator();  
  
    while (it.hasNext()) {  
        System.out.println(it.next());  
    }  
  
    //REMOVE AN ITEM FROM THE LIST  
    listTest.remove("item two");  
  
    //PRINT OUT THE NEW LIST  
    System.out.println("Whole list = " + listTest);  
  
    //GET THE ITEM AT INDEX POSITION 1  
    System.out.println("Position 1 = " + listTest.get(1));  
}
```

Take note of the line, which displays the whole listing:

System.out.println(“Whole list=” + listTest);

Through this, you can get a glimpse of the items that are included on the list.



```
Output - ListingArray (run) X  
run:  
item one  
item two  
item three  
8  
Whole list = [item one, item three, 8]  
Position 1 = item three  
BUILD SUCCESSFUL (total time: 1 second)
```

Remember, the ArrayList is useful if you are not certain of the number of elements that should be stored in the item lists.

Chapter 5 – Java String Methods

Not similar to double variables and int variables, strings are objects, so you can do certain things with text strings which you cannot do with double or int variables. The same principle applies to primitive types such as short, long, float, char, single, byte, and boolean.

Before we discuss the manipulation of text strings, let's first discuss basic strings.

HOW JAVA HANDLES STRINGS

String refers to a sequence of Unicode characters held within a name variable. Take note of the string below:

String sampleText = “Ron”;

This instructs Java to establish a string object with three characters: R, o, and n. In Unicode character set, these values are U+0052, U+006F, and U+006E. Unicode values are handled as hexadecimals. In the past section, we worked on an array that held text strings that we have sorted:

```
package arraystrings;

import java.util.Arrays;

public class stringsarry {

    public static void main(String[] args) {

        String[] stringArray = new String [5] ;

        stringArray[0] = "What";
        stringArray[1] = "does";
        stringArray[2] = "the";
        stringArray[3] = "fox";
        stringArray[4] = "say";

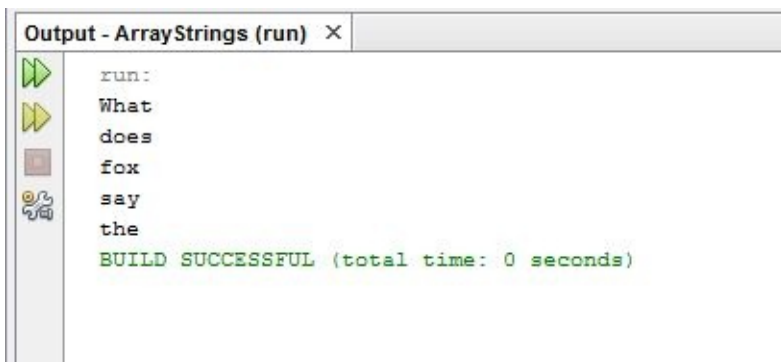
        Arrays.sort(stringArray);

        int i;
        for (i=0; i < stringArray.length; i++) {
            System.out.println( stringArray[i] );
        }

    }

}
```

Once you run the program, the output screen will be this:



```
Output - ArrayStrings (run) X
run:
What
does
fox
say
the
BUILD SUCCESSFUL (total time: 0 seconds)
```

We have noticed that the word “What” is listed first. If the array is arranged alphabetically, the word “does” should come first. However, “d” has a hexadecimal value of `u\0064`, which is the decimal number 100. Uppercase “W” has a hexadecimal value of `u\0057`, which is the decimal number 87. 87 is less than 100, so the “W” will be listed first.

Now, it’s time to manipulate some texts.

Upper and Lower Case

Transforming text Java strings to upper or lower case is quite straightforward. You can use the pre-existing methods `toUpperCase` and `toLowerCase`.

Initiate a new project and type in the following code:

```
package stringmanipulation;

public class projectstrings {

    public static void main(String[] args) {

        String caseChange = "object to change";
        System.out.println( caseChange );

        String result;
        result = caseChange.toUpperCase();

        System.out.println( result );

    }

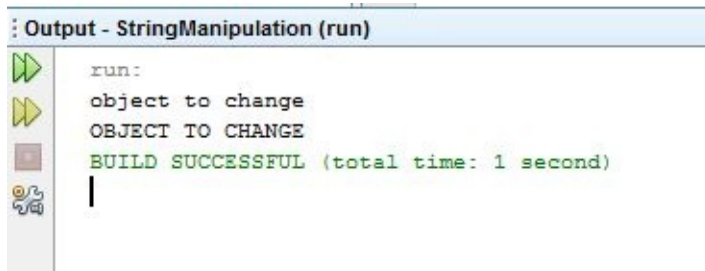
}
```

The function of the code's first two lines is to set up a variable `String` to store the text "object to change", and then print it out. The next line sets another variable `String` known as `result`. Then, we have another line to help us in conversion:

`result = caseChange.toUpperCase();`

In order to use a string method, we first need to encode the string that we like to work on. In this case, we are using the variable string we named **`caseChange`**. Include a dot after the name variable and a list of available methods will appear, which you can use on the string. Choose `toUpperCase`. Remember, the method still requires the empty rounded brackets. Once Java has converted the word to uppercase text, we will store the new string in variable string.

Once you run the program, the Output screen will be like this:



```
run:
object to change
OBJECT TO CHANGE
BUILD SUCCESSFUL (total time: 1 second)
|
```

However, we don't need to hold the changed text into another variable. The code below will just work as fine:

```
System.out.println( caseChange.toUpperCase( ) );
```

At this point, Java will proceed with the string conversion without the need to assign the result to another variable.

If you need to transform to lowercase, you can use the method **toLowerCase**. You could use this in precisely the same way as **toUpperCase**.

Strings Comparison

It is possible to do a comparison of strings. In comparison, Java will utilize the hexadecimal values instead of the letters. For instance, if you like to compare the terms Map and Man to know which will be listed first, you could utilize a string method known as **compareTo**. Check how this one works.

There is no need to begin a project. You can just delete or comment out the code that we have written. Then, type in the code below:

```
package stringmanipulation;

public class projectstrings {

    public static void main(String[] args) {

        int result;
        String Word1 = "Map";
        String Word2 = "Man";

        result = Word1.compareTo(Word2);

        if (result < 0) {
            System.out.println("Word1 is lower than Word2");
        }
        else if (result > 0) {
            System.out.println("Word1 is more than Word2");
        }
        else if (result == 0) {
            System.out.println("The same word ");
        }
    }
}
```

We have established two variable strings to hold the words “Map” and “Lap”. The method compareTo is the code below:

result = Word1.compareTo(Word2);

The method **compareTo** returns a value. The value, which is returned will be higher than

0, lower than 0, or zero as its value. When Word1 is listed before Word2, the value that will be returned is lower than zero. When Word1 is listed after Word2, then the value that will be returned is higher than zero. If the words are the same, then a the return value will be zero.

Hence, it is important to define the value that the method **compareTo** returns to a variable. We can assign value for an **int** variable known as result. In this code, the IF Statements basically checks the variable result.

But if we compare a text string with another, Java will compare their hexadecimal values instead of the actual letters. Since uppercase letters have lesser hexadecimal value compared to lowercase letter, an uppercase letter “M” in “Map” will be listed before the lowercase letter “m” in man. You can check it out. Change “Map” to “map” in your code. You will get an output that shows Word1 is higher than Word2”, which means that Java will list map after the word man in alphabetical order.

In order to resolve this problem, we can use a similar method known as **compareToIgnoreCase**. As you might guess, the program will ignore the uppercase and lowercase letters. Using this, “man “will come first.

The METHOD indexOf

To find a string or character inside another string, we could use the method **indexOf**. For instance, we can use this to check if there's at sign (@) in a given email address. We can use this example in a code. Take note that you can comment out or delete the code that we have written. Below is a code you can try:

```
package stringmanipulation;

public class projectstrings {

    public static void main(String[] args) {

        char ampersand = '@';
        String email_address = "harrypotter@hogwarts.com";

        int result;
        result = email_address.indexOf( ampersand );

        System.out.println( result );

    }

}
```

We need to test if the at sign (@) is present in the e-mail add. Hence, we first need to assign a variable char and define value @. Take note that we have used a pair of single quotations in the variable char. After specifying an e-mail add, we have included a variable result, which is a variable **int**. You might be wondering the code yielded an integer. The method `indexOf` will provide a value. This could return the number position of the character ampersand in the `email_address` string. Below is a related code:

result = email_address.indexOf(ampersand);

The textstring that we are trying to find will be listed first. Next to the dot, we have the method `indexOf`. Within the rounded braces of `indexOf`, we can add some options such as typing a symbol or the variable char. In this example, we are placing the variable `ampersand` within the rounded brackets of the `indexOf`. Then, Java will position the @ sign in the e-mail add and will hold the value in variable `result`.

Once this code is run, the output should be 11. You might be thinking that the at symbol is the 12th character of the string e-mail add. However, `indexOf` begins at zero. When the character isn't located within the string that you are searching, the method `indexOf` will

show a result of negative one (-1). To check this out, erase the at sign from the string email_address. Then run again the code. The result should be negative one (-1).

We can use the value of negative 1 to our benefit. Below is the code, but this time, we have an IF statement, which analyzes the value within the variable result.

```
package stringmanipulation;

public class projectstrings {

    public static void main(String[] args) {

        char ampersand = '@';
        String email_address = "hogwarts.com";

        int result;
        result = email_address.indexOf( ampersand );

        if (result== -1) {
            System.out.println( "Email Add is Invalid");
        }
        else {
            System.out.println( "Email Add is Fine");
        }

    }

}
```

Hence, the indexOf result is negative 1. Else will permit the user to proceed. If you like to check for more characters, we could also utilize indexOf. Below is a code to check the e-mail add if this is ending with .com.


```

package stringmanipulation;

public class projectstrings {

    public static void main(String[] args) {

        String dotCom = ".com";
        String email_address = "hogwarts.com";

        int result;
        result = email_address.indexOf( dotCom );

        if (result== -1) {
            System.out.println( "Email Add is Invalid");
        }
        else {
            System.out.println( "Email Add is Fine");
        }

    }

}

```

In this code, we are utilizing a variable String to store the text that we like to test (.com) and not a variable char.

Remember, you will get a -1 result if the object you are looking is not located in the String, which comes prior to the period of indexOf. Or else, indexOf will yield a position in the first character that matches. In the above code, the period is the 8th character of "hogwarts.com" if you begin counting from zero.

We can also define a beginning position for the searches. In the example above, we could begin looking for .com next to the at sign. Below is the code:

result = email_address.indexOf(dotCom, atPos);

The difference is the inclusion of an added variable within the indexOf brackets. We no obtained the string that we like to look for, which refers to the text that is inside the variable dotcom. However, we have now a beginning search position. This refers to the value of the variable named atPos. We obtain the atPos value through the indexOf to find the position of the at sign in the e-mail add. Java can initiate the search originating from the position, instead of beginning at zero that is the default set up.

Ends With...Starts With

In the above program, we can utilize the endsWith method:

```
Boolean ending = email_address.endsWith( dotcom );
```

We can define a Boolean var for the endsWith, since the method will return a true- false answer. The string that we need to check is within the rounded brackets of endsWith. Hence, the object that we are searching will go prior to it. When the character is inside the string search, then the true value will be returned, or else it could be false. So, you can include an if...else statement for value checking.

```
if (ending == false ) {  
System.out.println( "Email Add is Invalid" );  
}  
else {  
System.out.println( "Email Add is Fine " );  
}
```

The startsWith method can be utilized in a same manner:

```
Boolean startVal = email_address.startsWith( dotcom );
```

Remember, the return value is Boolean true-false.

SUBSTRING METHOD

Substring is another useful method in Java programming. This method will let you obtain a section of text from other strings. In the previous program, we can obtain the final five letter or signs from this address and check if this is co.us.

To try a substring, we can create a simple Name Swapper program. For this, we can transform the first 2 characters of the surname and change them with the first 2 letters of a first name, and the other way around. Let's say we have the name:

“Harry Potter”

We will then change the “Po” of “Potter” with the “Ha” of “Harry” to get “Hatter. The “Ha” of “Harry” could be exchanged with the “Po” of “Potter” to get “Porry”. The result printed will be “Porry Hatter”.

We can use substring such as below:

```
String FullName = “Harry Potter”;  
String FirstNameChars = ””;  
FirstNameChars = FullName.substring( 0, 2 );
```

We can establish a string that will look, in this example the string “Harry Potter”. The string that we are trying to locate will be written after the equals symbol. Next to the period, write the name of the substring method. We have two methods in using substring, and their distinction is the occupying numbers within the rounded brackets. In this code, there are two numbers: zero and 2. We are telling Java to obtain the chars occupying pos 1 in the string and pause harvesting if we have collected two. The two characters will then be returned and positioned in the FirstNameChars variable. If you like to go to the right to the string end, we can include this line:

```
String test = FullName.substring( 2 );
```

At this point, we have only one number within the rounded braces of substring. So, Java can begin at character 2 in the FirstName string, and then obtain the characters from pos 2 next to the string end.

To check this out, initiate a new project and at the end, include a print line.

The method substring will allow you to obtain the first two chars of the name “Harry”.

To obtain the first chars, 0 and 2 should be assigned inside the rounded brackets of the substring. You might be thinking that to obtain the “Po” of “Potter” we can instead include this line:

```
= FullName.substring(5, 2);
```

After all, we intend to get two characters. But at this point, the 5 will instruct Java to begin from the “P” of “Potter”. Take note that the 1st position in a string is zero and not 1. Hence, begin at pos 5 within the string & obtain two characters.

But running this code will return an error. This is because the 2nd number within the rounded substring braces will not signify the number of characters we intend to obtain. It signifies the string position, which we like to terminate. In assigning 2, we are instructing Java to terminate at the character located in position 2 of the string. Because we cannot move from pos 6 in reverse towards position 2, we will result to an error message.

Take note that if we begin counting at zero in the string “Harry” you may think that pos 2 is the letter “r”. You are correct. However, the substring begins prior to the character at this position and not next to it.

Hence, to obtain the “Po” of “Potter”, we can do this code:

```
FullName.substring( 5, FullName.length( ) - 3 );
```

Now, the 2nd number refers to string length, which is for this example is -3 characters. The string length refers to the number of characters a text has. “Harry Potter” has 12 characters, which includes the space. Eliminate 3 and we have 9. Hence, we are instructing substring to begin at char 5 and terminate at char 7.

We should also take note of the space position between the names. The two chars that we like to obtain from the 2nd name will be placed next to the character space. We need some code, which obtains these first two chars next to the space.

You can use the indexOf to take note of the space position:

```
int spacePos = FullName.indexOf(" ");
```

To define a character space, we can include a space in between the double quotes. This will then go within the indexOf rounded braces. The value that will be displayed could be an integer, and this is the location of the first occurrence of the character space in the FullName string. Check it by inserting the code above. Include a line print to test the Output.

In this program, the space is located at position 5 on the string. You can use this data to obtain the first 2 chars of “Potter”. You need to instruct Java to run from the 1st char next to the space and terminate at the next 2 chars:

FullName.substring(spacePos + 1, (spacePos + 1) + 2)

Hence, the 2 numbers that are assigned within the rounded brackets are actually substring of these codes:

spacePos + 1, (spacePos + 1) + 2

The intention is to begin at the 1st character after the space (space +1), and terminate 2 characters after the position – (spacePos+1)+2. Include the lines in the code. The new method substring will spill over the next two line, but if you like, you can keep your own code.

Now we have the “Ha” from Harry and the “Po” from Potter. We just need to obtain the remaining of the text from the 2 names then do the swapping.

Remember, you could utilize the substring to obtain the rest of the chars from the 1st name:

String OtherFirstChars = "";

OtherFirstChars = FullName.substring(2, spacePos);

System.out.println(OtherFirstChars);

As well as the rest of the chars originating from the 2nd name:

String OtherSurNameChars = "";

OtherSurNameChars = FullName.substring((spacePos + 1) + 2,

FullName.length());

System.out.println(OtherSurNameChars);

We are not looking for the numbers confined in the rounded brackets of the substring. To obtain the 1st name chars, you can use the numbers:

2, spacePos

This is telling Java to begin at pos 2 & proceed to the space position. But when it comes to the remaining characters of the 2nd name, it could be a tricky code:

(spacePos + 1) + 2, FullName.length()

Take note that the (spacePos+1)+2 signifies the beginning pos of the 3rd char of the 2nd name. We can end the string length that could allow us to run the remaining characters.

You can eliminate the line prints and allow a user to type a first_name and family_name. In this new code, we have added the input from the keyboard that you have learned in the previous chapters.

Of course, we need to include some lines to run some error check. However, we will presume the user will type in a first_name as well as the family name with a space in between the names. Without the space, the result will be an error.

THE METHOD EQUALS

The method known as Equals will allow you to test the strings if they're alike. Here is a code that you can try:

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

    String email_address1 = "meme@me.cob";
    String email_address2 = "meme@me.com";
    Boolean isMatch = false;

    isMatch = email_address1.equals(email_address2);

    if (isMatch == true) {
        System.out.println( "Email Address Match ");
    }
    else {
        System.out.println("Email addresses don't match");
    }
}
```

This code will allow us to test if an e-mail add is similar to another e-mail add. The first 2 code lines establish 2 variable strings for every e-mail add. The next code line establishes a variable Boolean. The equals method will yield value true | false. Line 4 signifies the method function:

isMatch = email_address1.equals(email_address2);

Within the rounded braces of the method equals, we will put the string that we are checking. The second string will go prior to the method equals. Java will next display true | false if these two strings are alike. The one checking is the IF statement.

However, the equals method can only compare objects. It can test strings, as they are objects. Still, it is not possible to use the method for comparing int variables. As an example, this code will result to an error message:

int num1 = 10;

int num2 = 11

Boolean isMatch = false;

isMatch = num1.equals(num2);

Remember, the variable `int` is not an object because it is a primitive type data. Still, it is possible to convert a data primitive `int` type within an object through this code:

```
int num1 = 10;
```

```
Integer num_1 = new Integer(num1);
```

In this code, the variable `int` known as `num1` will be converted into an object `Integer`. Notice that we have used a keyword. Within the rounded brackets for the `Integer`, we will place the primitive data `int` type that we like to turn into an object.

THE charAt METHOD

You can test to see what character is contained in a specific string. In Java, we are using the charAt method. Below is a code that you can work on:

```
String email_address = "albus@hogwarts.com";  
char aChar = email_address.charAt( 5 );  
System.out.println( aChar );
```

This code line will check the letter as at position 5 in string e-mail add. The value that will be returned is a char type variable.

```
char aChar = email_address.charAt( 5 );
```

When you run the code above, the output will be the at sign (@). The number within the charAt rounded braces is the string position that we are trying to test. In this code line, we need to locate the char in pos 5 of the string: email_address. Remember, the count begins at 0 similar to substring.

Another great use for charAt is to obtain a character from a variable string, which is provided by the user, & will be transforming it to one variable character. For instance, we can ask a user to enter Y to proceed or N to close.

It's not possible to directly employ the class Scanner to obtain one character to hold a variable char. Hence, we can utilize the method next () to obtain the following string, which the user can input. Then we have an integer next, then long, then double, and even a Boolean. However, there is no character next. This is the case even if user types a char, it could be read as a string, not as char. Take note that a variable char holds a number Unicode in integer form.

You can now use a charAt to obtain a char coming from a string, which the user will input despite the case that the user types in a letter:

```
char aChar = aString.charAt( 1 );
```

In this code, we are telling Java to obtain the char at pos 0 within the string known as aString, then hold it using the variable aChar.

We also included the IF statement in order to check the contents of the variable aChar. Notice the use of a pair of single quotations to confine Y.

THE REPLACE METHOD

In Java, the replace method is used to replace all the character occurrences in a particular string. Consider this sentence:

“Where are you wands?”

We need to replace “you” with “your”.

There are different ways to use the replace method, and the difference mainly lies on what you place within the rounded brackets of the method. We are replacing one series of characters with another. Consider the comma that separates the two as the term with. Then, you will have “replace you with your”.

We can also replace a single character:

aString.replace('\$', '%')

The code above means “Replace \$ with %”.

We can also use a normal expression in the replace methods, but that is beyond the coverage of this ebook.

TRIM

We can trim out the white space from the strings. The white space refers to the space characters, newline characters, and tabs. It is easy to use the trim method:

```
String amend = " blank space ";  
amend = amend.trim( );
```

Hence, the trim method will go after the string we like to change. The blank spaces before the word “blank” and after “space” in the above code should be deleted.

If we are getting user input, then it is ideal to use trim method on them.

Chapter 6 – Java Methods

In the previous chapters, we have been using java methods, and at this point, you should know that the established ones are really useful. Now is the time that you learn how to write your own Java methods.

THE FORMAT OF A JAVA METHOD

A method is just a section of a code, which performs a certain task. However, methods are structured in a specific format. It comes with a method header and a body. The header will signify Java the type of value such as string, double, or int. It also includes the return type with the name for the method that also included in the header. You can define values over the methods, and they could be written inside the rounded brackets. The body of method is where the code will run.

```
int total( int aNumber) {  
  
    int a_Value = aNumber + 10;  
  
    return a_Value;  
  
}
```

The return type of the Java method should go first, which is an int type in the example above. Next to this method is a space followed by the method name. In the example, we named it total. Within the rounded brackets, we have instructed Java that we are storing the method a variable known as aNumber, and this can be integer.

In order to chunk off this method from other code, you should use curly brackets. The code for the method should be confined within the curly brackets. Take note of the keyword **return** in the method. This is clearly the value, which you like a return for the method after you execute the code. However, it should be the same type like the return type in the header method. Hence, the return value cannot be a string if you begin the method with the **int total**.

There are instances that you do not like Java to return any value at all. A method that will not give any value could be established using the word void. In this case, the method doesn't require the keyword **return**. Below is an example of a method, which does not return a value.

```
void print_text(String someText) {  
  
    System.out.println( "Some Text Here" );  
  
}
```

The method above can print some message. It could still perform its function even without value so we have set it as a void method. Remember, methods don't need to be passed with values. We can just run some code. Here is another void method without values.


```
void print_text() {  
    System.out.println( "Some Text Here" );  
}
```

Here is another int method with no values passed:

```
int total() {  
    int a_Value = 10 + 10;  
    return a_Value;  
}
```

Notice that the rounded brackets don't have any values, yet they are still crucial. Without the rounded brackets, the code will run into an error.

How to Call Your Java Methods

Take note that methods cannot do anything unless you call them into action. Before we test it out, add a new class to the project, so we can place all the methods rather than cluttering the main class.

Begin a new application project. As usual, provide a project name and change the name of the main method to something else. In the code below, we have called our projects **projectmethods**, and the class **MethodTests**.

```
package projectmethods;

public class MethodTests {

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

}
```

To insert a new class to the project, choose New File from the File menu in the NetBeans. A dialogue box will appear. In the section for Categories, choose Java, and in the section for File Types, choose Java Class. At the bottom, click the Next button. In the second step, type a title for the new class. We will call our type SampleMethods.

Hence, we are creating a second class known as SampleMethods that will be in the Project projectmethods. Just click the Finish button and the program will create the new class file. Another tab will appear with default comments on how you can change the templates. Just delete the comments.

You will notice that there is no Main method in this code, but a blank class with the class name you have selected as well as a pair of curly brackets for the code. We can add another method.

This is the int method that you have been working earlier in the name total. It doesn't have anything within the rounded brackets so we are not storing any value. The method is just adding up values $5 + 5$ and handles the answer in a variable known as a_Value. The method will return this value, which should match the type return from the header method. This code is ok as these values are both int types.

Take note that the variable `a_Value` should not be seen outside the total method. We can't set up within a method that we can't access outside the method. This is called a variable local, which is local to the method.

In order to call the total method, choose the TestMethods tab in the NetBeans, which is the one with the Main method. We will call the total method from the Main method.

We first need to create a new object from the SampleMethods class.

In order to make a new object from a class, we will begin with the name of the class, SampleMethods in this example. This is in place of the String, double, or int variable. The variable type that we are creating is a SampleMethods variable. Next to the space, we are adding a title for the new SampleMethods variable. The name of our variable is test1. Notice that it is underlined in grey as we are not yet adding any command.

This will be followed by the equals symbol and then the keyword: new, which refers to new object. Next to this keyword, add a space followed by the name of the class. Remember, the name of the class requires rounded brackets. The line should be ended as usual with a semi-colon.

Using this code, we have created a new SampleMethods object using the name test1. The total method within the class SampleMethods will now be accessible using the Main method of the SampleMethods class.

We are defining an int variable using the name aVal. Next to the equals symbol comes the name of the test1 class. In order to access the class methods, enter a period, and NetBeans could show a popup box using the methods available:

The total variable is defined on the list, while the other variables are built in methods. The rounded brackets have no assigned values, because this method will not accept any value. However, the int return type is shown in the right.

Choose the total by double clicking it to add this in the code. Next, add a semi-colon at the end of the line. Then, we will add a print line.

In order to call a method, which returns a value, take note of the value that is being returned by the method. Then define this value to another variable, which is aVal in this example. However, the method must be available if you enter a period after the name of

the object.

But if the method is a void type, there is no need to assign it to another variable such as aVal. You can switch back to the SampleMethods class and include the method void, which you have already learned.

The print_text is the new method. Notice that it also has an empty pair of rounded brackets as we don't intend to store any values. Its function is to print out a message. After adding the void method, we can now switch back to the class SampleMethods. You can now insert the line below:

test1.print_text()

After adding the period, we can see the new method on the list.

These methods are now included in the list – print_text and total. Values that they will return will be shown on the right, void and int.

Since print_text is a void method, there is no need to establish a return value. We just need the name of the object, a period, and the void method that we like to call. Java will just get on with running the code within the method.

HOW TO PASS VALUES TO JAVA METHODS

To do something with the value, we can pass values to the method. The value should be written within the pair of rounded brackets of the method.

Now we have two methods using total as their name. The only difference is that the new method contains a value within the rounded brackets. Java will let you do this, through the process method overloading. Hence, you can include as many methods you like using the same name containing any value. But take note that it is not allowed to have the same kind of variables within the rounded brackets. Hence, you can't have two total methods, which return int values having the same int values confined within the rounded brackets. For example, the line below is not allowed:

```
int total( int aNumber ) {  
    int a_Value = aNumber + 10;  
    return a_Value;  
}  
  
int total( int aNumber ) {  
    int a_Value = aNumber + 20;  
    return a_Value;  
}
```

Even though the two methods perform different things, they still have similar method headers.

Insert some comments first before trying out the new method.

```
/**  
 * Returns an integer value, which is  
 * 20 plus the number passed as a paramater.  
 * @param aNumber Any Integer value  
 * @return 20 + the value of aNumber  
 */  
int total(int aNumber) {  
    int a_Value = aNumber + 20;  
  
    return a_Value;  
}
```

The param in the above comments refers to parameter, which is the technical word for the value within the rounded brackets of the method headers. This parameter is known as aNumber with its integer values. Take note that we are using the at sign character before the param and return.

In this code, we are passing an integer value and adding 20 to this value. The return_value will be the the sum of the two.













The next step is to return to the code and insert the line below:

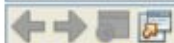
```
int aVal2 = test1.total(30);
```

Typing the period after the object test1 will show a list of options The total method will be included. Choose the new method.

The added comments are now confined in the blue box under the method options. Any user reading the method can easily figure out what the code is all about. When you add the method total2, insert the number 30 inside the rounded brackets. End this line with a semi-colon.

```
int aVal2 = test1.
```

	hashCode()	int
	total()	int
	total(int aNumber)	int
	equals(Object obj)	boolean
	getClass()	Class<?>
	notify()	void
	notifyAll()	void
	print_text()	void
	toString()	String
	wait()	void
	wait(long timeout)	void
	wait(long timeout, int nanos)	void



[prjmethods.MyMethods](#)

```
int total(int aNumber)
```

Returns an integer value, which is 20 plus the number passed as a paramater.

Parameters:

aNumber - Any Integer value

Returns:

20 + the value of aNumber

When you hand over the value, the method will perform its function.

Include a print line to the code:

```
System.out.println( "Method result2= " + aVal2 );
```

Run the program. The Output screen should be like this:

```
run:
Method result= 20
Some Text Here
Method result2= 50
BUILD SUCCESSFUL (total time: 1 second)
```

Conclusion

I hope this book was able to help you to learn Basic Java Programming.

The next step is to practice your Java programming skills and continue learning advanced skills including:

- Writing your own Java Methods
- Writing your own Java Classes
- Handling Java Errors
- Managing Java Text Files
- Form Controls
- Java Databases
-

Finally, if you enjoyed this book, then I ' d like to ask you for a favor, would you be kind enough to leave a review for this book on Amazon? It ' d be greatly appreciated!

[Click here to leave a review for this book on Amazon!](#)

Thank you and good luck!



If you would like to receive free and bargain books, join this exclusive book club by clicking on the image above. You will receive free copies of ebooks in .pdf or .mobi format.