Unit - 3: Fundamentals of Java Programming

3.1 Introduction to Java

Welcome to “Fundamentals of Java Programming”! In this unit, you will learn to write and execute Java programs. But before you start writing one, it is important to know what a program is. A computer program is a sequence of instructions given to the computer in order to accomplish a specific task. The instructions tell the computer what to do and how to do it. Programming is the preparation and writing of programs for computers.

Programmers write their programs in a high level programming language such as Java, C++, Python, Ruby or Scala. However, as you are already aware, a computer only understands its own language called “machine language”.**Therefore, a compiler is needed** to translate high level program code into machine language code that will be understood by the computer. After a program is compiled, the machine code can be executed on the computer, say Windows, for which it was compiled. If the program is to be executed on another platform, say Mac, the program will first have to be compiled for that platform and can then be executed.

Java is a very popular high level programming language and has been used widely to create various types of computer applications such as database applications, desktop applications, Web based applications, mobile applications, and games among others. A Java compiler instead of translating Java code to machine language code, translates it into Java **Bytecode** (a highly optimized set of instructions). When the bytecode (also called a Java class file) is to be run on a computer, a Java interpreter, called the **Java Virtual Machine (JVM)**, translates the bytecode into machine code and then executes it. The advantage of such an approach is that once a programmer has compiled a Java program into bytecode, it can be run on any platform (say Windows, Linux, or Mac) as long as it has a JVM running on it. This makes Java programs **platform independent** and highly **portable**.

To write a Java program, you will need a Text Editor (for writing the code)and a Java compiler (for compiling the code into bytecode). There are a wide variety of **Java Integrated Development Environments (IDEs)** available in the market that come equipped with a text editor and a Java compiler thus simplifying writing, compiling and executing Java programs. Most of them are freely downloadable from the Internet. We will be using the open source and free **Java Net Beans IDE** for writing Java programs.

So let's dive straight in and write a simple Java program that prints a Welcome Message (“Hello World”) on the screen. Since this will be your first Java program, you will first need to setup the programming environment (Refer to **Appendix A** for installation and starting Net Beans).

Step 1: Double click the Net Beans icon on your computer to start Net Beans. It will open the Net Beans IDE (Figure 3.1(a).

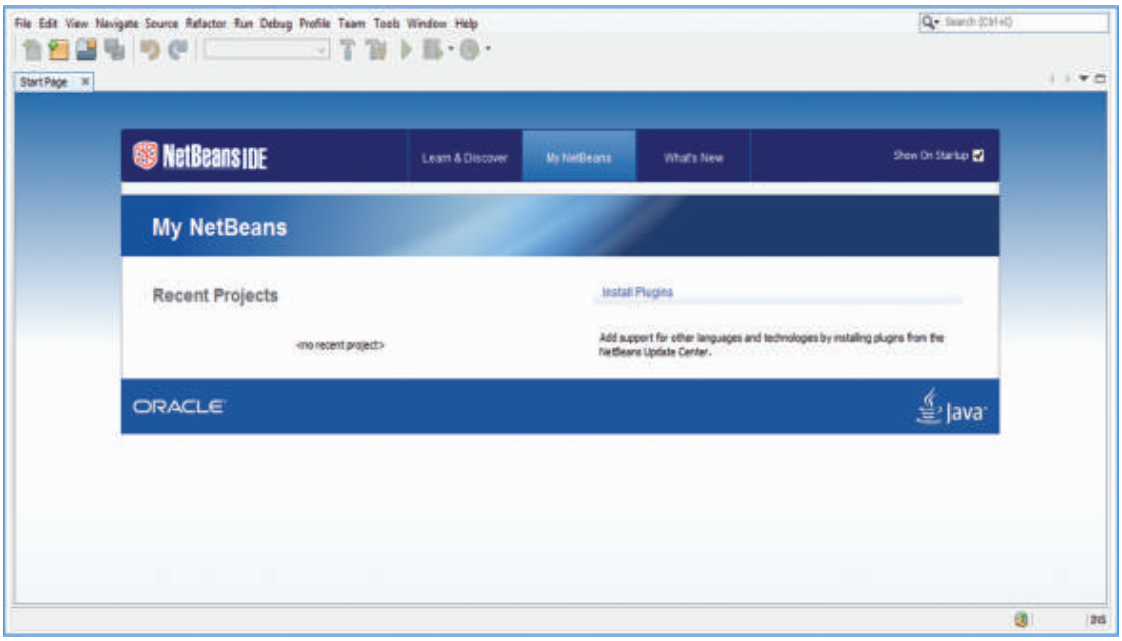


Figure 3.1(a): Java NetBeans IDE

Step 2: All software development in NetBeans is organized in the form of Projects, so we begin a new Project. In the IDE click **File> New Project (Ctrl + Shift + N)** Figure 3.1(b).

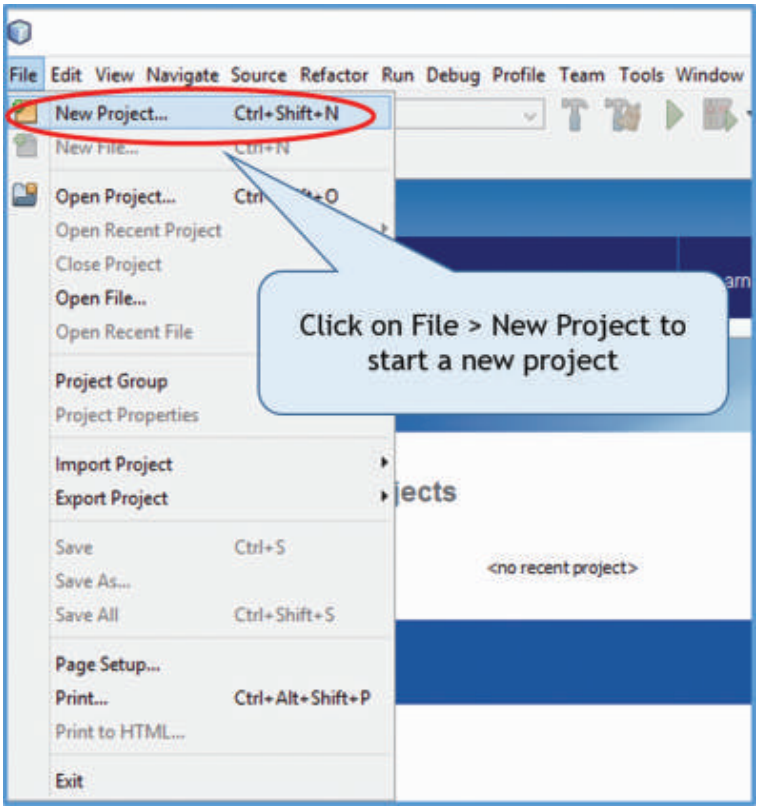


Figure 3.1(b): Create a New Project

Step 3: In the **New Project** dialog box that appears, under the **Categories** list**,** select **Java** and under the **Projects** list select **Java Application** (they should be already selected). Click on **Next** to create a new Java Application Project Figure 3.1(c).

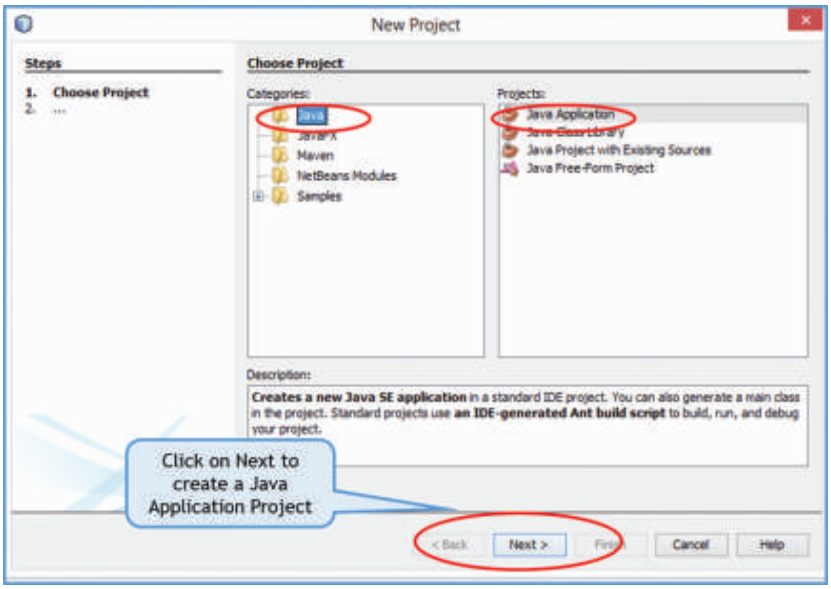


Figure 3.1(c): Create a New Java Application

Step 4: In the **New Java Application** dialog box that appears, in the **Project Name** field, type a name for the Project Figure 3.1(d). Here we have named the Project "`HelloWorld`". When you type the name of the project, the **Project Folder** field also changes automatically. So does the **Create Main Class** field. You can optionally change the **Project Location** and the **Project Folder** by clicking the **Browse** button. Click on **Finish** to finish creating the Java Application Project and to return to the IDE.

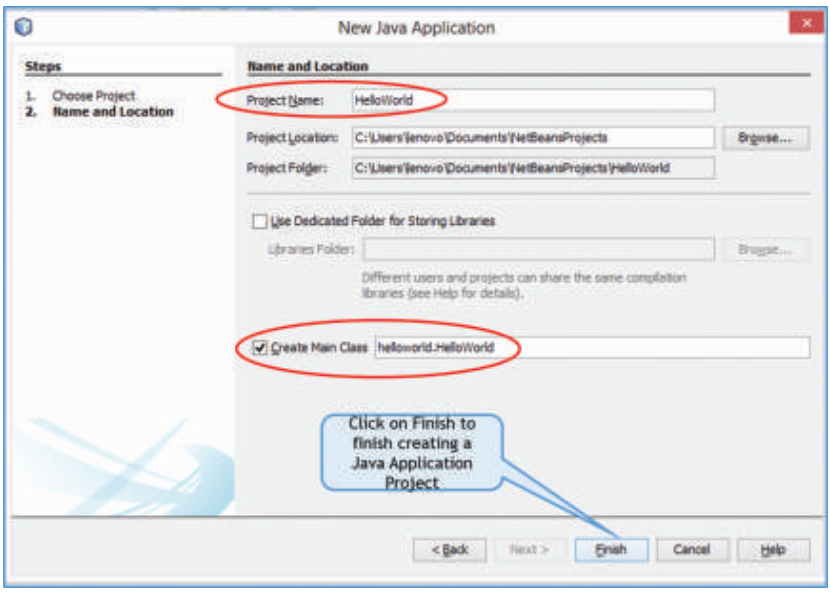


Figure 3.1(d): Finish creating new Java Application

On the left side of the NetBeans IDE, observe the **Projects** tab Figure 3.1(e). (If you can't see it, click **Windows > Projects** on the menu bar).

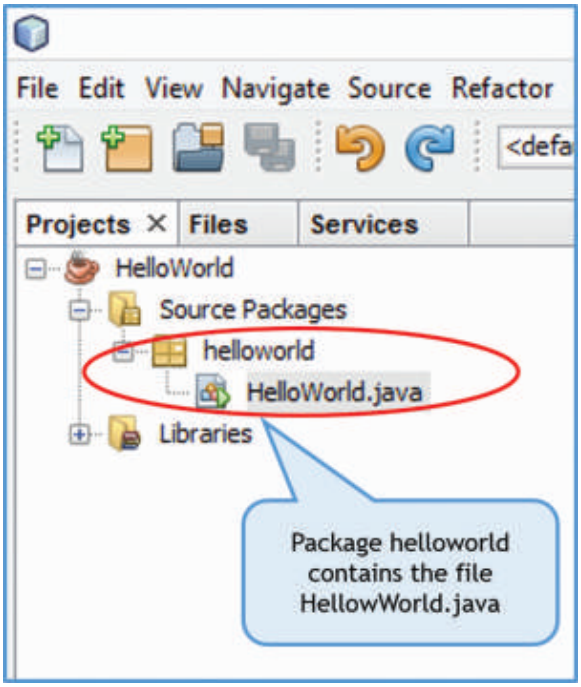


Figure 3.1(e): Projects Tab

Notice that a source code file called `HelloWorld. Java` has been created in a package called `Helloworld programs`. On the right side of the IDE is the code editor window where the Java program code in the `HelloWorld. Java` file is displayed. NetBeans has already filled in some code for you Figure 3.1(f).

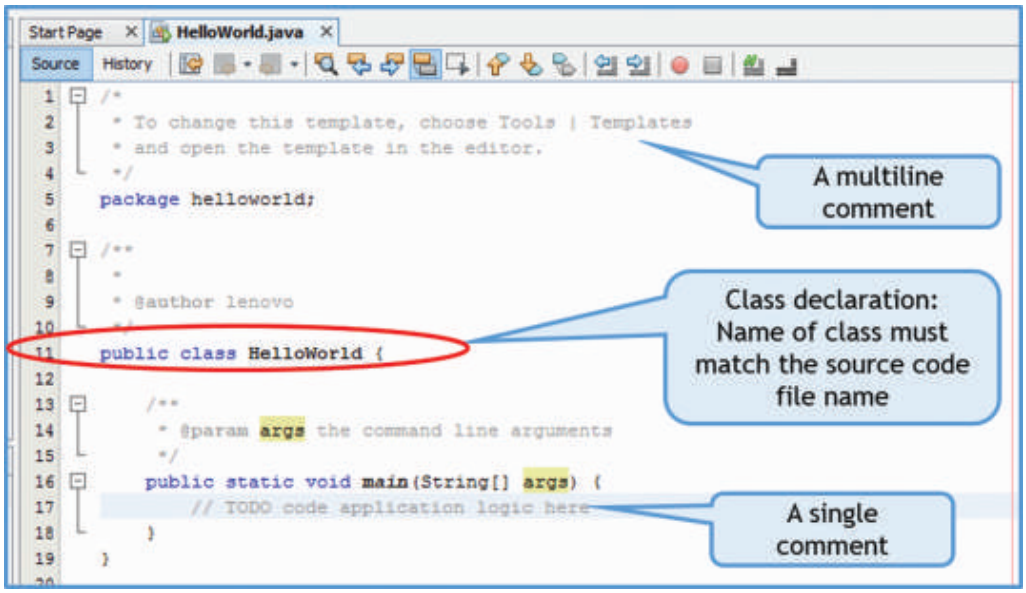


Figure 3.1(f): Code Editor Window

Observe that quite a bit of the code in the code editor is greyed out. All the grey parts are **comments**. Comments are used in code by programmers to *document* their programs - to provide explanatory notes to other people who read the code. This is especially useful in the real world, where large programs are written by one programmer and maintained by other programmers. You can write comments in a Java program in the following two ways:

- Beginning a comment line with two consecutive forward slashes (//)
- Writing the comment between the symbols /* and */

The former method is used for single line comments while the latter is generally preferred for multiple line comments. Comments are used for enhancing the readability of code and are ignored by the compiler when compiling a file.

If we ignore the comments, the `Helloworld.Java` program is as below:

```
package HelloWorld;
public class HelloWorld {
      public static void main (String[] args) {
   }
}
```

Notice that the first line in the program is a **package** statement. A package in java is a group of related classes. In our program this statement declares that the `HelloWorld`. Java program is part of the `HelloWorld` package. All classes in a package can share their data and code.

The next line declares a class called `Helloworld.Java` demands that the class name should be the same as the source file name. However the package name can be different from either the class or the source file name. NetBeans, by default, names the package with the same name as the project, except that it is in all lower case.

The contents of a class are enclosed within curly braces. Within the contents of the `Helloworld` class, a **method** called main has been declared. A method is a group of statements written to perform a specific task. The method body is enclosed within a pair of curly braces and contains the statements that the method will execute. Main is a special method that every Java application must have. When you run a program, the statements in the main method are the first to be executed. The main method in the `HelloWorld` class has only a single line comment within it.

```
// TODO code application logic here
```

In the program that we are building, we want to instruct the computer to display a welcome message in the Java output window. So within the main method we will write a single line of code that does just that. For this we will use the most common pre-built Java output method `System.out.println()`.

Step 5: In the Code Editor window, find the line"// TODO code application logic here". Click on the right end of that line. Press Enter and type in the following line of code

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

Notice that as you type the dot after the word System followed by a dot, NetBeans will display a list of available options Figure 3.1(g).

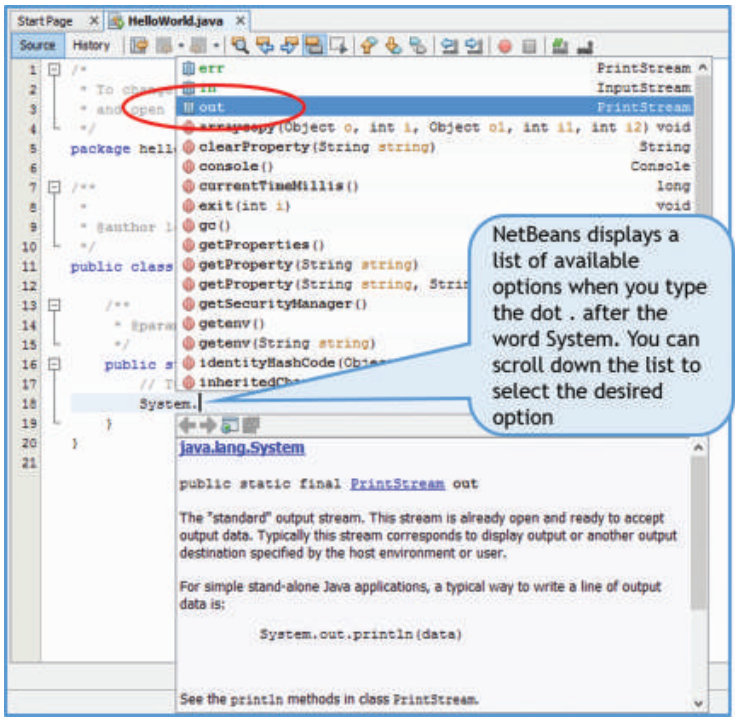


Figure 3.1(g): NetBeans Displays a List of Available Methods

At this point, you can continue to type the word **out** or alternatively, you can scroll down the displayed list to reach the word **out** and then either double click or press enter. When you type the dot after the word **out**, another list appears. Again either select `println` from the list or continue to type. After you complete the line, your code editor window should appear as in Figure 3.1(h).

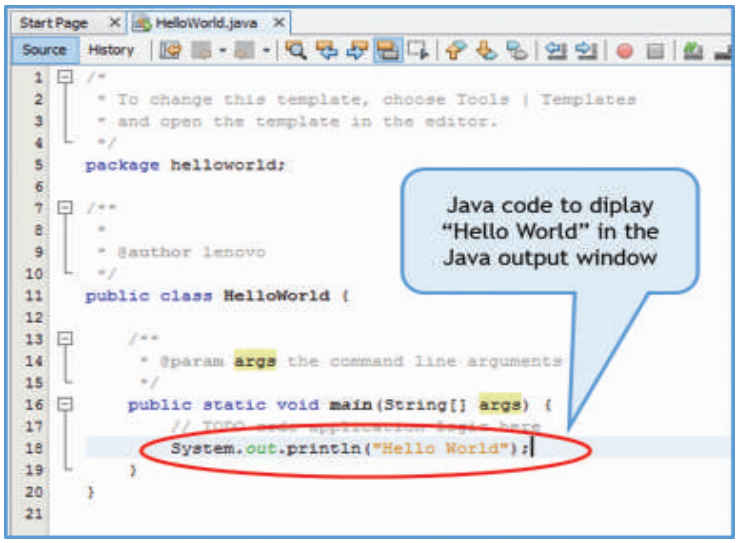


Figure 3.1(h): The Helloworld.java Program

Don't forget to type the **semicolon** at the end of the statement. All Java statements must end with a semicolon. If you make any errors while typing your code, for example, if you forget a semicolon at the end of the statement or if you misspell a keyword, NetBeans will warn you with a glyph - an exclamation mark in a red circle, in the left margin of the line Figure 3.1(i). If you bring the cursor near the warning, you will see helpful hints to correct the errors. You can use these hints to correct your errors.

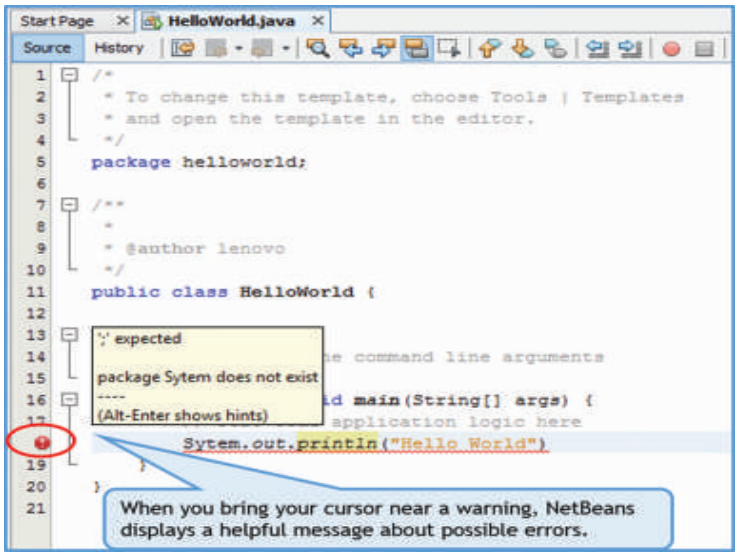


Figure 3.1(i): NetBeans Warning

Step 6: In the IDE toolbar, click on **File > Save (Ctrl + S)** or **File > Save All (Ctrl + Shift + S)** to save the `HelloWorld.Java` program Figure 3.1(j). Because NetBeans has a default **Compile on Save** feature, you do not have to compile your program manually, it is compiled automatically when you save it.

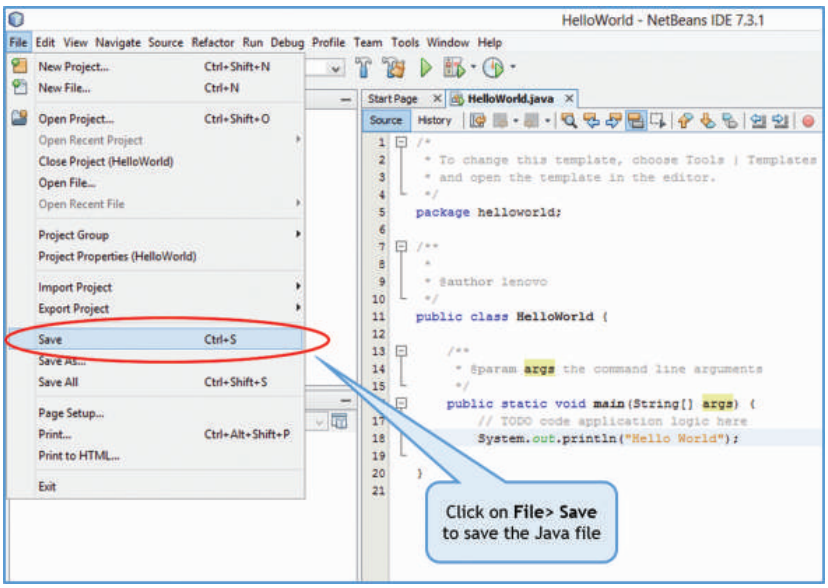


Figure 3.1(j): Saving a File

Step 7: In the IDE toolbar, click on **Run > Run Main Project (F6)**or **Run >Run File (Shift + F6)** to execute the Java Program Figure 3.1(k).

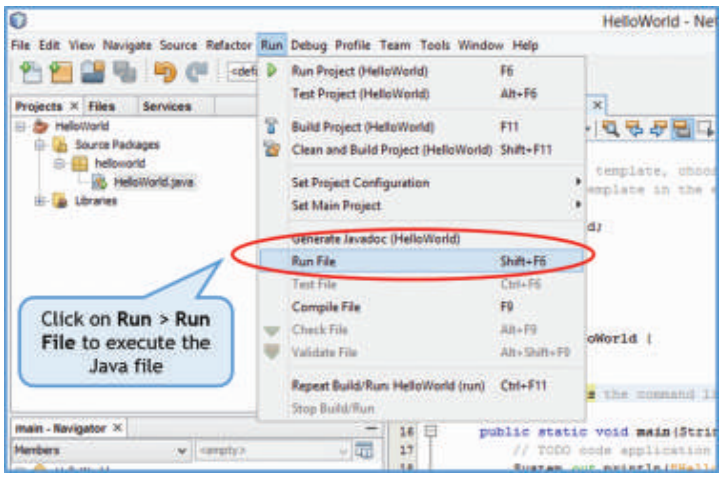


Figure 3.1(k): Executing a Java Program

You can also run your program by clicking the green arrow button on the toolbar Figure 3.1(l).

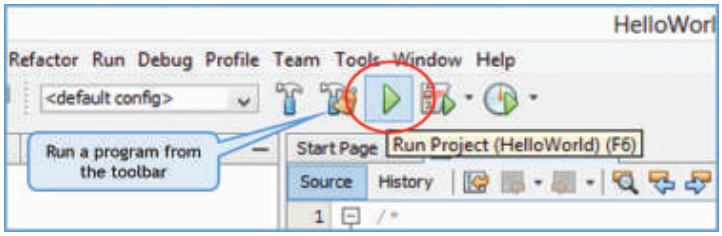


Figure 3.1(l): Executing a Java Program from the Toolbar

If there are no errors in your program, the NetBeans IDE compiles and then executes your program. You should now see the program output (the “Hello World” message) in the Output Display Window near the bottom of the IDE as in Figure 3.1(m).

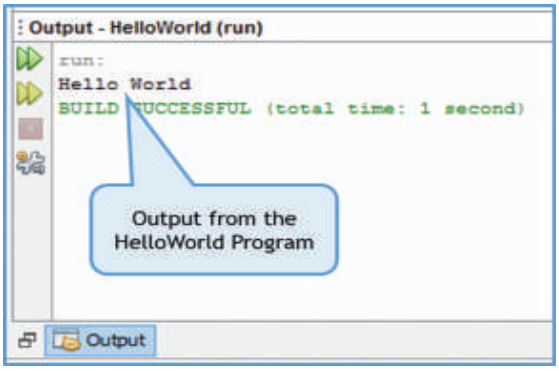


Figure 3.1(m): NetBeans Output Window

The `System.out.println("HelloWorld");` statement printed the line “`Hello World`” on the screen followed by a *newline* causing the subsequent line to be printed on the next line. Try changing the program code to

```
System.out.print("Hello World");
```

and run the program again. You should see the output as in Figure 3.1(n). Do you notice the difference from the previous run? This time a *newline* is not printed after the "`Hello World`" causing the subsequent line to be printed on the same line.

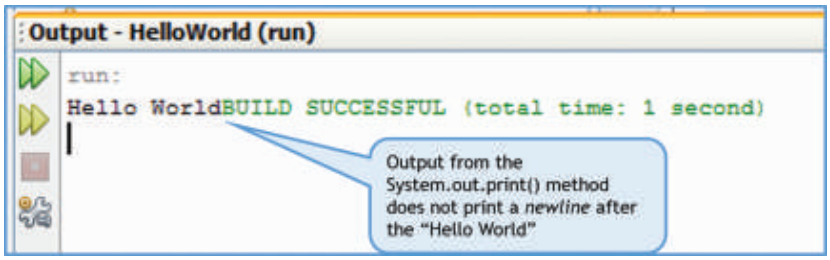


Figure 3.1(n): Output from the System.out.print() Method

Now, you have successfully written and executed your first Java Program. **Congratulations, you are now a Java programmer!**

3.2 Data Types and Variables

In the previous section, you learnt how to write and execute a simple Java program. This time let us write a program that does something a little more useful than just display a message. We will write a program that handles data. Given the marks obtained by a student and the total number of marks for an exam, our program will calculate the percentage obtained by the student. And we will calculate the percentage for two students.

3.2.1 Variables

To store the program data we will use **variables.** A variable is a placeholder for data that can change its value during program execution. Technically, a variable is the *name* for a storage location in the computer's internal memory. The *value* of the variable is the contents at that location. In our percentage calculator program, we will use three variables named `marks_obtained`, `total_marks` and `percentage` (Variable names have to be a single word, that's why we have added an underscore between the words "`marks`" and "`obtained`", "`total`" and "`marks`").

All data variables in Java have to be declared and initialized before they are used. When declaring variables, we have to specify the **data type** of information that the member will hold – integer, fractional, alphanumeric, and so on. The type of a variable tells the compiler, how much memory to reserve when storing a variable of that type.

In our percentage calculator program, the variable `total_marks` can have only integer values. So we declare it to be of type `int` – a keyword in Java that indicates an integer data type. The variables `marks_obtained` and `percentage` are declared of type **double**, since they can have fractional values (floating point numbers). Inside the main method, we declare these variables, we also assign them values using the = operator as shown below:

```
int total_marks = 400;
double marks_obtained = 346;
double percentage = 0.0;
```

To calculate the percentage, we construct an expression using the `total_marks` and `marks_obtained` variables and assign the value to the percentage variable. The * operator is used for multiplication and the / operator for division.

```
percentage = (marks_obtained/total_marks)*100;
```

To display the percentage in the IDE output window, we use our old friend – the `System.out.prinltn()` method.

```
System.out.println("Student1's Percentage = "+percentage);
```

Notice that the variable to be displayed is not put within the double quotes. To print the word "`Percentage`", we put it inside double quotes, to display the value stored in the `percentage` variable, we put it outside the double quotes. The + operator within the `System.out.println` statement, concatenates the string given in double quotes and the value of the variable.

To calculate the percentage for another student, we reuse the variables. We change the value of the variable `marks_obtained` and calculate the percentage again. There is no need to change the `total_marks` because its value remains the same as before.

```
marks_obtained = 144;
percentage = (marks_obtained/total_marks)*100;
System.out.println("Student2's Percentage = "+percentage);
```

To write the percentage calculator program in NetBeans, follow the steps given below:

Step 1: First, we will create a new Class file within the package HelloWorld (that was created in the previous section). Click on **`File > New File (Ctrl + N)`** to create a new file Figure 3.2(a).

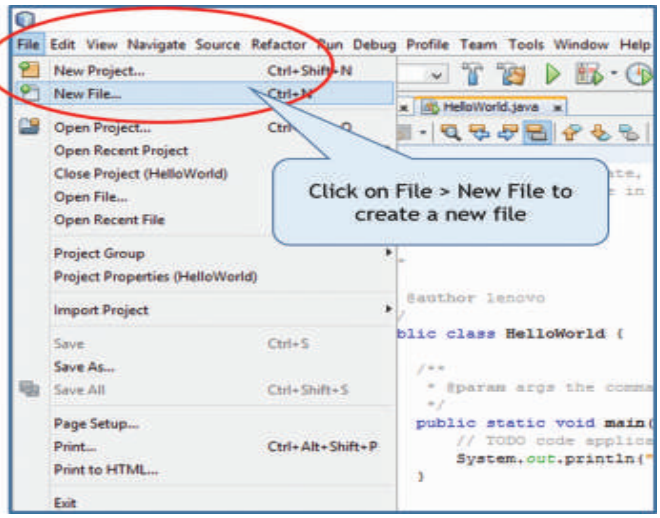


Figure 3.2(a) Creating a New File

Step 2: In the **New File** dialog box that appears, under the **Categories** list, select **Java** and under the **File Types** list select **Java Class** (they should be already selected). Click on **Next** to create a new Java Class file Figure 3.2(b).

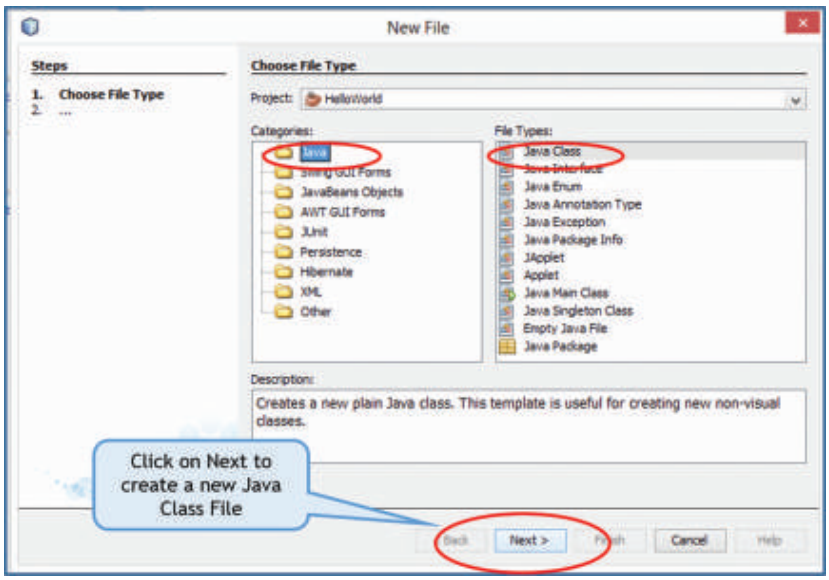


Figure 3.2(b): Creating a New Java Class File

Step 3: In the **New Java Class** dialog box that appears, type a name in the **Class Name** text box Figure 3.2(c). For our program, we type `Percentage Calculator`. When you type the name of the class, the **Created File** field also changes automatically. Make sure the **Package** field displays the `HelloWorld` package, if not click on the drop down list icon next to the field and select the `HelloWorld` package. Click on **Finish** to finish creating the New Java Class File and to return to the IDE.

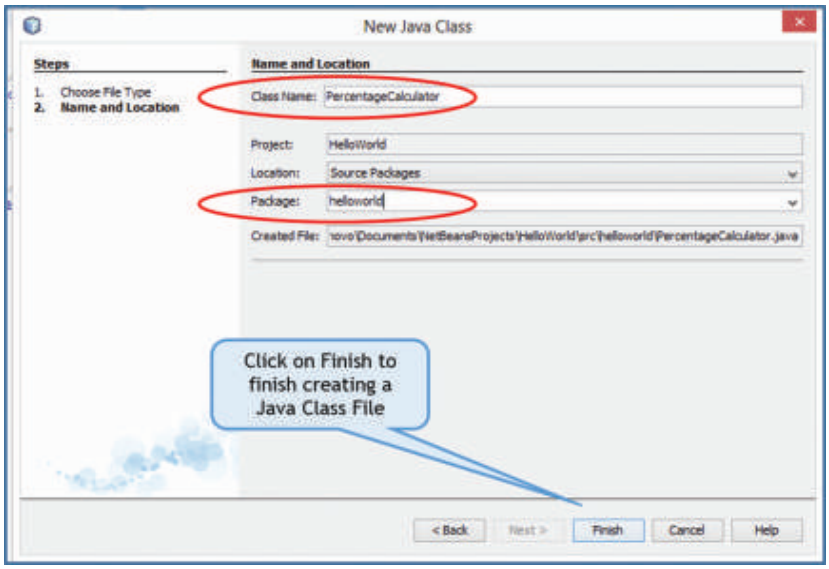


Figure 3.2(c): Finish Creating a New File

Observe that in the Projects tab, a file called `PercentageCalculator.java` has been added to the `HelloWorld` package Figure 3.2(d).

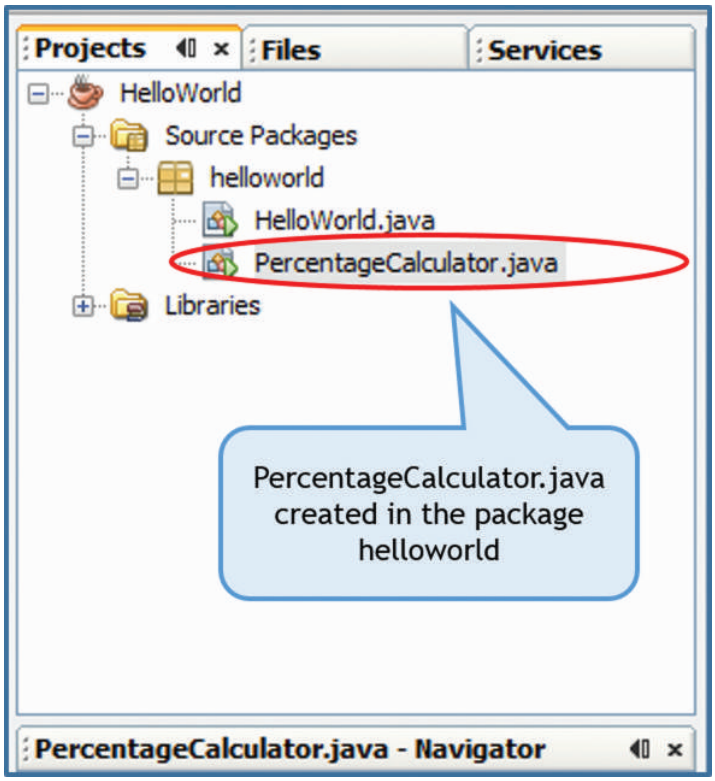


Figure-3.2(d): New File Added to the Package

Step 4: Click in the Code Editor Window under the `Percentage Calculator.Java` tab and type the code as shown in Figure 3.2(e). The comments inserted by NetBeans have been deleted for simplicity.

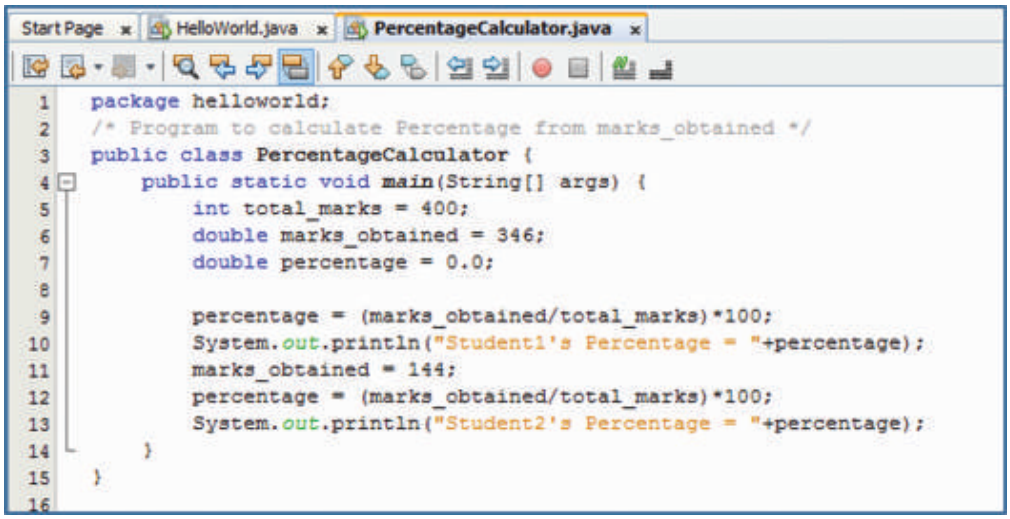


Figure 3.2(e): The Percentage Calculator Program

Step 5: Click **File > Save (Ctrl+S)**to save the file.

Step 6: Click **Run> Run File (Shift + F6)** to execute the program. The output of the program appears in the Output window as shown in Figure 3.2(f).

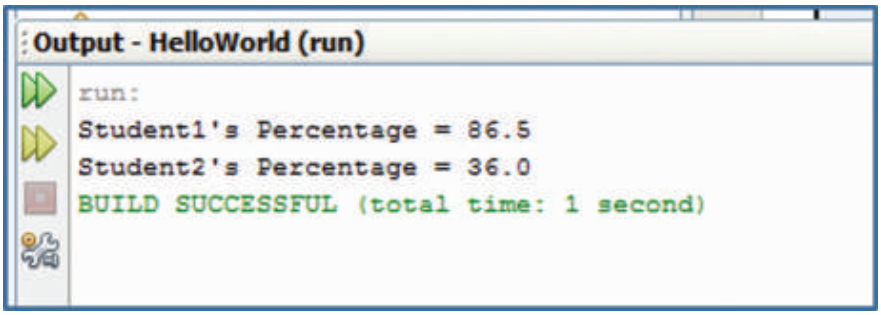


Figure 3.2(f): Output from the Percentage Calculator Program

As you may have noticed, the variables in a program act as placeholders for data handled by the program. Variables can change their value during program execution. The variables that you have seen in this section are **local variables**. Such variables are available only inside the methods where they are declared.

3.2.2 Primitive Data Types

Besides the two Java data types (int, double) that you have seen in the `Percentage Calculator` program, we can use six more data types. In all, Java supports eight primitive data types as in Table 1.

Table 1: Java Primitive Data Types

Data Type	Type of values	Size
byte	Integer	8-bit
short	Integer	16-bit
int	Integer	32-bit
long	Integer	64-bit
float	Floating Point	32-bit
double	Floating Point	64-bit
char	Character	16-bit
boolean	True or False	1-bit

STYLE TIP – Variable Names

1. Variable names can begin with either an alphabetic character, an underscore (_), or a dollar sign ($). However, convention is to begin a variable name with a letter. They can consist of only alphabets, digits, and underscore.

2. Variable names must be one word. Spaces are not allowed in variable names. Underscores are allowed. "`total_marks`" is fine but "`total marks`" is not.
3. There are some **reserved words** in Java that cannot be used as variable names, for example - int.
4. Java is a case-sensitive language. Variable names written in capital letters differ from variable names with the same spelling but written in small letters. For example, the variable name "percentage" differs from the variable name "`PERCENTAGE`" or "`Percentage`".
5. It is good practice to make variable names meaningful. The name should indicate the use of that variable.
6. You can define multiple variables of the same type in one statement by separating each with a comma. For example, you can define three integer variables as shown:

```
int num1, num2, num3;
```

3.2.3 String Variables

In the `Percentage Calculator` program of the previous section, we used variables to store numeric data. Quite often we want variables to store textual data, for example, the name of a student.

A variable of the primitive data type **char** can be used to store a single character. To assign a value to a char variable we enclose the character between single quotes.

```
char middle_name = 'M';
```

To store more than one character, we use the **String** class in Java. To assign a text value to a String variable we enclose the text between double quotes. For example,

```
String first_name = "Mayank";
String last_name = "Saxena";
```

To print char or String variables, we can use the `System.out.println()` method.

```
System.out.println(first_name+" "+ middle_name+" "+last_name);
```

The output from the statement given above is:

```
Mayank M Saxena
```

Note the use of the + operator. In Java, when used with numbers as operands, the + operator adds the numbers. When used with Strings as operands, the + operator concatenates the strings together. Also, note that the single space String literal (" ") is used so as to print a gap between the first and middle name and middle and last name.

3.3 Operators

Operators are special symbols in a programming language and perform certain specific

operations. You have already seen two operator * and /. Table 2 lists the Java Arithmetic operators, Table 3 lists the Relational operators, Table 4 lists the Assignment operators, and Table 5 lists the Logical operators available in Java.

Table 2 – Arithmetic Operators

Operator	Description	Explanation	Example (int a = 20, b = 30)	Result
+	Addition	Returns the sum of values of operands	a+b	50
-	Subtraction	Returns the difference of values of operands	a-b	-10
*	Multiplication	Returns the product of values of operands	a*b	600
/	Division	Returns the quotient of division of values of operands	b/a	1
%	Modulus	Returns the remainder of division of values of operands	a%b	10
++	Increment	Increments the operand by 1	a++ Or ++a	21
--	Decrement	Decrements the operand by 1	a-- Or --a	29

Table 3 – Relational Operators

Operator	Description	Explanation	Example	Result (int a=20, b=30)
==	equal to	Returns true if values of a and b are equal, false otherwise	a==b	false
!=	Not equal to	Returns true if values of a and b are not equal, false otherwise	a!=b	true
>	Greater than	Returns true if value of a is greater than that of b, false otherwise	a>b	false
<	Less than	Returns true if value of a is less than that of b, false otherwise	a<b	true
>=	Greater than or equal to	Returns true if value of a is greater than or equal to that of b, false otherwise	a>=b	false
<=	Less than or equal to	Returns true if value of a is greater than or equal to that of b, false otherwise	a<=b	true

Table 4 – Assignment Operators

Operator	Description	Explanation	Example	Result (int a=20, b=30)
=	Simple Assignment	Assigns value of left side operand to right side operand	a=b	a becomes 30
+=	Add and Assignment	Adds value of left side operand to right side operand and assigns the result to the right side operand Same as a = a+b	a+=b	a becomes 50 (20+30)
-=	Subtract and Assignment	Subtracts value of left side operand from right side operand and assigns the result to the right side operand Same as a = a-b	a-=b	a becomes-10(20-30)
*=	Multiply and Assignment	Multiplies value of left side operand to right side operand and assigns the result to the right side operand Same as a = a*b	a*=b	a becomes 600(20*30)
/=	Divide and Assignment	Divides value of left side operand by right side operand and assigns the quotient to the right side operand Same as a = a/b	a/=b	a becomes 0(20/30)
%=	Modulus and Assignment	Divides value of right side operand by left side operand and assigns the remainder to the right side operand Same as a = a%b	a%=b	a becomes 20(20%30)

Table 5 – Logical Operators

Operator	Description	Explanation	Example	Result (int a=20, b=30)
&&	Logical AND	Returns true if values of both a and b are true, false otherwise	a&&b	false
||	Logical OR	Returns true if values of either a or b is true, false otherwise	a||b	true
!	Logical NOT	Returns true if value of a false, true otherwise	!a	false

Java also provides Bitwise operators which are beyond the scope of this text.

3.4 Control Flow

Recall, that a program is nothing but a sequence of instructions. Java executes the instructions in sequential order, that is, one after the other. However, sometimes you might

want to execute an instruction only if a condition holds true. Or you may want to execute a set of instructions repeatedly until a condition is met. Java provides selection structures for the former and repetition structures for the latter.

3.4.1 Selection Structures

In real life, you often select your actions based on whether a condition is `true` or `false`. For example, if it is raining outside, you carry an umbrella, otherwise not.

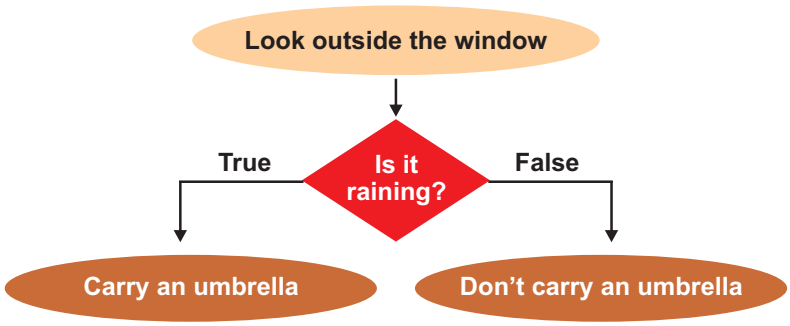


Figure 3.4(a): Selection in Real Life

Similarly in a program, you may want to execute a part of the program based on the value of an expression. Java provides two statements – the `if else` statement and the `switch` statement for executing a block of code conditionally (A block of code is a sequence of statements enclosed in curly braces).

3.4.2 The if Else Statement

The `if` statement in Java lets us execute a block of code depending upon whether an expression evaluates to true or false. The structure of the Java `if` statement is as below:

```
if (expression) {
   statements
}
```

The expression given in the round brackets after the if keyword, is a condition – an expression constructed using relational and logical operators. If the condition evaluates to true, the block of code following `if` is executed, otherwise not. If there is only one statement in the block after the `if`, the curly braces are optional. Let us enhance our `PercentageCalculator` program. If the percentage secured by the student is greater than or equal to 40%, we will declare the student as passed. So, in our percentage calculator program, we write,

```
if (percentage >= 40) {
   System.out.println ("PASSED");
}
```

If we also want to display "FAILED" when the student has scored less than 40%, we use the **if else** statement. The structure of the Java if else statement is as shown below:

```
if(expression) {
   statements
}
else {
   statements
}
```

As before, the block of code following `if` is executed if the conditional expression evaluates to `true`. The block of code following the **else** is executed if the conditional expression evaluates to false. In our `Percentage Calculator` program, we can write,

```
if (percentage >= 40) {
   System.out.println("PASSED");
}
else {
   System.out.println("FAILED");
}
```

Sometimes, you may want to execute a block of code based on the evaluation of two or more conditional expressions. For this, you can combine expressions using the logical `&&` or `||` operators. Let's say, we want to print the division with which the student has passed, then, we can write the following code. The first if statement uses the logical AND (`&&`) operator to combine two relational expressions.

```
if ((percentage >= 40)&& (percentage < 60)) {
   System.out.println("PASSED with II Division");
}
if (percentage >= 60) {
   System.out.println("PASSED with I Division ");
}
if (percentage < 40) {
   System.out.println("FAILED");
}
```

`if-else` statements can also be nested – you can have another `if-else` statement within an outer `if` or `else` statement. The example below is a nested if.

```
if (percentage >= 40) {
   System.out.println("PASSED");
   if (percentage>= 75) {
      System.out.println("With Distinction!");
   }
```

```
}
else {
   System.out.println("FAILED");
}
```

You can see the complete code listing of the new Percentage Calculator program in Figure 3.4(b) and its output in Figure 3.4(c).

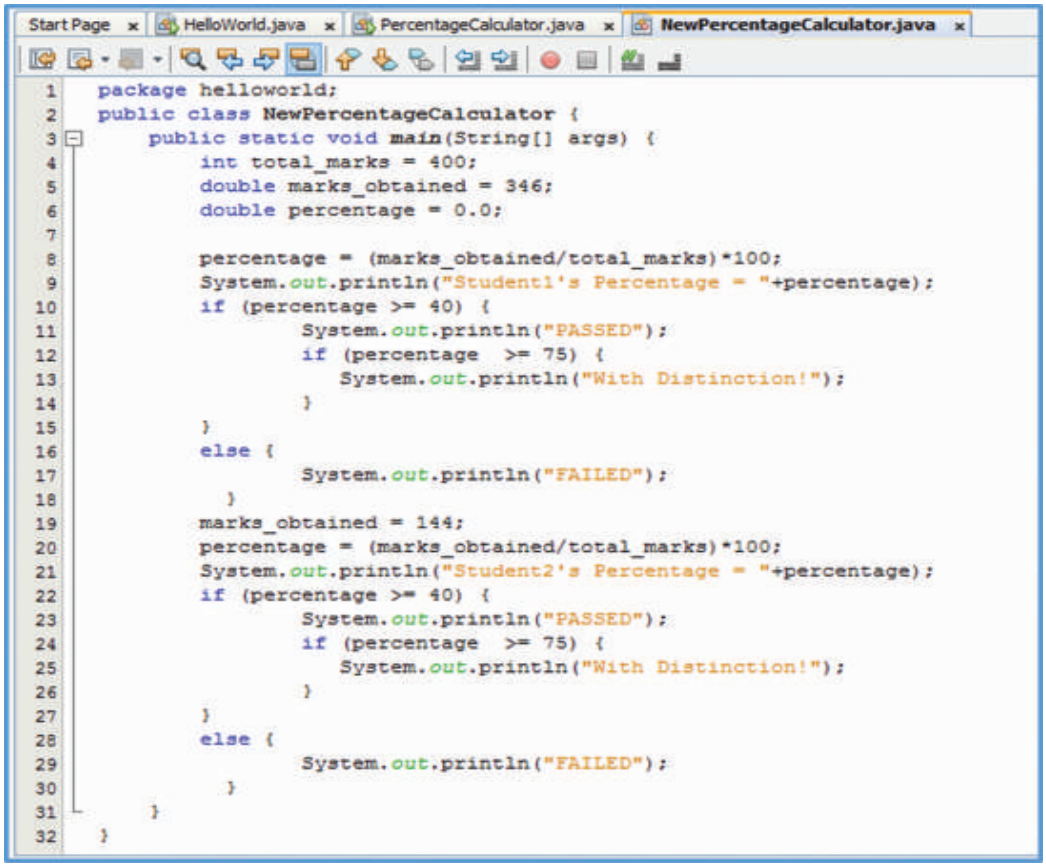


Figure 3.4(b): New Percentage Calculator

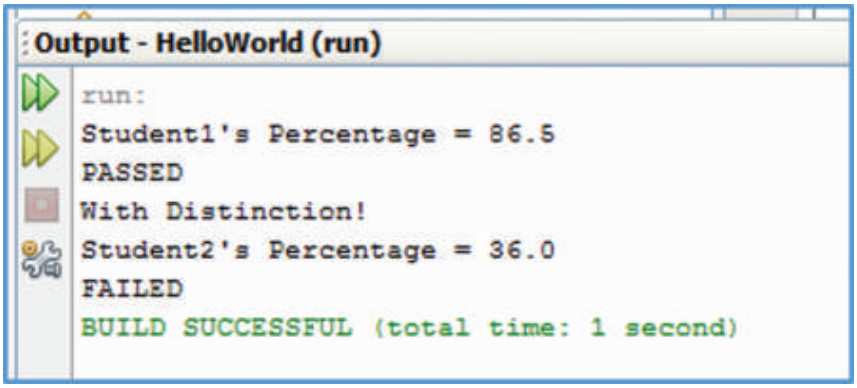


Figure 3.4(c): Output from New Percentage Calculator

3.4.3 The Switch Statement

The switch statement is used to execute a block of code matching one value out of many possible values. The structure of the Java switch statement is as follows:

```
switch (expression) {
case constant_1 : statements;
                  break;
case constant_2 : statements;
                  break;
…
…
default         : statements;
                  break;
}
```

Within the `switch` statement, as you can see, there can be many case statements. The expression is compared with the constants in each case group and the matching case group is executed. If the expression evaluates to some `constant = constant_1`, the statements in the case group `constant_1` are executed. Similarly, if the expression evaluates to `constant_2`, the statements in the case group `constant_2` are executed. The `break` statement after each case group terminates the `switch` and causes execution to continue to the statements after the `switch`. If there is no match for the expression with any case group, the statements in the `default` part are executed.

The expression in the `switch` statement must evaluate to `byte, short, int, or char`.

The program code below demonstrates usage of the switch statement.

```
public class SwitchDemo {
   public static void main (String[ ] args) {
      int today = 5;
      String day = "";
      switch (today) {
          case 1: day = "Monday"
                  break;
          case 2: day = "Tuesday";
                  break;
          case 3: day = "Wednesday";
                  break;
          case 4: day = "Thursday";
                  break;
          case 5: day = "Friday";
                  break;
```

```
            case 6: day = "Saturday";
                    break;
            case 7: day = "Sunday";
                    break;
            default: day = "Incorrect Day!
                    break;
        }
        System.out.println (day);
    }
}
```

Figure 3.4(d) displays the output from the `SwitchDemo` program. Since the expression in the `switch`, is the variable `today` which is equal to 5, the case corresponding to the constant 5 is matched. The statements in the case 5 are executed which set the variable day to "`Friday`". The break after the assignment causes the `switch` to terminate. Next, execution of the statement after the `switch`, i.e. the `System.out.println()` statement takes place which prints the value of the variable day.

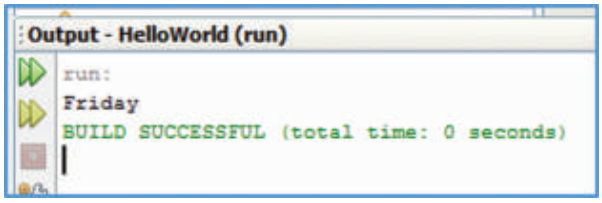


Figure 3.4(d): Output from Switch Demo Program

Can you guess what the output would be if the variable day was set to 10? Since 10 does not match any of the case statements, the default case would be executed and the output would be "`Incorrect Day!`".

3.4.4 Repetition Structures

In real life you often do something repeatedly, for example, consider a task such as reading a book, first you open the book, and then repeatedly - read a page; flip the page – until you get to the end of the book, then close the book.

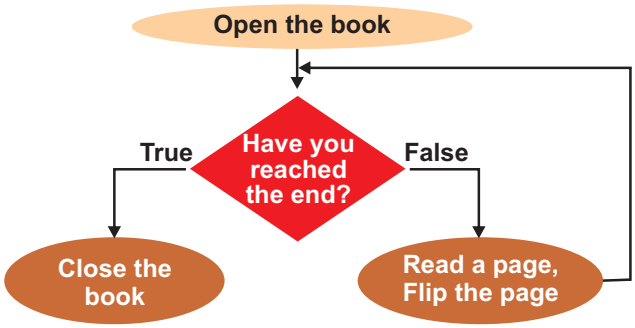


Figure 3.4(e): Repetition in Real Life

Similarly, when writing programs, you might need to perform the same sequence of statements repeatedly until some condition is met. The ability of a computer to perform the same set of actions again and again is called **looping**. The sequence of statements that is repeated again and again is called the **body** of the loop. The **test conditions** that determine whether a loop is entered or exited is constructed using relational and logical operators. A single pass through the loop is called an **iteration**. For example, a loop that repeats the execution of the body three times goes through three iterations.

Java provides three statements – the `while` statement, the do `while` statement, and the for statement `for` looping.

3.4.5 The While Statement

The `while` statement evaluates the test before executing the body of a loop. The structure of the Java `while` statement is as shown:

```
while (expression)
{
   statements
}
```

Let us write a program to print the squares of numbers from 1 to 5. The following steps need to be performed.

1. Initialize number = 1
2. Test if the number <= 5
3. If yes, print the square of the number;

   Increment number (number = number + 1)

   Go back to step 2
4. If no, exit the loop.

The tasks that need to be performed repeatedly are in step 3 – these constitute the body of the loop. The test condition is in step 2. The corresponding Java code is given below:

```
public class WhileDemo {
   public static void main (String[ ] args) {
      int number = 1;
      while (number <= 5) {
          System.out.print ("Square of " + number);
          System.out.println (" = " + number*number);
          ++number;
      }
   }
}
```

Note the use of the ++ operator to increment the value of number. The statement `++number` or `number++;` has the same effect as the statement `number = number + 1;`

The output from the `WhileDemo` program is displayed in Figure 3.4(f).

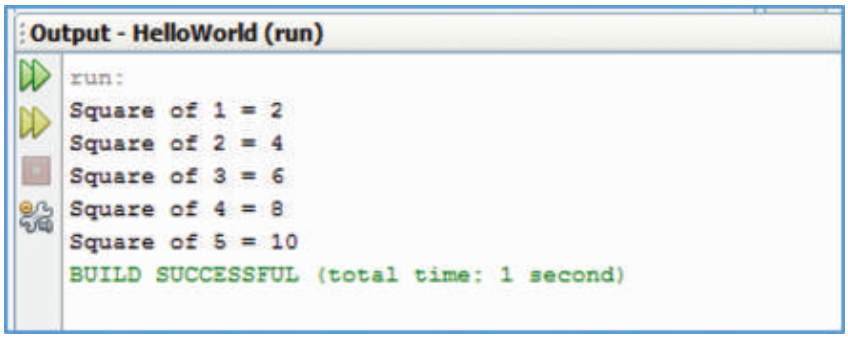


Figure 3.4(f): Output from While Demo Program

STYLE TIP: LOOPS

1. Indent your loops, begin the statements in the body of the loop with a tab/ spaces inside the curly braces. This will make the code easier to read, understand and debug. The NetBeans IDE automatically provides code indentation.

Indented Code	Non Indented Code

`int number = 1;`
`while (number >=10)`
`{`
`  System.out.println(nu`
`  mber);`
`  number = number + 1;`
`}`
	`int number = 0;`
	`while (number >=0)`
	`{`
	`System.out.println(numb`
	`er);`
	`number = number + 1;`
	`}`

2. If there is only one statement in the body of the loop, the set of curly braces enclosing the body can be omitted. For example,

```
while ( number < another_number)
   ++number;
```

3.4.6 The Do While Statement

The do `while` statement evaluates the test after executing the body of a loop. The structure of the Java do `while` statement is as shown:

```
do
{
   statements
} while (expression);
```

Let us write the same program to print the squares of numbers from 1 to 5. This time we will use a do-while loop. The following steps need to be performed.

1. Initialize number = 1
2. Print the square of the number
3. Increment number (number = number + 1)
4. Test if the number <= 5
5. If yes, Go back to step 2
   If no, Exit the loop

The tasks that need to be performed repeatedly are in steps 2 and 3 – these constitute the body of the loop. The test condition is in step 4.The corresponding Java code is as below:

```
public class DoWhileDemo {
   public static void main (String[ ] args) {
   int number = 1;
   do {
      System.out.print ("Square of " + number);
      System.out.println (" = " + number*number);
      ++number;
   }  while (number <= 5);
}
}
```

The output of the `DoWhileDemo` program is the same as Figure 3.4f. You might be wondering if both the `while` and `do-while` loops give the same output, what is the use of having two types of loop constructs. Well, consider the case when the variable number is initialized to 6.

```
int number = 6;
```

The `while` loop would first test whether number `<= 5`. Since `6 > 5`, the condition is `false` and the loop body would not be entered and as a result nothing is displayed in the output window. However in the case of the do-while loop, since the body of the loop is executed before the test condition, the square of 6 is printed (`"Square of 6 = 36"`), then, number is incremented to 7. Now the test for number >= 5 is made. Since `7 > 5`, the condition is false and the loop is exited. To summarize, the `do-while` loop always executes at least once whereas the `while` loop executes zero or more times.

Table 6 summarizes the difference between a `while` and a `do-while` loop.

Table 6 – Difference between while and do-while loop

`while` LOOP	`do while` LOOP
A `while` loop is an entry controlled loop – it tests for a condition prior to running a block of code	A `do-while` loop is an exit control loop - it tests for a condition after running a block of code

A `while` loop runs zero or more times Body of loop may never be executed	A `do-while` loop runs once or more times but at least once
The variables in the test condition must be initialized prior to entering the loop structure.	Body of loop is executed at least once It is not necessary to initialize the variables in the test condition prior to entering the loop structure.

`while (condition)`
`{`
`  statements`
`}`
	`do {`
	`  statements`
	`} while (condition);`

Common Coding Errors: Loops

1. **Infinite Loops**

   This type of error occurs when the loop runs forever, the loop never exits. This happens when the test condition is always true.

   For example,

```
int number = 1;
while (number <= 5)
{
   System.out.print("Square of " + number);
   System.out.println(" = " + number*number);
}
```

   This loop will run forever since number never changes – it remains at 1.

2. **Syntax Errors**

   - Do not forget the semi colon at the end of the test condition in a `do-while` loop. Otherwise you will get a compiler error.
   - Do not place a semi colon at the end of the test condition in a `while` loop.

     For example,

```
int x = 1;
while ( x <=5 );
   x = x +1;
```

   This loop is an infinite loop - The semicolon after the while causes the statement to be repeated as the null statement (which does nothing). If the semi colon at the end is be removed the loop will work as expected.

3.4.7 The for Statement

The for loop is the most widely used Java loop construct. The structure of the Java for statement is as below:

```
for (counter=initial_value; test_condition;change counter)
{
    statements
}
```

Semicolons separate the three parts of a for loop:

- The *initial_value* initializes the value of the loop counter.
- The *test_condition* tests whether the loop should be executed again. The loop is exited when the test condition fails.
- The *step* updates the counter in each loop iteration.

For example, consider the following loop that prints the square of numbers from 1 to 5:

```
for (int number = 1; number<= 5; ++number)
{
    System.out.print("Square of "+ number);
    System.out.println(" = "+ number*number);
}
```

When the loop begins, the variable number is set to 1. Then the `test_condition` tests whether the loop variable, number <=5. If it is, the body of the loop is executed – the square of 1 is printed. The *step* then increments count by 1. The `test_condition` is tested again to decide whether to execute the body. If it is less than 5, the square of the variable number is printed. When number exceeds 5, the loop is exited. The output from the program is the squares of numbers from 1 to 5.

In English, we read the loop above as follows:

for number is equal to 1; number is less than or equal to 5; execute the body of the loop; increment number; loop back to the test condition.

The loop given above counts up the loop index. It is an **incrementing** loop. We can also count down in a loop for a **decrementing** loop. The following decrementing loop will execute 5 times.

```
for (int number = 5; number >= 1; number = number-1)
{
    System.out.print("Square of " + number);
    System.out.println(" = " + number*number);
}
```

The loop index can be incremented or decremented by any value, not just 1. The following loop increments the loop index by 2. It displays all odd numbers between 1 and 20.

```
for ( int count = 1; count <= 20; count = count +2)
{
    System.out.println(count);
}
```

The loop index can begin with any value, not necessarily 1. The following loop also iterates 5 times and prints number from 5 to 9.

```
for ( int count = 5; count < 10; count++)
{
    System.out.println(count);
}
```

The counter in the loop iterates with count values – 5,6,7,8,9 for a total of five times. Notice that the test condition is count < 10 instead of count <= 10. If it was the latter, the loop would have iterated 6 times for loop index count as 5,6,7,8,9,10.

STYLE TIP: For LOOP

1. If there is only one statement in the body of the loop, the set of curly braces enclosing the body can be omitted. For example,

```
for ( int count = 5; count < 10; count++)
    System.out.println (count);
```

2. You can use multiple items in the initialization and the updation part of the loop by separating them with the comma operator.

   For example,

```
for ( x = 0, y = 10; x < 10; x++, y--)
    System.out.println("x = " + x + "y = " + y);
```

3. Do remember to use indentation to align the body of the loop, it will make it easier for you to debug your code.

Common Coding Errors: The for Loop

1. Initial value is greater than the limit value and the loop increment is positive.

   For example,

```
for ( int count = 5; count <= 1; count++)
```

   In this case, body of the loop will never be executed

2. Initial value is lesser than the limit value and the loop increment is negative.

   For example,

```
for ( int count = 1; count >= 5; count --)
```

   In this case also, body of the loop will never be executed.

3. Placing a semicolon at the end of a for statement:

```
for ( int count = 1; count <= 5; count++);
```

```
{
    //body of loop
}
```

This has the effect of defining the body of the loop to be empty. The statements within the curly braces will be executed only once (after the for loop terminates) and not as many times as expected.

4. Executing the loop either more or less times than the desired number of times.

   For example, the following loop iterates 4 times and not the intended 5 times because it exits when count = 5.

   ```
   for ( int count = 1; count < 5; count ++)
   ```

   The correct way to loop five times would be to test for count **<=** 5.

   Such errors are known as off by one errors.

5. Using a loop index (declared within the loop) outside the loop.

   ```
   for ( int count = 1; count < 5; count ++)
   {
       System.out.println(count);
   }
       System.out.println(count);    //error!!
   ```

   The scope of the variable count is only within the body of the loop. It is not visible outside the loop.

3.5 Arrays

Previously you learned about variables that are place holders for a single value. Arrays are variables that can hold more than one value, they can hold a list of values of the same type. For example, to store the marks of a student, we used a variable:

```
double marks_obtained = 346;
```

To store the marks of another student, we changed the variable `marks_obtained` as follows.

```
marks_obtained = 144;
```

By changing `marks_obtained`, we have lost information about the first student's marks. What if we wanted to save the first student's marks for later processing? We would need two variables, say marks1 and marks2. But, what if there were 5 students in the class. We would need 5 variables!

```
double marks1 = 346, mark2 = 144, marks3 = 243,
marks4=256.5, marks5 = 387.5;
```

Java provides a simpler way to store the marks of 5 students. We use an array and define it as below:

```
double[] marks;
```

The [] brackets after the double data type tell the Java compiler that instead of a single value, the marks variable is an array that can hold more than one value, each of type `double`. To specify that the array can hold up to 5 marks, we specify the size of the array using the keyword new as shown below:

```
marks = new double[5];
```

The two statements – declaring an array and specifying its size can also be done in one statement.

```
double[] marks = new double[5];
```

Now we can store five marks in the array, each element of the array is indexed by its position, starting from 0. So the five elements of the array are available at positions 0 to 4 as given below:

```
marks[0], marks[1], marks[2], marks[3], marks[4]
```

Note that the array index goes from 0 to n-1 for an array of size n and not from 1 to n. We can initialize the array **statically** (that is at compile time) as shown below:

```
double[]marks = {346, 144, 103, 256.5, 387.5};
```

The statement above creates an array of double of size 5, and elements are assigned with the numbers given in the curly braces. Components of an array may be assigned value using an assignment statement. For example,

```
inti i = 3;
marks[3] = 210.75;
```

To print all the marks, we can use a `for` loop, varying the loop index from 0 to 4.

```
for (int i = 0; i< 5; i++)
   System.out.println(marks[i]);
```

An array variable has a useful property, the `length property`, which is the size of the array. For example, for the marks array, `marks.length` will be 5. We can thus code the previous for loop to print all the elements of an array as:

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

The following code shows the use of an array to print the class report card for the five students.

```
for (int i = 0; i <marks.length; i++) {
   double percentage = (marks[i]/total_marks)*100;
   String result;
   if (percentage >= 40)
      result = "Passed";
   else
      result = "Failed";
   System.out.print((i+1)+"\t");
```

```
System.out.print(marks[i]+"\t");
System.out.print(percentage+"\t\t");
System.out.println(result);
}
```

Notice the use of `\t` to print a tab between the numbers in the print statement. Also note that the roll number of the student has to be printed as (i+1) since the array index begins from 0 but roll numbers begin from 1. Figure 3.5(a) shows the complete program listing and Figure 3.5b displays the output from the `ArrayDemo` program.

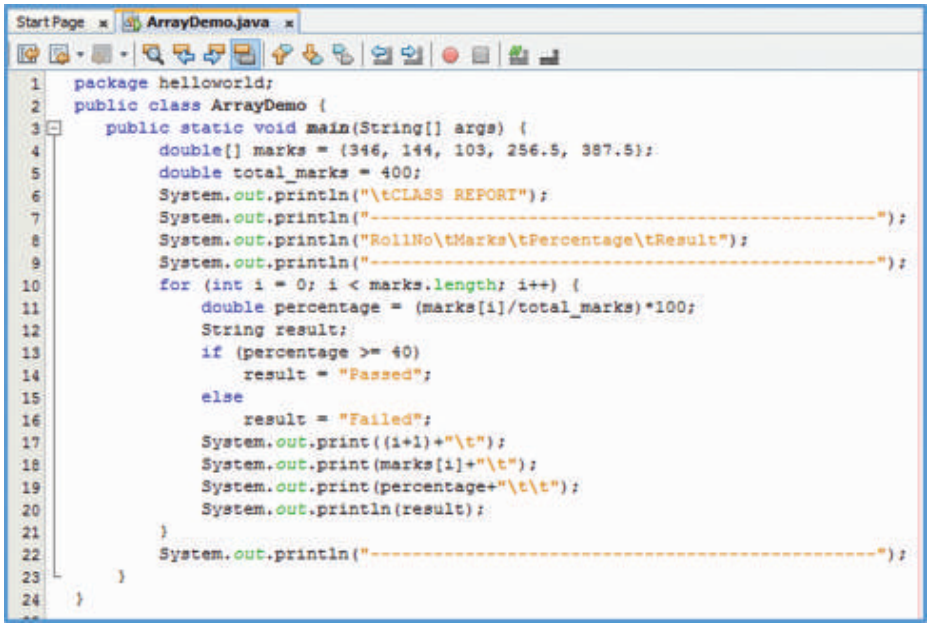


Figure 3.5(a): Array Demo Program

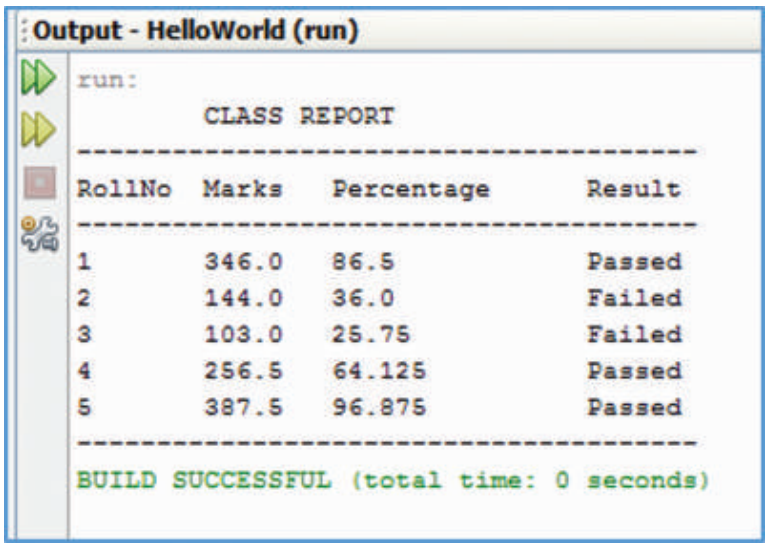


Figure 3.5(b): Output from Array Demo Program

Common Coding Errors: Arrays

1. In an off by one error, the loop index if varied from 0 to n instead of 0 to n-1. For example, if the array size is 5, then `loop index = 5` is an off by one error since array index can go only from 0 to 4.

```
double [] ] marks = new double [5];
for (int i = 0; i <= 5; i++) {
   System.out.print(marks[i]);
}
```

In this case an Array index out of bounds error occurs and the program terminates unexpectedly with an error as below.

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at javaprograms.ArrayDemo.main(ArrayDemo.java:11)
Java Result: 1
```

3.6 User Defined Methods

A method in Java is a block of statements grouped together to perform a specific task. A method has a *name*, a *return type*, an optional *list of parameters*, and a *body*. The structure of a Java method is as below:

```
return_type method_name(list of parameters separated by commas)
{
    statements
    return statement
}
```

The method name is a word followed by round brackets. The parameter list placed within the round brackets allows data to be passed into the method. This list is a comma separated list of variables (as many as needed) along with their data types. The variables of the parameter list act just like regular local variables inside the method body. The method body is enclosed within a pair of curly braces and contains the statements of the method. The statements end with an optional `return` statement that returns a value back to the calling method. The return type is the type of the value that the method returns.

Let us write a method that given the `length` and `breadth` of a rectangle as parameters returns the `area` of the rectangle.

```
static double rectangle_area (double length, double breadth)
{
    return (length * breadth);
}
```

The name of the method is `rectangle_area`. The method has two parameters – both of type `double`. The body of the method has only a single statement, one that calculates the area based on the parameters and returns it. The return type from the method is of type

double. We have declared this method as a **static** method. The static modifier allows us to call this method without an invoking object. (You will learn more about objects later).

A method is called/invoked from another method. When a method is called, control is transferred from the calling method to the called method. The statements inside the called method's body are executed. Control is then returned back to the calling method.

Let us call/invoke the rectangle_area method from the main method.

```
public static void main(String[] args) {
    double a = 0;
    a = rectangle_area(45.5, 78.5);
    System.out.println("Area of the rectangle = "+ a);
}
```

When we call the rectangle_area method, we supply the length and breadth of the rectangle as **arguments** in the method call. The arguments with which the rectangle_area method is called are copied into its parameters. In this case, 45.5 is copied into the parameter length and 78.5 is copied into the parameter breadth.

```
double rectangle_area (double length, double breadth)

    double a = rectangle_area (45.5, 78.5);
```

The method is executed and the return value from the method is copied into the variable a. In this case, a gets the value 3571.75, the product of 45.5 and 78.5.

```
double rectangle_area (double length, double breadth)
3571.75
    double a = rectangle_area(45.5, 78.5);
```

The System.out.prinltn() then displays the area in the output window.

```
Area of the rectangle = 3571.75
```

If we can call the rectangle_area method with different arguments and we will get a different result.

Figure 3.6(a) shows the complete program listing of the MethodDemo program and Figure 3.6(b) displays the output from the program.

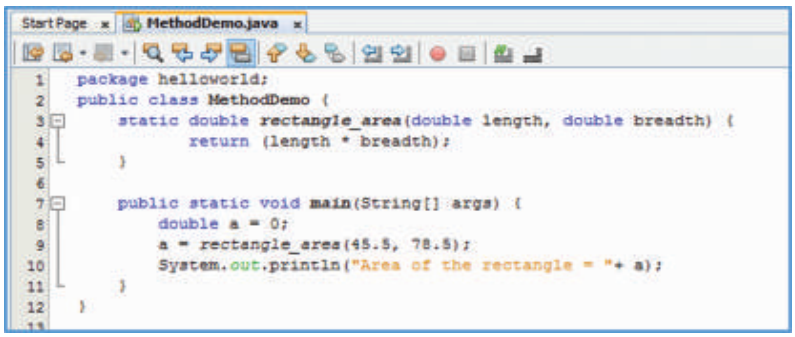


Figure 3.6(a): User Defined Method Demo Program

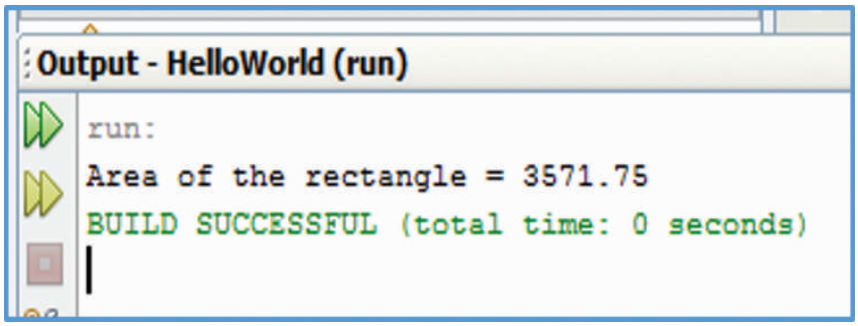


Figure 3.6(b): Output from User Defined Method Demo Program

A special return type `void` can be used if a method does not return any value. For example, a method that just prints a string need not have a return value can use `void` as the return type.

```
void print_message(String message) {
   System.out.println("The message is: "+ message);
}
```

This method can be called with a statement such as

```
print_message("This is a secret message!");
```

3.7 Object Oriented Programming

By now you know quite a bit of Java programming. Now we begin to look at Java's most fundamental feature – **Classes** and **Objects**. Java is an **Object Oriented Programming (OOP) language**. In an OOP language, a program is collection of *objects* that interact with other objects to solve a problem. Each object is an instance of a *class.*

Imagine you are creating a database application for a bookstore. You will need to store data about all the books in the store. So, each book will become an object in your program. Further, each book will have certain characteristics or attributes such as its Title, Author, Publisher, and Price. You may also want to perform certain actions on a book such as display its details on the computer screen or find its price. In an OOP language, such as Java, an entity such as a book in the example is referred to as a **class**. A class is a physical or logical entity that has certain attributes. The title, author, publisher, genre and price are the **data members** of the class Book. Displaying book details and getting its price are **method members** of the class Book. Method members can get, set or update the class data members.

Class Book

Data Members:

Title

Author

Publisher

Genre

Price

Method Members:

display()

getPrice()

To describe one particular book in the library, you would create an **object** of the class book and fill in all its data members. For example one book in the library would be an object populated with the following data members:

Book: book1

Title: Game of Thrones

Author: George R Martin

Publisher: Harper Collins

Genre: Fiction

Price: 450

Another book would be another object populated with the following data members:

Book: book2

Title: Fundamentals of Database Systems

Author: Shamkant Navathe

Publisher: Pearson

Genre: Educational

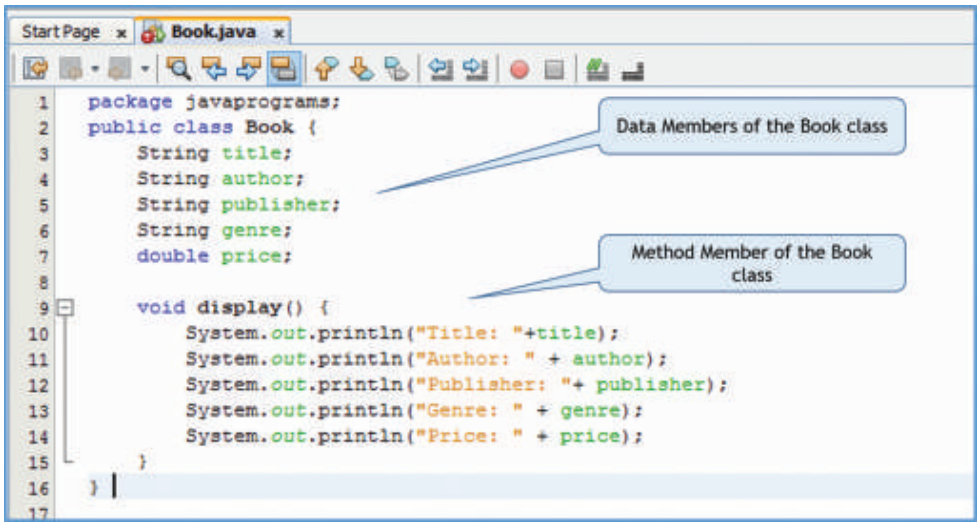
Price: 400

You can think of a class as a template from which objects are created. All objects of the same class have the same type of data members.

Method members of a class are invoked on an object to perform the action associated with that method. For example, to display the details of book1 you would call the method member `display()` of the class with book1 as the invoking object. To display the details of book2 you would call the same method member `display()` but with the invoking object as book2.

3.8 Class Design

We are now ready to create a class in Java. A class in Java begins with the keyword `class` followed by the name of the class. The body of the class is enclosed within curly braces. The body contains the definitions of the data and method members of the class. The data members are defined by specifying their type. The method members of the class are defined just like the user defined methods we saw earlier. The method members have access to all the data members of the class and can use them in their body. The data members of a class are like **global** variables – they can be accessed by all the method members of the class. Figure 3.8(a) shows how to code the `class Book` in Java.




```
1  package javaprograms;
2  public class Book {
3      String title;
4      String author;
5      String publisher;
6      String genre;
7      double price;
8
9      void display() {
10         System.out.println("Title: "+title);
11         System.out.println("Author: " + author);
12         System.out.println("Publisher: "+ publisher);
13         System.out.println("Genre: " + genre);
14         System.out.println("Price: " + price);
15     }
16 }
17
```

Figure 3.8(a): The Book Class

3.8.1 Constructors

A data member that is declared but not initialized before using, is assigned a default value by the compiler usually either zero or null. However, it is good programming practice to always initialize variables before using them.

A special method member called the **constructor** method is used to initialize the data members of the class (or any other initialization is to be done at time of object creation). The constructor has the same name as the class, has no return type, and may or may not have a parameter list. Whenever a new object of a class is created, the constructor of the class is invoked automatically. We do not call the constructor explicitly.

Let us define a constructor for the `Book class` that takes parameters corresponding to each data member and uses the parameters to initialize the data members Figure 3.8(b).

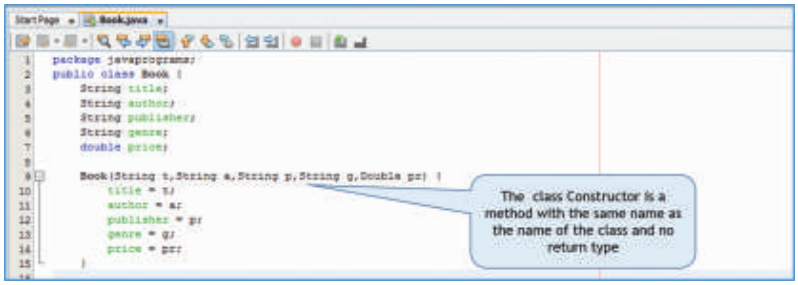


Figure 3.8(b): The Book Class Constructor

We can now use our constructor to create an object of the `Book class`.

```
Book book1 = new Book("Game of Thrones",
                      "George R Martin",
                      "Harper Collins",
                      "Fiction", 450.0);
```

This has the effect of creating a new object `book1` with its data members set as per the parameter list of the constructor.

Note that once we have defined a parameterized constructor, we will not be able to create a `Book object` without any parameters, that is, with a statement like:

```
Book book3 = new Book();
```

The compiler will complain that the methods actual and formal arguments differ in length. If you want to still be able to create a book object without specifying any parameters, you will have to also define a parameter-less constructor as below.

```
Book() {
   title = "";
   author = "";
   publisher = "";
   genre = "";
   price = 0;
}
```

The parameter-less constructor above initializes all the data members of the book with default values (null string in case of `String` parameters and 0 for the `double` parameter).

Once the class is defined, we can create objects of the class and access its members.To create an object of a class we use the keyword new. We can create objects of the `class Book` in a Java class file of our package. So first we create a new Java class file, called `BookTest.java`, in our package. Then in the main method of the `BookTest class`, we create a `Book object` Figure 3.8(c). The following statement creates an object named book1 of the `class Book`.

```
   public static void main(String[] args) {
   Book book1 = new Book();
}
```

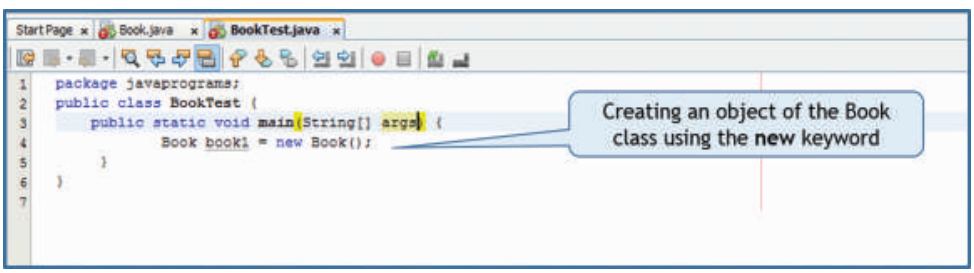


Figure 3.8(c): The BookTest Program

Data members and method members of an object are accessed using the dot (.)operator. As soon as you type the name of the object followed by a dot, `NetBeans` will prompt you with a list of available data and method members that you can access Figure 3.8(d).

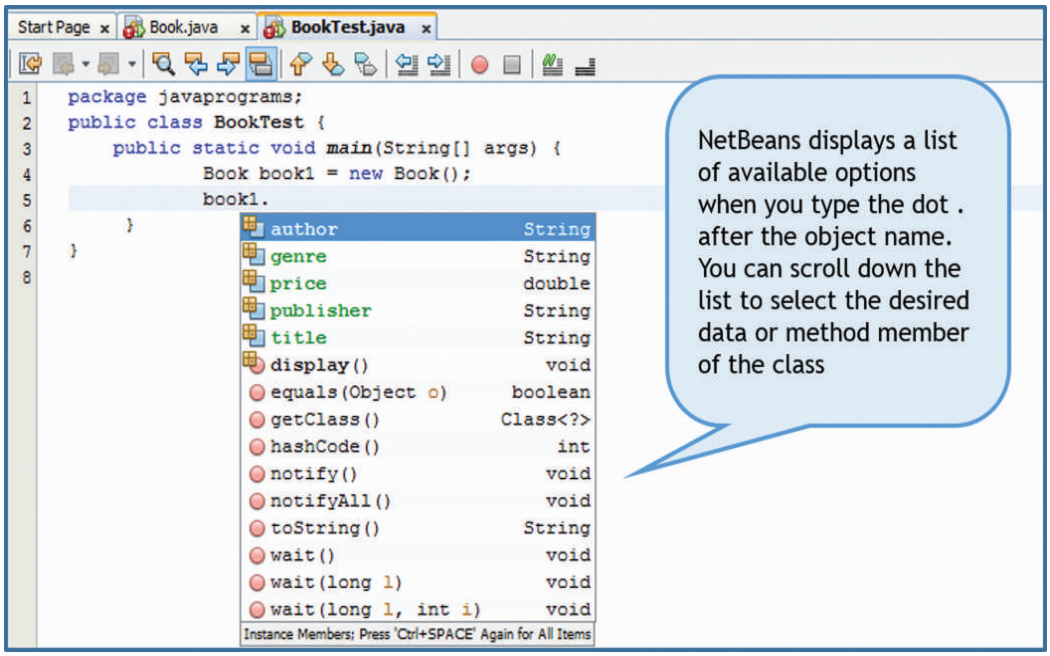


Figure 3.8(d): List of available members of the Book Class

Note that the data members and method member that you defined in the Book class are available in the list. Also note that a number of methods that you did *not* define in the Book class, such as, `hashCode()`, `wait()` and so on, are also available in the list. This is because all Java classes inherit from the base class **Object**. These extra methods are defined in the base class and are therefore available to all Java classes. Objects of `class Book` can invoke the method member defined in the class Figure 3.8(e). For example, in the following statement, the object book1 invokes the method member `display()`.

```
book1.display();
```

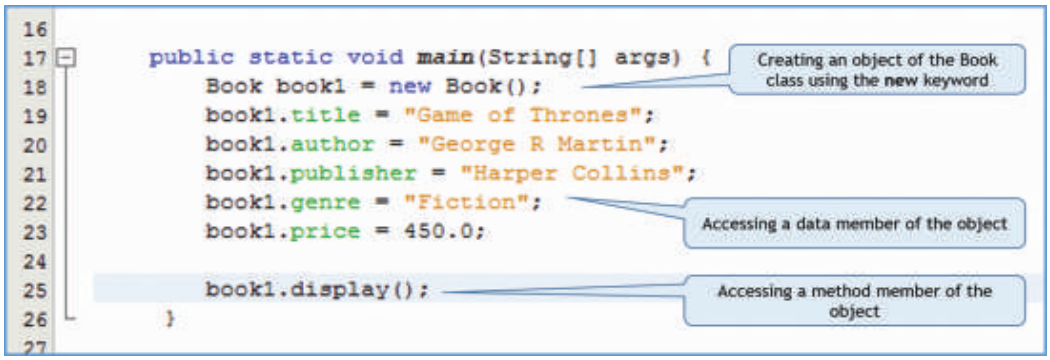


Figure 3.8(e): Accessing the Members of an Object

The `display()` method as per its method body prints the data members of its invoking object, in this case `book1`. Figure 3.8(f) displays the result of executing the `BookTest` program.

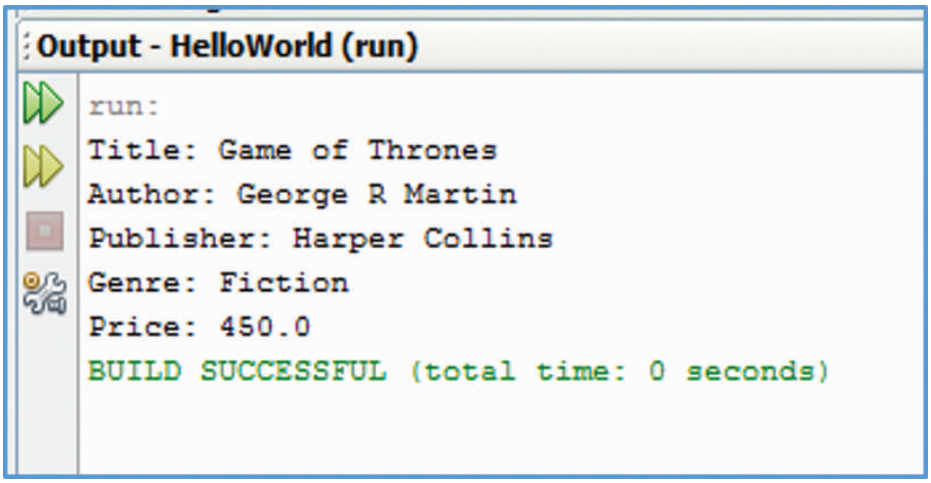


Figure 3.8(f): Output from the BookTest Program

In the `BookTest.java` program, try to create another object book2, initialize its data members, and then call `book2.display()`. Run the modified program to verify that now `book2`'s data members are displayed.

3.8.2 Access Modifiers

Data members of a class can be accessed from outside the class by default. However, it is generally not good programming practice to allow data members to be accessed outside the class. By allowing objects to change their data members arbitrarily, we lose control over the values being held in them. This makes debugging code harder and our code vulnerable to security issues. To make a data member or a method member of a class visible only within the class, we add the keyword `private` before its declaration Figure 3.8(g).

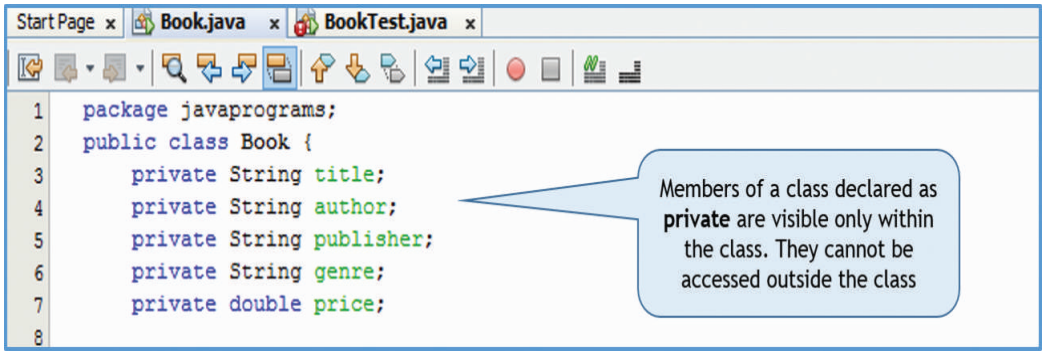


Figure 3.8(g): Private Access Modifier

Private members of a class cannot be accessed outside the class. Declaring the data members of the `Book class` private, we won't be allowed access to them in the `BookTest` class Figure 3.8(h).

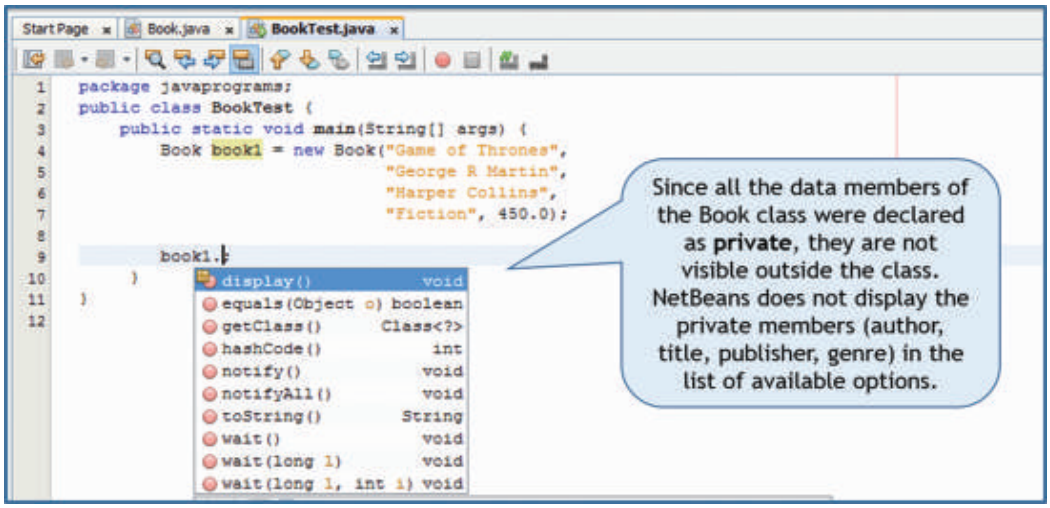


Figure 3.8(h): Private Data Members are not visible outside the class

As opposed to private, a public member of the class has visibility both within and outside the class. By default, all members of a class are public.

3.8.3 Getter and Setter Methods

Private data members of a class cannot be accessed outside the class however, you can give controlled access to data members outside the class through getter and setter methods. A getter method returns the value of a data member. For example we could define a getter method in the `Bookclass` for the `price` data member as given below:

```
double getPrice ( ) {
   return price;
}
```

Similarly, we define a setter method but control how the `price` is set. We do not allow a book `price` to become lower than 100.0.

```
void setPrice(double newprice) {
   if (newprice < 100)
      System.out.println("Price cannot be set lower than
      100!");
   else
      price = newprice;
}
```

In the `BookTest class`, the getter and setter methods can be used to retrieve and set a new price for a book (Figure 3.8(i)). Let's say the `BookTest` class wants to discount the `price` of the `book1` object by 20%, we first get the book's price using the `getPrice()` method, use the price to determine a discounted price for the book, and then use the **`setPrice()`** method to change the price of the book. The code for the same is as follows:

```
double bookprice = book1.getPrice();
bookprice = bookprice - 20*bookprice/100;
book1.setPrice(bookprice);
```

Now `book1's` data member `price` is set to 360.0 Figure 3.8(j). However if a discount caused a book object's price to fall below 100, the setter method would not change the `price`. Without a controlled setter function, the `BookTest` class could have changed the book's price arbitrarily.

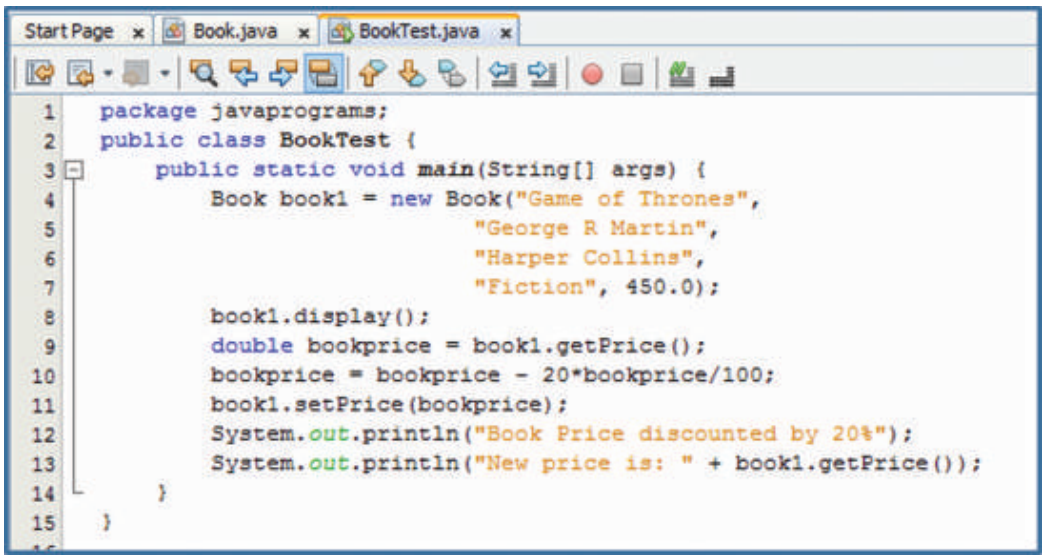


Figure 3.8(i): Use of Getter and Setter Methods

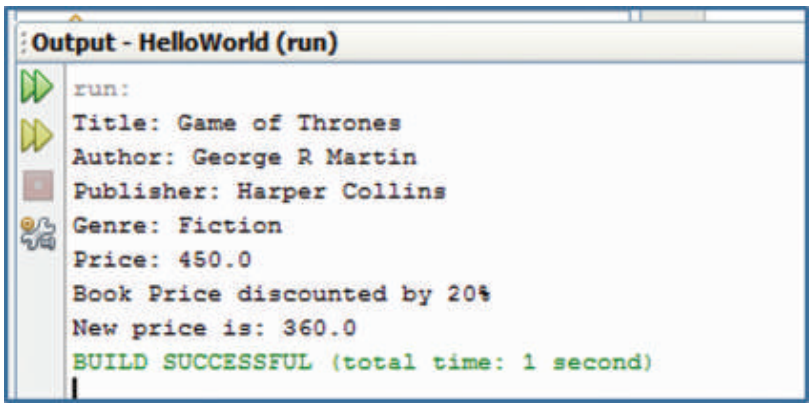


Figure 3.8(j): Output from use of Getter and Setter Methods

3.9 Java Libraries

The power of Java comes from the hundreds of Java classes that are already prebuilt and can be used in your programs. To use a prebuilt class and associated methods in those class, all you have to do is to use the keyword **import** to import the class from the package in which it is contained into your space. The `import` statements must appear before any class definitions in the file. In the following sections, we will look at some useful classes.

3.9.1 Data Input

The programs we have written up to now haven't been interactive. A program is interactive if it is able to take input from the user and respond accordingly. Let us write a program to take user input. To take user input we use the prebuilt Scanner class. This class is available in the `java.util` package. First we import this class,

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

Now we create an object of the class `Scanner`. The constructor of this class requires the source from which input is to be taken. Since we will take input from the console, we use the `System.in` object.

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

Then, we invoke the `next()` method of the Scanner class that returns the token read from the input stream as a `String` object.

```
String name = user_input.next();
System.out.println("Hello "+ name);
```

To read numeric data input, again a Java prebuilt class comes to our rescue. This time we use the `Integer class`. The class has a static method `parseInt()` that takes a `String` as parameter and returns the equivalent integer.

```
String age_string = user_input.next( );
int age = Integer.parseInt(age_string);
System.out.print("In 5 years you will be "+ (age +5));
System.out.println(" years old.");
```

Since the Integer class is in the `java.lang` package, which is already imported for all classes, we do not need to import the package. Similar to the Integer class, there is the **Float** class with a `parseFloat()` method, a `Double` class with a `parseDouble()` method, and other such classes that you can use to read a `String` and convert it to the required data type.

Start Page | DataInputDemo.java

```
1  package javaprograms;
2  import java.util.Scanner;
3  public class DataInputDemo {
4
5      public static void main(String[] args) {
6          Scanner user_input = new Scanner( System.in );
7          System.out.print("Enter your name: ");
8          String name = user_input.next( );
9          System.out.print("Enter your age: ");
10         String age_string = user_input.next( );
11         int age = Integer.parseInt(age_string);
12
13         System.out.println("Hello "+ name);
14         System.out.print("In 5 years you will be "+ (age +5));
15         System.out.println(" years old.");
16     }
17 }
```

Figure 3.9(a): Data Input Demo Program

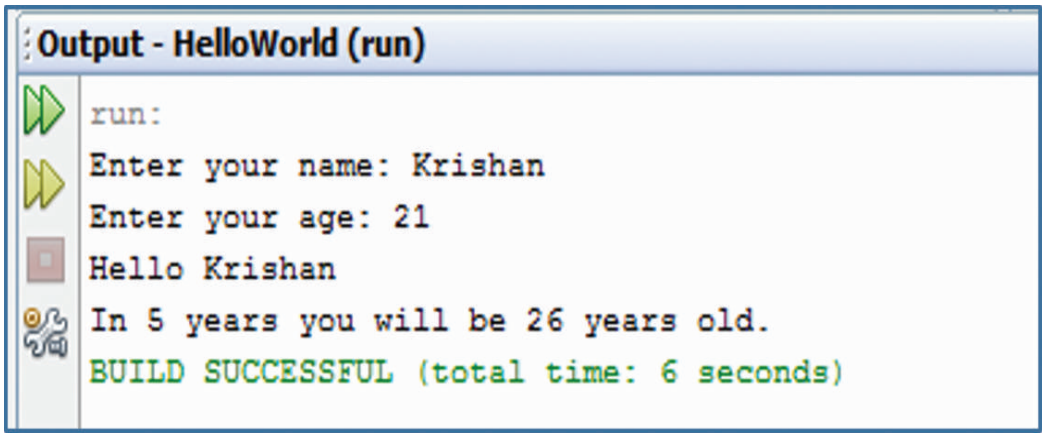


Figure 3.9(b): Output from the Data Input Demo Program

Figures 3.9(a) and 3.9(b) show the Data Input Demo program and its output respectively.

3.9.2 Array Manipulation

In the previous section you learned about some useful prebuilt classes for data input. In this section, we will explore the prebuilt Arrays class from the `java.util` package for array handling. The `Arrays class` has a number of useful methods. Let us start by using the `sort()` method to sort an array of integers in ascending order.

First we **import** `java.util.Arrays` class. Then in the `main()` method, we invoke the `Arrays.sort()` method on the array we need to sort.

```
double[] marks = {103, 144, 256.5,346, 387.5};
Arrays.sort(marks);
```

The marks array after sorting becomes = {103.0, 144.0, 256.5, 346.0, 387.5}. Sorting makes it easier for us to find the lowest and highest marks obtained by a student. To print the lowest marks, we can now write

```
System.out.println(marks[0]);
```

To print the highest marks,we can write

```
System.out.println(marks[marks.length-1]);
```

The same method can be used to sort an array of Strings in alphabetic order.

```
String[] names =
{"Sarthk","Saumya","Mayank","Mudit","Shiva","Anju",
"Savita"};
Arrays.sort(names);
```

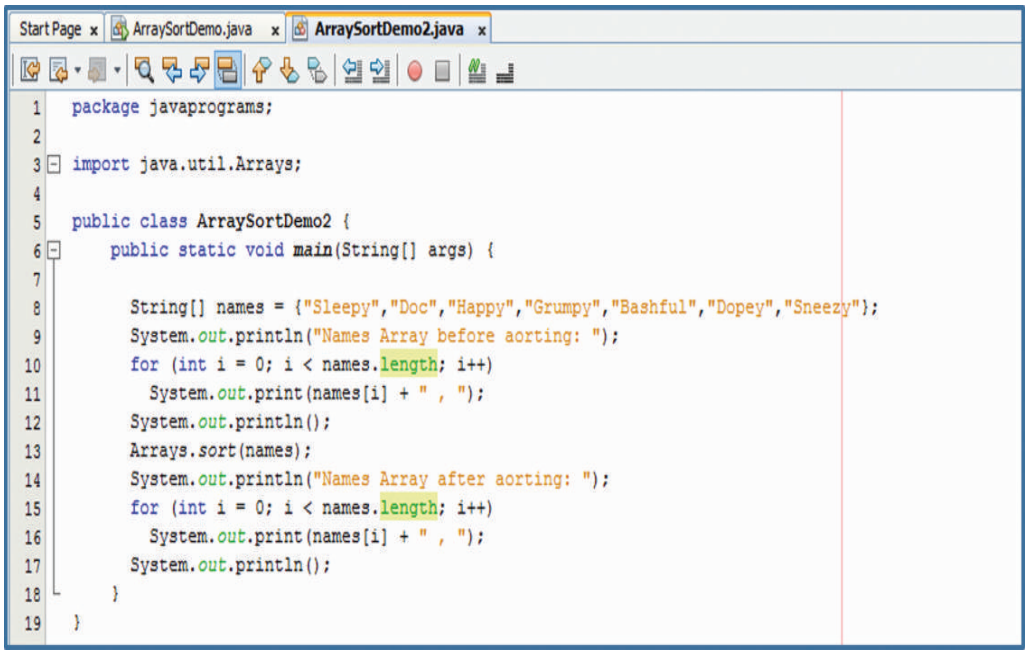


Figure 3.9(c): The ArraySortDemo2 Program

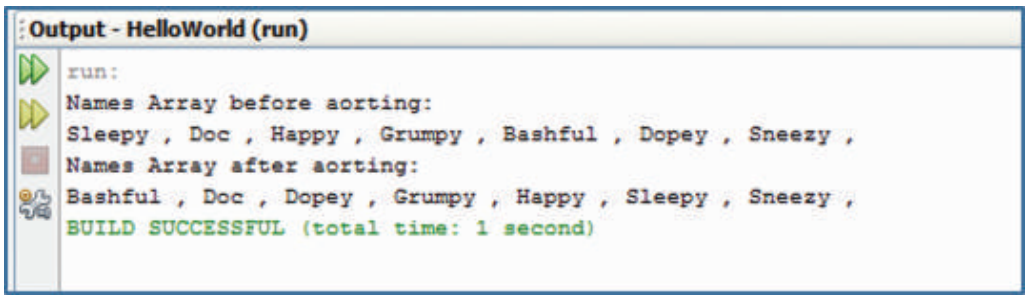


Figure 3.9(d): Output from the ArraySortDemo2 Program

Figures 3.9(c) and 3.9(d) show the `ArraySortDemo2` program and its output respectively.

The `binarySearch()` method of the `Arrays` class helps us search for a specific element in the array. The parameters it needs are the array to be searched and the key element to be searched. The method returns the index of the array where the key is present. If the key is not found in the array, the `binarySearch` method returns -1.

```
double[] marks = {103, 144, 256.5, 346, 387.5};
int key = 346;
int index = Arrays.binarySearch(marks,key);
```

Note that the array must be sorted before invoking `binarySearch()`. If it is not sorted, the results are undefined

```
Start Page | ArraySearchDemo.java

1  package javaprograms;
2
3  import java.util.Arrays;
4  import java.util.Scanner;
5  public class ArraySearchDemo {
6      public static void main(String[] args) {
7        double[] marks = {346, 144, 103, 256.5, 387.5};
8        Scanner user_input = new Scanner( System.in );
9        System.out.print("Enter number to search: ");
10       int key = Integer.parseInt(user_input.next());
11       int index = Arrays.binarySearch(marks,key);
12       if (index != -1)
13          System.out.println("Element found at position "+ (index+1));
14       else
15           System.out.println("Element not found");
16     }
17 }
```

Figure 3.9(e): The ArraySearch Demo Program

```
Output - HelloWorld (run)

run:
Enter number to search: 103
Element found at position 3
BUILD SUCCESSFUL (total time: 16 seconds)
```

Figure 3.9(f): Output from the Array Search Demo Program

Figures 3.9(e) and 3.9(f) show the `ArraySearchDemo` program and its output respectively.

You can visit **http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html** to explore all the methods available for the Array class.

3.9.3 String Manipulation

In the previous section, we learned to manipulate arrays using Java prebuilt classes. In this section we learn to manipulate Strings using the String class present in the java.lang package.

The first method we will use from the String class is the `toUpperCase()` method. This method converts a string to all uppercase letters.

```
String myString = "Hello World";
System.out.println("UpperCase: " + myString.toUpperCase());
```

The output of the code given above will be:

```
HELLO WORLD
```

The following table shows the commonly used `String` class methods and the result of applying them on the string "`Hello World`".

Method	Description	Application myString = "Hello World"	Output
`char charAt (int index)`	Return the character at the given index	`myString. charAt(6)`	`W`
`String concat (String str)`	Concatenate the specified string at the end of this string	`myString.concat ("Today")`	`Hello World Today`
`boolean contains (String s)`	Returns true if this string contains the specified substring	`myString.contains ("Hell")`	`True`
`boolean endsWith (String suffix)`	Test whether this string end with the given suffix	`myString.endsWith ("old")`	`False`
`boolean equals (Object anObject)`	Compare this string with specified object	`myString.equals ("Goodbye World")`	`False`
`boolean equalsIgnoreCase (String another)`	Compare this string with specified string ignoring case	`myString. equalsIgnoreCase ("hello world")`	`True`
`int indexOf (int c)`	Return the index of the first occurrence of given character	`myString. indexof ('W')`	`6`
`int indexOf (String str)`	Return the index of the first occurrence of given substring	`myString. indexof ("rld")`	`8`
`boolean isEmpty()`	Returns true if the length of this string is 0	`myString. isEmpty()`	`False`
`int length()`	Returns the length of the string	`myString. length ()`	`11`
`String replace(char oldChar, char newChar)`	Returns a new string after replacing all occurrences of oldChar in this string with newChar	`myString.replace ('l','*')`	`He**o Wor*d`
`String replace(String oldStr, String newStr)`	Returns a new string after replacing all occurrences of oldStr in this string with newStr	`myString.replace ("Hello", "Yellow")`	`Yellow World`

`String toLowerCase()`	Converts all of the characters in this String to lower case	`myString.toLowerCase ( )`	`hello world`
`String toUpperCase()`	Converts all of the characters in this String to upper case	`myString.toLowerCase ( )`	`HELLO WORLD`

Figures 3.9(g) and 3.9(h) show the String Demo program and its output respectively.

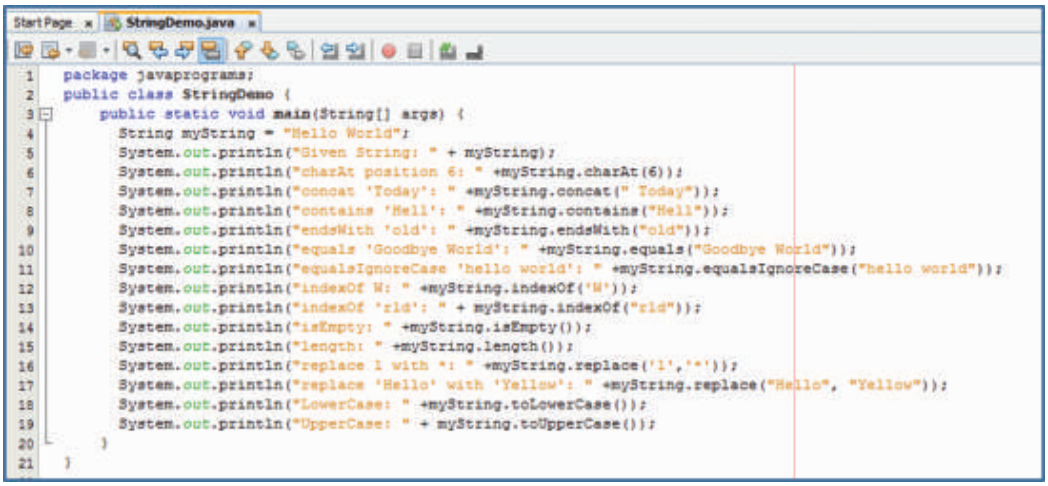


Figure 3.9(g): The String Demo Program

Figure 3.9(h): Output from the String Demo Program

You can visit **http://docs.oracle.com/javase/6/docs/api/java/lang/String.html** to explore all the methods available for the String class.

3.10 Exception Handling

By now you may have written quite a few programs. Some of your programs when executed may have terminated unexpectedly with runtime errors. The errors could have occurred because an array index reference was out of range, or an attempt was made to divide an integer by zero, or there was insufficient computer memory and so on. Such an error situation that is unexpected in the program execution and causes it to terminate unexpectedly is called an `exception`. As a programmer, you should anticipate exceptions in your program that could lead to unpredictable results and handle the exceptions. Consider the program in Figure 3.10(a) that has a method called `divide()`. The method takes two numbers as parameters, divides them and returns the quotient:

```
int divide(int dividend, int divisor) {
   return dividend/divisor;
}
```

Since division by 0 in integer arithmetic causes a program to terminate prematurely, a call to the function divide such as `divide(10,0);` will cause the program to fail Figure 3.10(b).

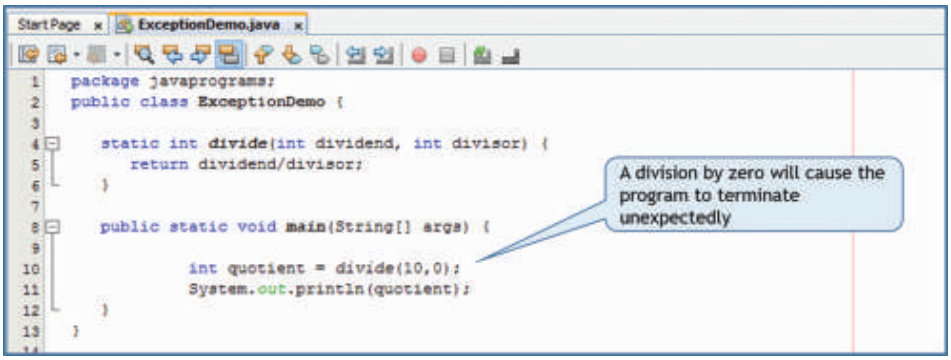


Figure 3.10(a): The Exception Demo Program

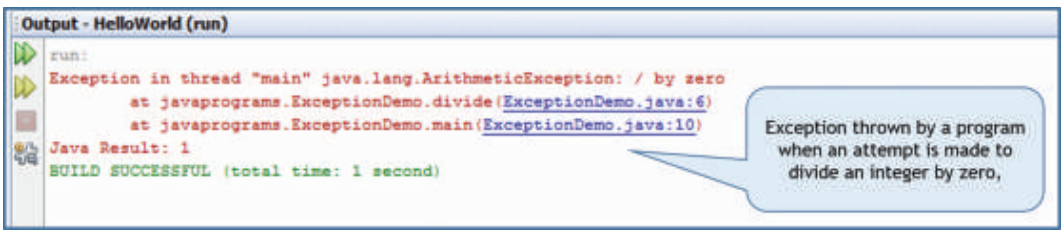


Figure 3.10(b): Output from the ExceptionDemo Program

Note: If the method `divide()` had used floating point numbers instead of integers, there would not have been an error, since in floating-point arithmetic, division by zero is allowed — it results in infinity, which would be displayed as `Infinity`.

Java provides an exception handling mechanism so that a program is able to deal with exceptions, and continue executing or terminate gracefully. The basic idea in exception

handling is to

- Denote an exception block - Identify areas in the code where errors can occur
- Catch the exception - Receive the error information
- Handle the exception - Take corrective action to recover from the error

Java provides the following keywords to handle an exception:

1. **`try`** – A `try` block surrounds the part of the code that can generate exception(s).
2. **`catch`** – The catch blocks follow a `try` block. A `catch` block contains the exception handler - specific code that is executed when the exception occurs. Multiple `catch` blocks following a `try` block can handle different types of exceptions.

The structure of a `try-catch` statement block for exception handling is as below:

```
try {
   // Part of the program where an exception might occur
}
catch (exceptiontype1 argument1) {
   // Handle exception of the exceptiontype1
}
catch (exceptiontype2 argument2) {
   // Handle exception of the exceptiontype2
}
finally {
   //Code to be executed when the try block exits
}
```

The `try` block is examined during execution to detect any exceptions that may be thrown by any statements or any calls to methods within the block. If an exception is thrown, an exception object is created and thrown. The program execution stops at that point and control enters the `catch` block whose argument matches the type of the exception object thrown. If a match is found the statements in that `catch` block are executed for handling the exception.

If no exception is thrown during execution of the statements in the `try` block, the catch clauses that follow the try block are not executed. Execution continues at the statement after the last catch clause.

The optional `finally` block is always executed when the try block exits. This means the `finally` block is executed whether an exception occurs or not. Programmers use this block to put in *clean up* code, for example, freeing up resources allocated in the `try` block.

The following code fragment handles a division by zero exception:

```
try {
      int quotient = divide(10,0);
      System.out.println(quotient);
    } catch (Exception e) {
```

```
System.out.println(e.getMessage());
}
```

We catch an exception of the type `Exception`. An object of the class `Exception` is a "catch all" exception that returns a general error message encountered during program execution. The exception handler in the `catch` block uses the `getMessage()` method of the `Exception` class to print the error that caused the exception.

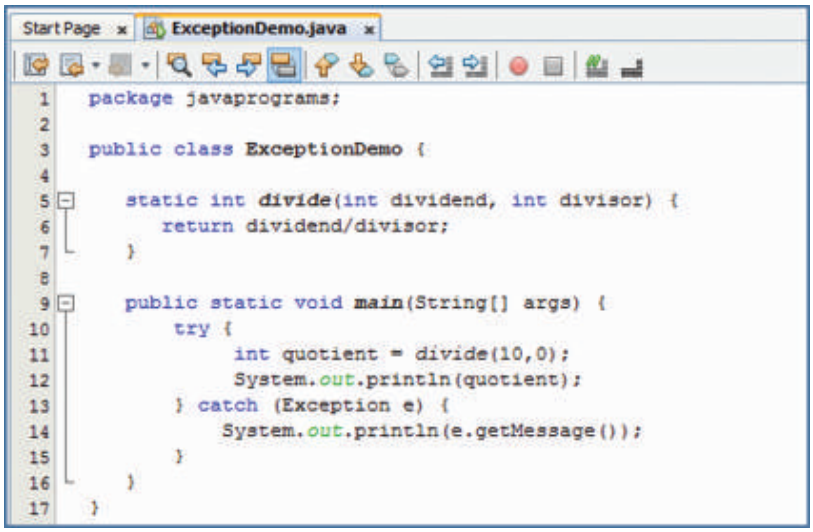


Figure 3.10(c): Exception Handling Demo Program

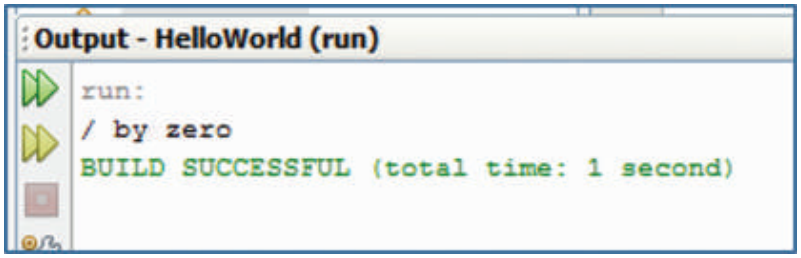


Figure 3.10(d): Output from the Exception Handling Demo Program

Figures 3.10(c) and 3.10(d) show the Exception Handling Demo program and its output respectively.

3.11 Database Connectivity

In this section, we will learn to connect a MySQL database to a Java program and get back the results from executing an SQL query on the database. Connecting a database to a Java program is easy with NetBeans since it allows us to connect directly to a MySQL server.

3.11.1 Connecting to the MySQL Server in NetBeans

We first need to configure NetBeans to Register and connect a MySQL Server. Follow the steps below to do just that.

Step 1: Click on the Services tab located on the left side of the NetBeans IDE. Right

click the **Databases** node and select **Register MySQl Server** (Figure 3.11(a)).

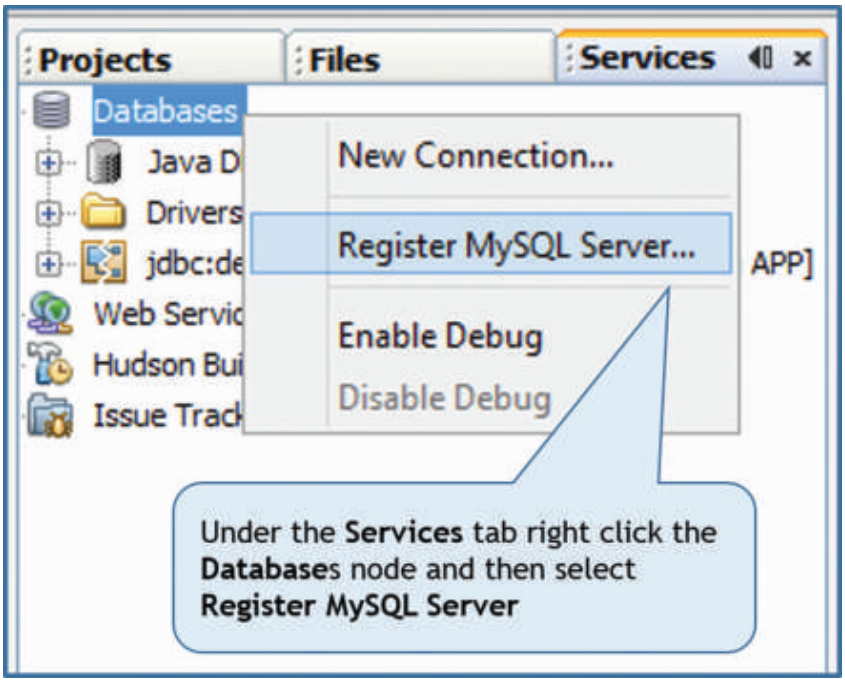


Figure 3.11(a): Register MySQL Server

Step 2: In the **MySQL Server Properties** Dialog Box that opens up, type in the **Administrator User Name** (if not displayed). Also type in the **Administrator Password** for your MySQl Server. Check the Remember Password checkbox and click on **OK** Figure 3.11(b).

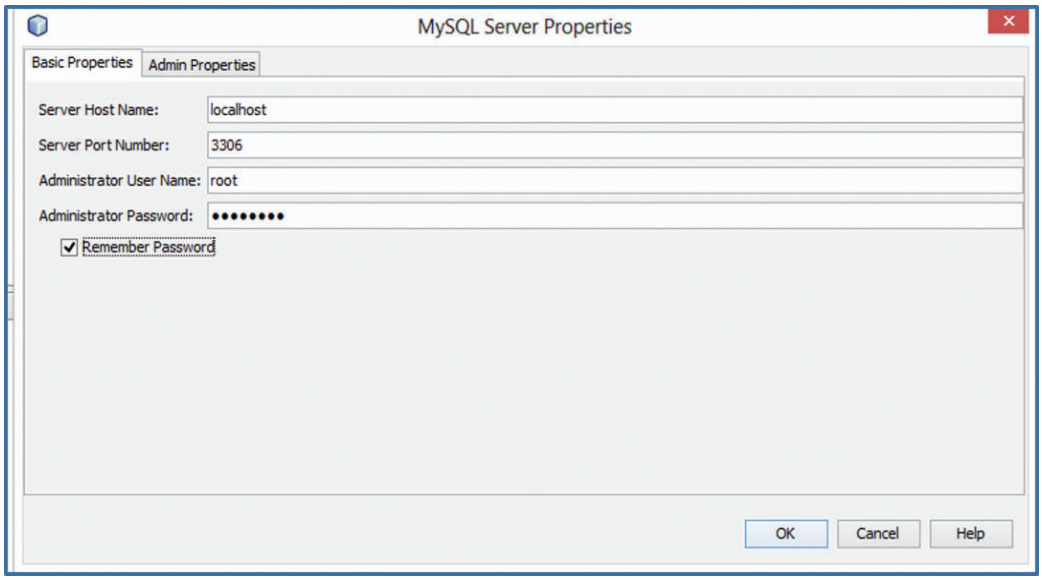


Figure 3.11(b): MySQL Server Properties Window

Step 3: In the same MySQL Server Properties Dialog Box, click on the **Admin**

Properties tab.

In the **Path/URL to admin tool field**, type or browse to the location of your MySQL Administration application **mysqladmin** (you will find it in the bin folder of your MySQL installation directory).

In the **Path to start command**, type or browse to the location of the MySQL start command **mysqld** (you will find it in the bin folder of your MySQL installation directory).

In the **Path to stop command** field, type or browse to the location of the MySQL stop command **mysqladmin** (you will find it in the bin folder of your MySQL installation directory). In the **Arguments** field, type **-u root stop** to grant root permissions for stopping the server.

When finished, the Admin Properties tab should appear similar to Figure 3.11(c). Click on OK.

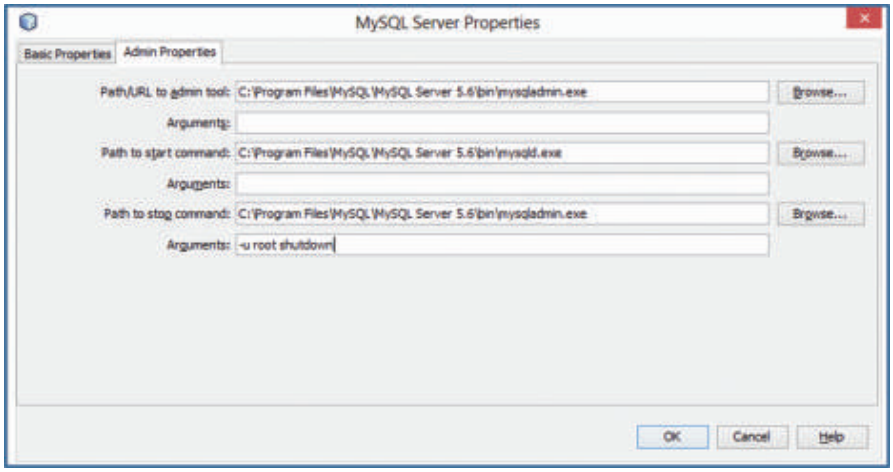


Figure 3.11(c): MySQL Server Properties Window (Admin Properties Tab)

The MySQL Server should now appear under the Database node in the Services tab in the NetBeans IDE Figure 3.11(d). However, it is shown disconnected.

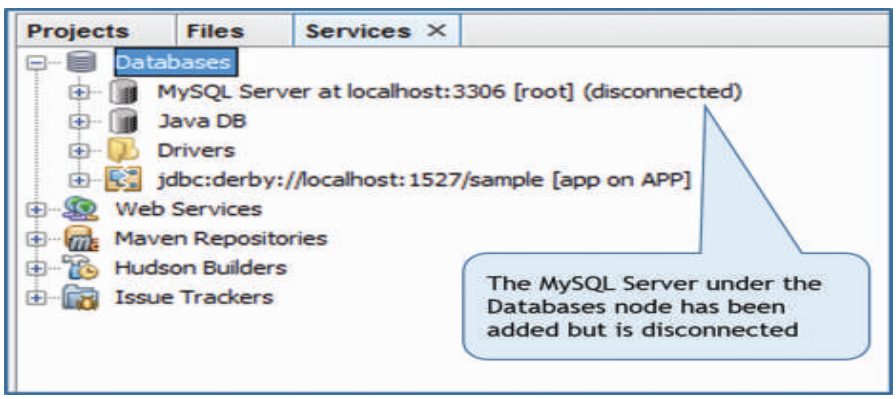


Figure 3.11(d): MySQL Server added to the Databases

Step 4: To connect the MySQL Server to NetBeans, under the Databases node, right click the **MySQL Server at localhost:3306 [root] (disconnected)** and select **Connect** Figure 3.11(e).

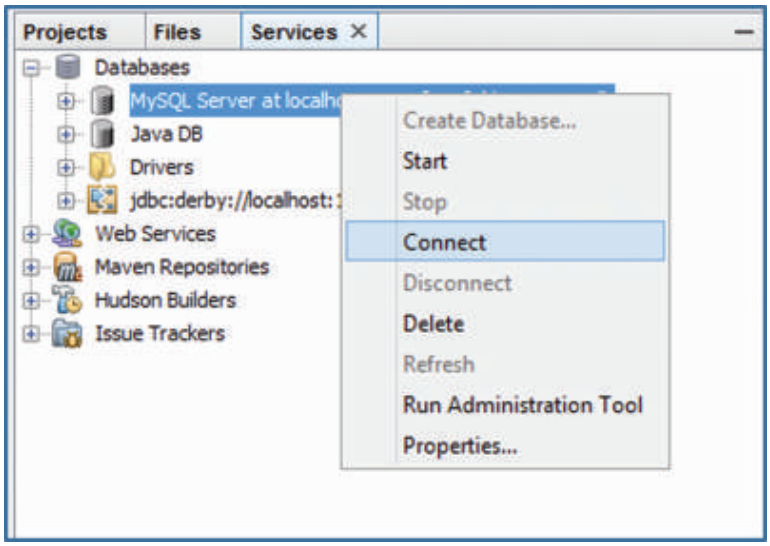


Figure 3.11(e): Connect to the MySQL Server

Note: The MySQL Server should be up and running before you connect to it through the NetBeans IDE. If it is not, start the MySQL Server by running the **mysqld** command from the bin folder of the MySQL installation directory. Then, attempt Step 4.

Step5: When the server is connected you should see the [disconnected] removed from the **MySQL Server at localhost:3306 [root]** database. You should also be able to expand the MySQL Server node by Clicking on the + sign to view all the available MySQL databases Figure 3.11(f).

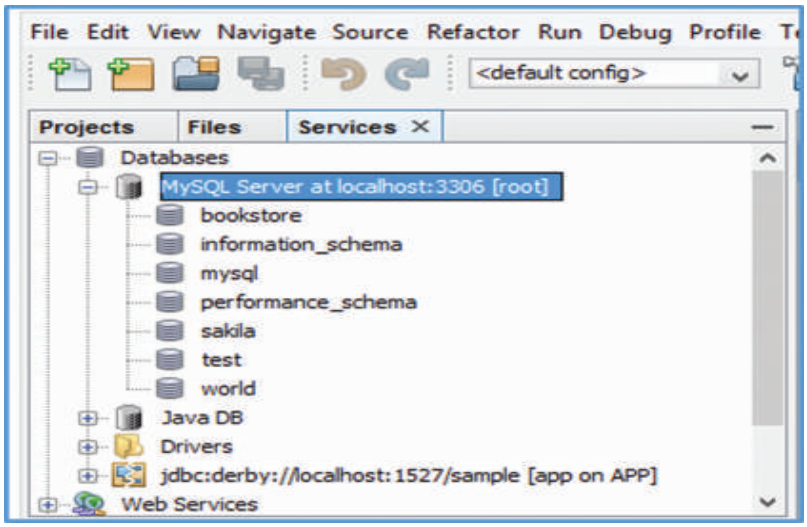


Figure 3.11(f): MySQL databases

That completes connecting the MySQL Server to NetBeans.

3.11.2 Adding the MySQL Connector JAR to the NetBeans Libraries

Before writing the Java program to connect to the database, we also need toadd the mysql connector JAR file to the Libraries in our project. The following steps show you how to do just that:

Step1: Under the **Projects** tab, right click on the **Libraries** node and select **ADD JAR/ Folder…** Figure 3.11(g).

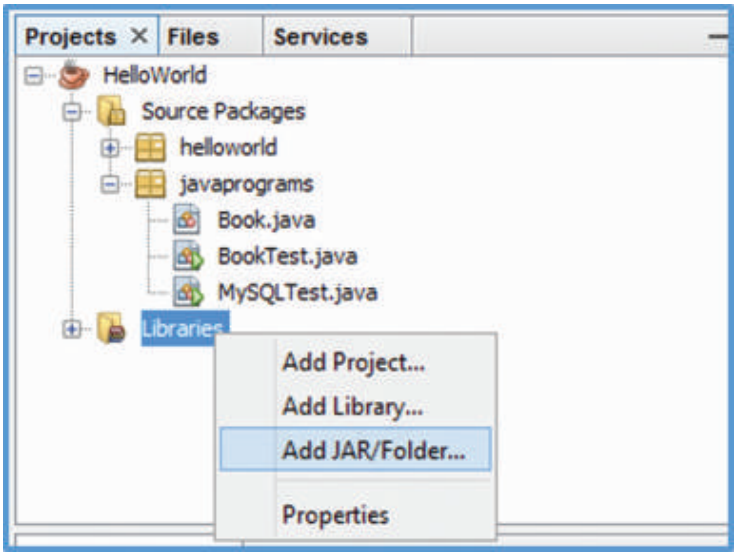


Figure 3.11(g): Add JAR/Folder to Libraries

Step 7: In the **Add JAR/Folder** dialog box that appears, navigate to the your **NetBeans Installation Folder**. Then navigate to the **/ide/modules/ext** folder and select the **mysql-connector-java-5.1.23-bin.jar** file. Click on **Open** Figure 3.11(h).

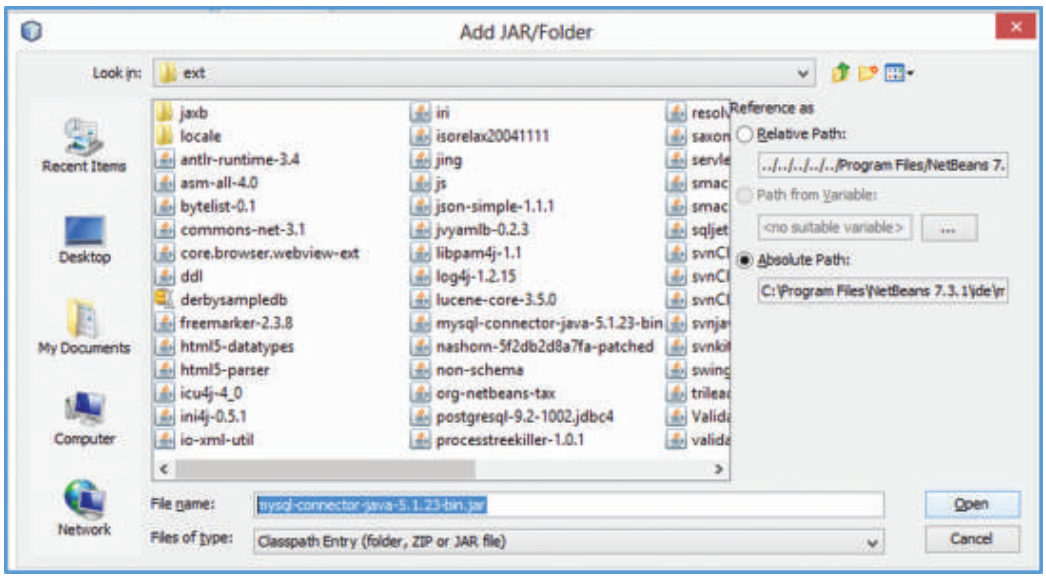


Figure 3.11(h): Add the mysql-connector-java-5.1.23-bin.jar

Now, expand the Libraries node (click on the + sign on its left), the mysql connector jar should have been added to the NetBeans Libraries Figure 3.11(i).

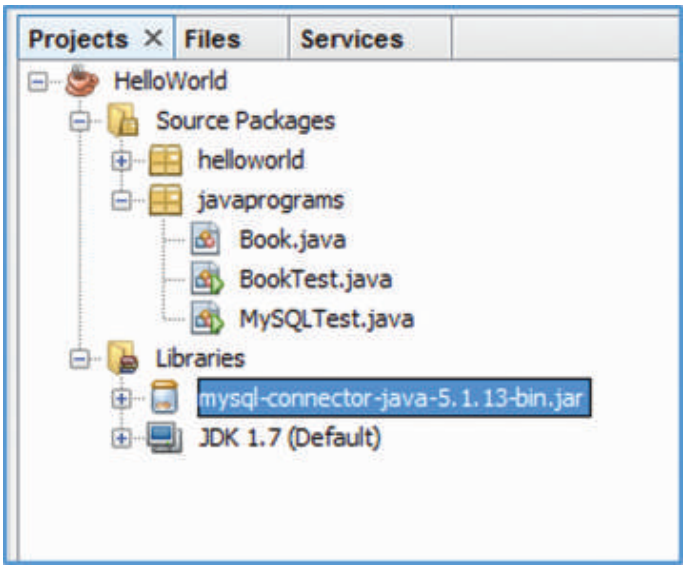


Figure 3.11(i): Libraries

3.11.3 Database Connection from Java

We are now ready to write a Java program to connect to a database and execute a SQL query on the database. In our MySQL database, we have already created a database called bookstore and a table called book within it. The columns of this table are (title, author, publisher, genre, and price). We have also inserted 5 rows into the table Figure 3.11(j).

```
mysql> select * from book;
+---------------------------------+-----------------+-----------------+-------------+-------+
| title                           | author          | publisher       | genre       | price |
+---------------------------------+-----------------+-----------------+-------------+-------+
| Effective Java                  | Joshua Bloch    | Pearson         | Educational |   500 |
| Fundamentals of Database Systems | Shamkant Navathe | Pearson         | Educational |   400 |
| Game of Thrones                 | George R Martin | Harper Collins  | Fiction     |   450 |
| Hold My Hand                    | Durjoy Dutta    | Penguin         | Fiction     |   150 |
| Programming with Java           | E Balagurusamy  | Tata McGraw-Hill | Educational |   220 |
+---------------------------------+-----------------+-----------------+-------------+-------+
5 rows in set (0.00 sec)
```

Figure 3.11(j): Table book from the Bookstore Database

We will now learn how to retrieve data from the MySQL table book in the bookstore database from a Java program.

All the classes that we need are in the java.sql package. So we import the required classes as in Figure 3.11(k).

First, to establish a database connection to the MySQL Server, we invoke the `getconnection()` method of the `DriverManager` class. This method needs three parameters –URL of the database, username, password to connect to the database.

Each database driver has a different syntax for the URL. The MySQl URL has a hostname, the port, and the database name. In our program we construct a String with `hostname` as *`localhost`*, port number as *3306*, and the database name as *bookstore*.

```
String dbURL = "jdbc:mysql://localhost:3306/bookstore";
```

We also assign the username and `password`, this has to be the same username and password that is used for starting the MySQL Server.

```
String username ="root";
String password = "password";
```

Next, we invoke the `getconnection()` method using the URL, `username`, and password:

```
Connection dbCon =DriverManager.getConnection(dbURL,
username, password);
```

Next, we use the Connection object returned by the `getconnection()` method and invoke the `createStatement()` method. This method returns a `Statement` object for sending SQL statements to the database.

```
Statement stmt = dbCon.createStatement();
```

Next, we invoke the `executeQuery()` method of the `Statement` object to execute an SQL query. This method returns a single `ResultSet` object. The `ResultSet` is a table of data returned by a specific SQL statement.

```
String query ="select * from book";
ResultSet rs = stmt.executeQuery(query);
```

Finally, we use the `next()` method of the `ResultSet` object to iterate through all the rows of data returned by the query. When there are no more rows left, the `next()` method will return false. Since we know there are 5 columns in the book table, we use a for loop and the `getString()` method of the `ResultSet` object to print all the five columns.

```
while(rs.next()){
   for (int i = 1; i <=5; i++) {
      System.out.print(rs.getString(i));
      System.out.print("|");
   }
   System.out.println();
}
```

All the statements described above have to be put in a try catch block to catch Exceptions (of the `Exception` type `SQLException`) that can occur while connecting or fetching data from the database.

Figures 3.11(k) and 3.11(l) show the complete listing of the MySQL Demo program and its output respectively.

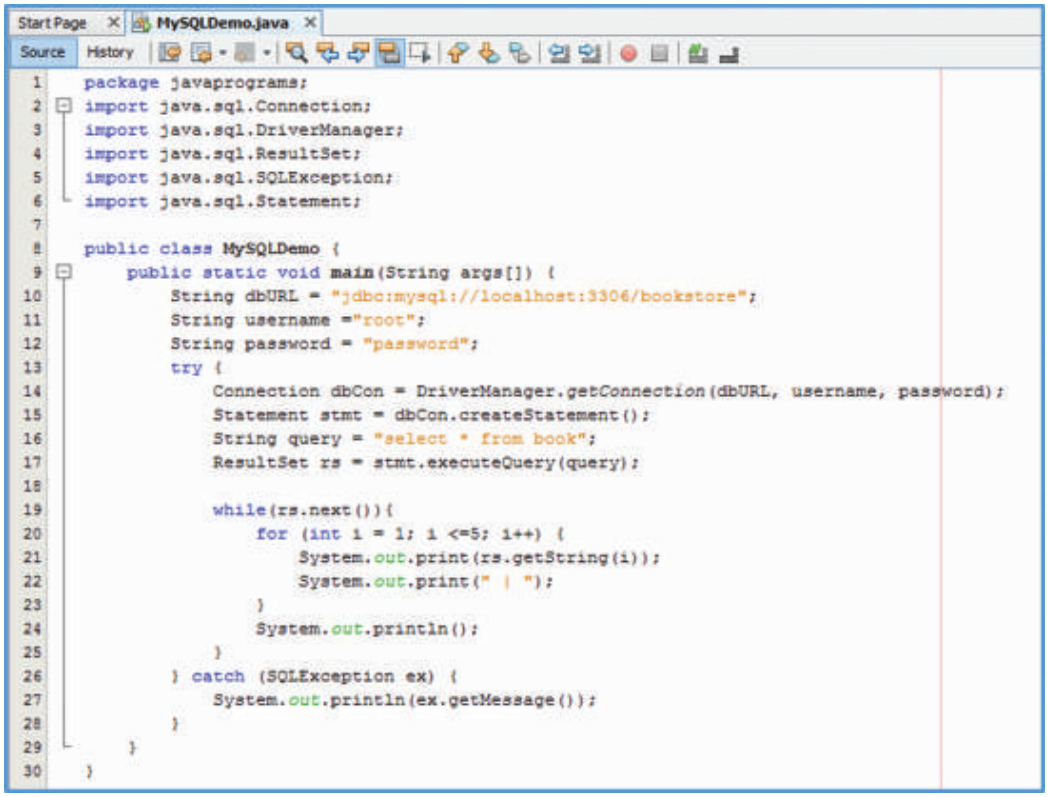


Figure 3.11(k): The MySQL Demo Program

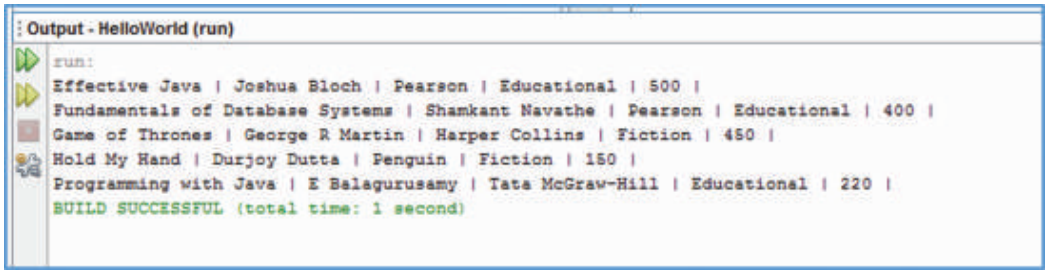


Figure 3.11(l): Output from the MySQL Demo Program

Once you complete running your database program, you can disconnect the MySQL Server from NetBeans. Under the Databases node, right click the **MySQL Server at localhost:3306 [root]** and select **Disconnect.**

3.12 Assertions, Threads, and Wrapper Classes

In this section, we will look at some advanced features of Java.

3.12.1 Assertions

An assertion is a useful mechanism for effectively identifying/detecting and correcting

logical errors in a program. When developing your Java programs, it is good programming practice to use assert statements to debug your code. An assert statement states a condition that should be true at a particular point during the execution of the program.

There are two ways to write an assertion

assert `expression;`

assert `expression1 : expression2`

The first statement evaluates `expression` and throws an AssertionError if `expression` is false. The second statement evaluates `expression1` and throws an AssertionError with `expression2` as the error message if `expression1` is false.

The program fragment below demonstrates usage of the `assert` statement.

```
assert age >= 18:"Age not Valid";
```

When this statement is executed, we assert that the value of the variable age should be >= 18. If it is not, an AssertionError is thrown and the error message "`Age not Valid`" is returned.

Note that, since assertions reduce runtime performance, they are disabled by default. To enable assertions at runtime, you can enable them from the command line by using the –ea option.

```
java –eaAssertionDemo
```

Alternatively, to enable assertions in NetBeans, Right click on your project>Properties> Run>VMOptions. Type–ea in the text box next to VM Options and click OK.

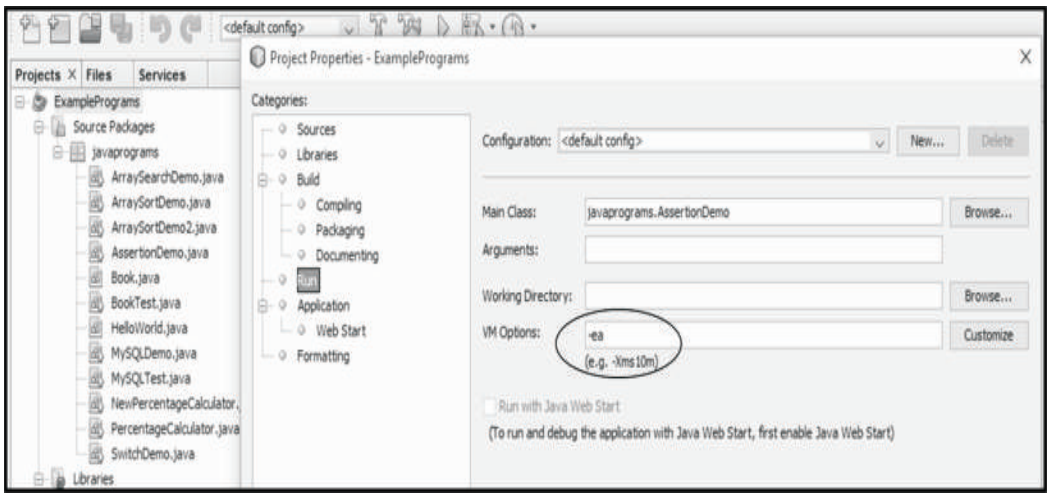


Figure 3.12(a): Enabling Assertions in NetBeans

You can see the complete code listing of the `AssertionDemo` program in Figure3.12(b) and its output in Figure 3.12(c).

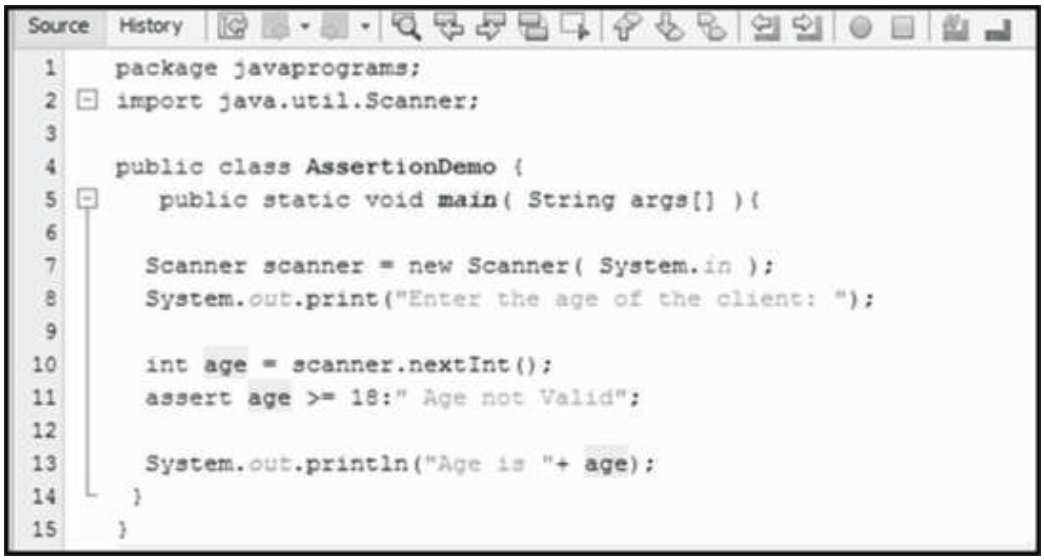


Figure 3.12(b): The AssertionDemo Program

Figure 3.12(c): Output from the AssertionDemo Program

3.12.2 Threads

A multithreaded program is one that can perform multiple tasks concurrently so that there is optimal utilization of the computer's resources. A multithreaded program consists of two or more parts called threads each of which can execute a different task independently at the same time.

In Java, threads can be created in two ways

1. By extending the Thread class
2. By implementing the Runnable interface

The first method to create a thread is to create a class that extends the Thread class from the java.lang package and override the run() method. The run() method is the entry point for every new thread that is instantiated from the class.

```
public class ExtendThread extends Thread {
   public void run() {
      System.out.println("Created a Thread");
```

```
        for (int count = 1; count <= 3; count++)
            System.out.println("Count="+count);
    }
}
```

To create a thread, instantiate it from the ExtendThread class, and to start its execution, call the start() method of the Thread class.

```
public static void main(String args[]) {
ExtendThread t1 =new ExtendThread();
t1.start();
}
```

The program in Figure 3.12(d) demonstrates creation of two threads using the ExtendThread class and Figure 3.12(e) shows its corresponding output. As you can see, the operating system alternates the execution of both the threads, that is, both the threads execute simultaneously.

Start Page | ExtendThread.java

Source | History

```
1  package javaprograms;
2
3  public class ExtendThread extends Thread {
5      public void run() {
6          System.out.println("Created a Thread");
7          for (int count = 1; count <= 3; count++)
8              System.out.println("Count="+count);
9      }
10
11     public static void main(String args[]){
12         ExtendThread t1 =new ExtendThread();
13         ExtendThread t2 =new ExtendThread();
14
15         t1.start();
16         t2.start();
17     }
}
```

Figure 3.12(d): The ExtendThread Demo Program

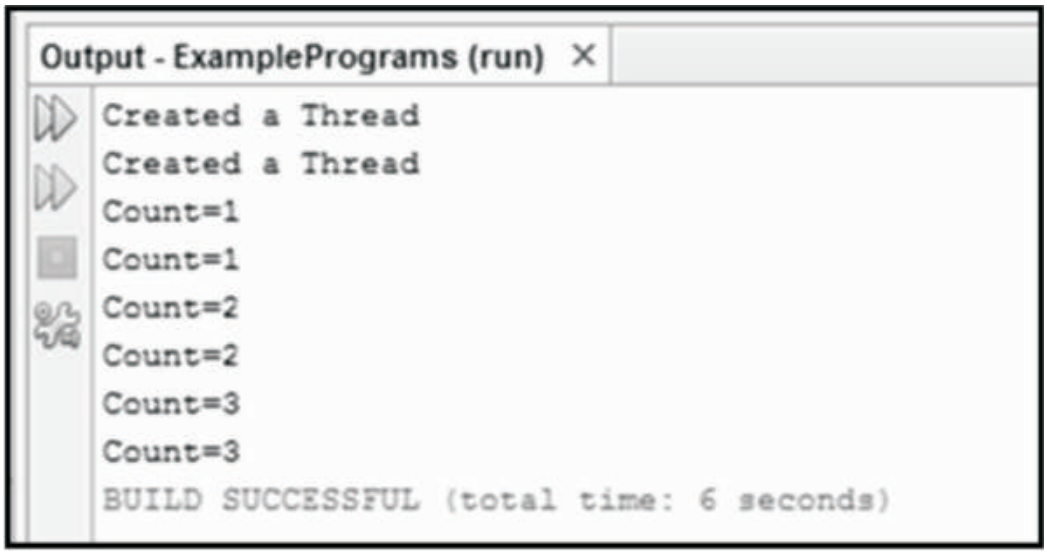


Figure 3.12(e): Output from the ExtendThread Program

Classes that you create by extending the Thread class cannot be extended further. The second method to create a thread is to create a class that implements the Runnable interface and override the run() method. Implementing the Runnable interface gives the flexibility to extend classes created by this method.

```
public class RunnableDemoimplements Runnable {
   public void run() {
      System.out.println("Created a Thread");
      for (int count = 1; count <= 3; count++)
          System.out.println("Count="+count);
   }
}
```

To create a thread, first instantiate the class that implements the Runnable interface, then pass that object to a Thread instance. As before, to start the execution of the thread call the start() method.

```
public static void main(String args[]){
   RunnableDemo r = new RunnableDemo();
Thread t1 = new Thread(r);
   t1.start();
   }
```

The program in Figure 3.12(f) demonstrates creation of two threads using the RunnableDemo class. The output from the program is the same as that in Figure 3.12(e).

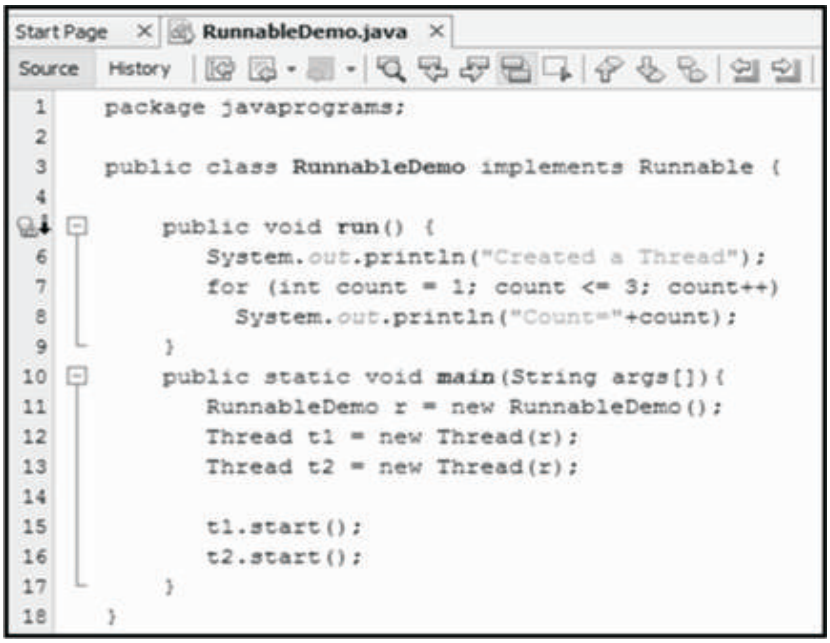


Figure 3.12(f): The RunnableDemo Program

When many threads are running, there is no guarantee of the order in which the threads will be executed. However, if you want some thread to have a higher priority than others, you can change its priority level using the `setPriority()` method of the Thread class. The Priority levels can range from 1 to 10. The `sleep()` method causes the thread to suspend execution by the specified number of milliseconds parameter. A thread can enter a wait state by invoking the `wait()` method. This method is useful when you have multiple threads running but you want one of them to start execution only when another one finishes and notifies the first one to resume execution.

3.12.3 Wrapper Classes

By default, the primitive datatypes (such as `int`, `float`, and so on) of Java are passed by value and not by reference. Sometimes, you may need to pass the primitive datatypes by reference. That is when you can use wrapper classes provided by Java. These classes wrap the primitive datatype into an object of that class. For example, the Integer wrapper class holds an int variable.

Consider the following two declarations:

```
int      a = 50;
Integer  b = new Integer(50);
```

In the first declaration, an int variable is declared and initialized with the value 50. In the second declaration, an object of the class Integer is instantiated and initialized with the value 50. The variable a is a memory location and the variable b is a reference to a memory location that holds an object of the class Integer. Figure 3.12(g) illustrates the difference between an int and an Integer variable.

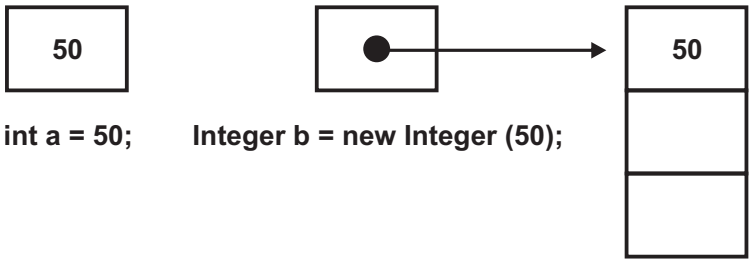


Figure 3.12(g): Memory assignment of Primitive datatype versus Wrapper Class

Access to the value of a wrapper class object can be made through getter functions defined in the class. For example, the `intValue()` member function of the Integer wrapper class allows access to the int value held in it.

```
int c = a + b.intValue();
```

Another useful function defined in the Integer wrapper class lets you convert a string into its integer value. The following statement converts the string "3456" into the integer3456 and stores it in the int variable d.

```
int d = Integer.parseInt("345");
```

Note that the `parseInt` method is a static member of the Integer class and can be accessed using the name of the class, that is, without creating an instance of the class. Similar to the parseInt method, the toString() method allows conversion from an integer value to a String as shown in the statement below.

```
String s = Integer.toString(3456);
```

Similar to the Integer wrapper class for the int datatype, each of the eight primitive types has a wrapper class defined for it (in the `java.lang` package) all of which are imported by default in Java programs. Table 3.12(g) lists the wrapper classes for the corresponding datatype.

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

Table 3.12(g): Wrapper Classes

Exercise:

Q1. What is Java Bytecode?

Q2. Explain the difference between a Class and an Object with an example.

Q3. What is a constructor? Why is it used?

Q4. How are exceptions handled in Java?

Q5. How can threads be created in Java? Briefly explain with an example.

Lab Exercises:

Q1. Write a program in Java to implement the formula.

area = length * width * height

Q2. Write a program in Java to find the result of the following expressions.

(Assume a = 20 and b = 30)

i) a%b

ii) a /= b

iii) (a + b * 100) /10

iv) a && b

v) a++

Q3. Write a program in Java to print the square of every alternate number of an array.

Q4. Write a program in Java to create class Triangle with the data members base, height, area and color. The members base, height, area are of type double and color is of type string. Write getter and setter methods for base, height and color, and write method to compute_area (). Create two object of class Triangle, compute their area, and compare their area and color. If area and color both are same for the objects then display "Matching Triangles" otherwise display "Non matching Triangles".

Q5. Write a program in Java to enable user to handle divide by zero exception.

Unit - 4: Work Integrated Learning IT – DMA

4.1 Introduction

Previous chapters provided you an insight into how database management concepts are helpful for organizing the data in meaningful ways. They facilitate fast retrieval of information as and when required. You learnt data manipulation language SQL for creating, selecting and modifying the data in database. You also learnt regarding various web applications that use databases for managing the data.

In this chapter, you will learn about various work areas that use database management systems. Also, we shall develop a shopping application using database management systems and java. During the course of discussion, we have raised some questions and encourage you to think about their solution and explore related issues.

4.2 Identification of Potential Work Areas

Database management systems have found application in various domains as they provide efficient storage and fast retrieval of data. Following are a few domains where database applications may be used:

(A) Education

- For storing information such as student details, marks and result.
- For storing information about faculty and staff members.
- For storing details about school/college such as infrastructure details, department and offered course details.

(B) Banking

- For storing information about customers. For example,
  - (i) Personal details such as address, age, PAN card, occupation, contact numbers.
  - (ii) Accounts and loans related information.
  - (iii) Information regarding daily transactions.
- For storing employee details such as their personal information, salary, leaves taken, joining date, retirement year.

(C) Hospitals

- For maintaining information regarding patients such as their personal details, health records, hospitalization date.
- For storing information regarding doctors, nurses, staff members, rooms, medical equipment, and medicines.

(D) Government Sector

- For storing details of electoral roll, all types of taxes (Income tax, sales tax, house tax etc.), criminal records.
- For storing details of PAN cards, AADHAR cards, vehicle registration, birth/death certificate registration.

(E) Companies

- For storing information regarding employees such as name, address, contact number, salary, position, joining date.
- For maintaining information regarding the projects handled by them.
- For keeping track of infrastructure, sales, and investments.

(F) E-Commerce

- For storing information regarding products such as price, quantity, quality, manufacturing date, seller.
- For storing information regarding customers such as name, contact number, address and their orders.

(G) Airlines

- For storing information about flight details such as arrival time, departure time, fares, passenger capacity, number of bookings.
- For keeping track of online and offline reservations.

(H) Railways

- For storing information about train details such as arrival time, departure time, fares, passenger capacity, number of bookings.
- For keeping track of online and offline reservations.

(I) Telecommunications

- For storing information regarding communication networks, customers, call records, and their bills.
- For storing information about the plans offered and plans subscribed by the users.

(J) Hotels

- For storing information regarding guests, their check-in and check-out times, duration of stay, room number allocated.
- For keeping track of booked and available rooms.
- For storing information regarding hotel staff, menu items, and infrastructure.

Various applications in above mentioned work areas can be developed to cater their respective needs. For example,

(i) College/school management application.

(ii) Online reservation application for airline, railways, buses, movie and hotels.

(iii) Application for online tutorials and tests.

(iv) Hospital management application.

(v) Company management application.

(vi) Bank management application.

(vii) Shopping application.

(viii) Bill payment and bill generation application.

(ix) Library management application.

(x) Real estate management application.

(xi) Inventory control application.

(xii) Shop management application.

(xiii) Restaurant management application.

(xiv) Telecom management application.

(xv) Insurance management application.

4.3 A Shopping Website - A Case Study

In this section, we shall study database management application used in **XYZ Shopping Website** that allows users to shop online by providing 24 X 7 access. The website offers a wide range of products ranging from books, clothes, stationery, kitchen appliances to electronic items. A customer can select items to be purchased by putting them in shopping cart. The website also allows users to make a choice of the product by performing comparative analysis of the products. The order can be completed by choosing to pay through Cash on Delivery option, credit/debit card, or net banking. Once the payment is made, goods will be delivered at the address specified by the customer Figure 4.3(a) shows the layout of the website.

Figure 4.3(a): XYZ Shopping Website

Left panel of the website shows various categories of products. In the center, there are pictures of some of its products. Right panel shows details of offers, link to contact, login, and track order.

4.3.1 Entities Involved

Now, let us have a look at types of information required to be maintained in the database for creating the shopping application. We have identified below some of the entities that would be useful for such an application. For each entity, a separate table is to be created using **create** SQL command. The primary key is underlined in each case.

(i) CATEGORY: This table stores categories of products available. It will store information such as category id, name of category, and its description.

Schema: CATEGORY(Category_id, Name, Description)

Name	Type	Remarks
Category_id	INT(10)	Category number (Primary Key)
Name	VARCHAR(20)	Name of category
Description	VARCHAR(40)	Description of category

Figure 4.3(b): Category Table

(ii) PRODUCT: This table stores details of products available for shopping. It will store information such as product_id, name, category_id, price, quantity, discount, brand, color, size, seller_id and its description.

Schema: PRODUCT (Product_id, Name, Category_id, Price, Quantity, Discount, Brand, Color, Size, Seller_id, Description)

Name	Type	Remarks
Product_id	INT(10)	Product number (Primary Key)
Name	VARCHAR(20)	Name of the product
Category_id	INT(10)	Category to which product belongs (Foreign Key)
Price	INT(10)	Price of product
Quantity	INT(10)	Number of items left in stock

Discount	INT(3)	Discount available for the product
Brand	VARCHAR(20)	Brand to which product belongs
Color	VARCHAR(20)	Color of the product
Size	VARCHAR(20)	Size of the product
Seller_id	INT(10)	Seller of the product (Foreign Key)
Description	VARCHAR(20)	Description of product

Figure 4.3(c): Product Table

(iii) CUSTOMER: This table stores customer details specified by him either at login time or during order. It will store information such as customer_id, password, first name, last_name, address, and email_id and contact_num.

Schema: CUSTOMER (Customer_id, Password, First_name, Last_name, Address, Email_id, Contact_num)

Name	Type	Remarks
Customer_id	INT(10)	Customer number (Primary Key)
Password	VARCHAR(30)	Login password of customer
First_name	VARCHAR(20)	First name of customer
Last_name	VARCHAR(20)	Last name of customer
Address	VARCHAR(50)	Address of customer
Email_id	VARCHAR(20)	Email id of customer
Contact_num	INT(10)	Contact number of customer

Figure 4.3(d): Customer Table

(iv) SELLER: This table stores details about seller of the products. It will store information such as seller_id, and rating of seller.

Schema: SELLER (Seller_id, Rating)

Name	Type	Remarks
Seller_id	INT(10)	Seller number (Primary Key)
Rating	INT(1)	Rating of seller

Figure 4.3(e): Seller Table

(v) WISH_LIST: This table stores details about product, customer wishes to buy in future. It will store information such as customer_id and product_ids of products in his wish list. Note, the primary key of this table is a combination of both its attributes the cutomer_id and product_id.

Schema: WISH_LIST (Customer_id, Product_id)

Name	Type	Remarks
Customer_id	INT(10)	Customer number (Primary Key and Foreign Key)
Product_id	INT(10)	Product number (Primary Key and Foreign Key)

Figure 4.3(f): Wish List Table

(vi) ORDER: This table stores details of all the orders placed till date. It will store information such as order id, customer_id of user who gave the order, product_id, shipment_id, order_date, and current_status.

Schema: ORDER (Order_id, Customer_id, Product_id, Shipment_id, Order_date, Current_status)

Name	Type	Remarks
Order_id	INT(10)	Order number (Primary Key)
Customer_id	INT(10)	Customer who gave the order (Foreign Key)
Product_id	INT(50)	Product ids of products ordered (Maximum 5) (Foreign Key)
Shipment_id	INT(10)	Shipment number (Foreign Key)
Order_date	DATE	Date of order
Current_status	VARCHAR(20)	Pending or Delivered

Figure 4.3(g): Order Table

(vii) SHIPMENT: This table stores details of shipment used for dispatching a product. It will store information such as shipment_id, details and shipment_date.

Schema: SHIPMENT(Shipment_id, Details, Shipment_date)

Name	Type	Remarks
Shipment_id	INT(10)	Shipment number (Primary Key)
Details	VARCHAR(30)	Shipment details such as name, location from which order is dispatched
Shipment_date	DATE	Date of shipment

Figure 4.3(h): Shipment Table

(viii) PAYMENT: This table stores details of payment made for an order. It will store information such as payment_id, order_id, payment_amount, payment_date, and mode of payment.

Schema: PAYMENT (Payment_id, Order_id, Payment_amount, Payment_date, Mode)

Name	Type	Remarks
Payment_id	INT(10)	Payment number (Primary Key)
Order_id	INT(10)	Order number for which payment is made (Foreign Key)
Payment_amount	INT(10)	Amount of payment
Payment_date	DATE	Date of payment
Mode	VARCHAR(20)	Mode of payment such as debit card, credit card, net banking, cash on delivery, E-gift voucher.

Figure 4.3(i): Payment Table

Now, let us examine how these entities are linked with each other. Figure 4.3(j) shows relationships between these entities, in an informal manner.

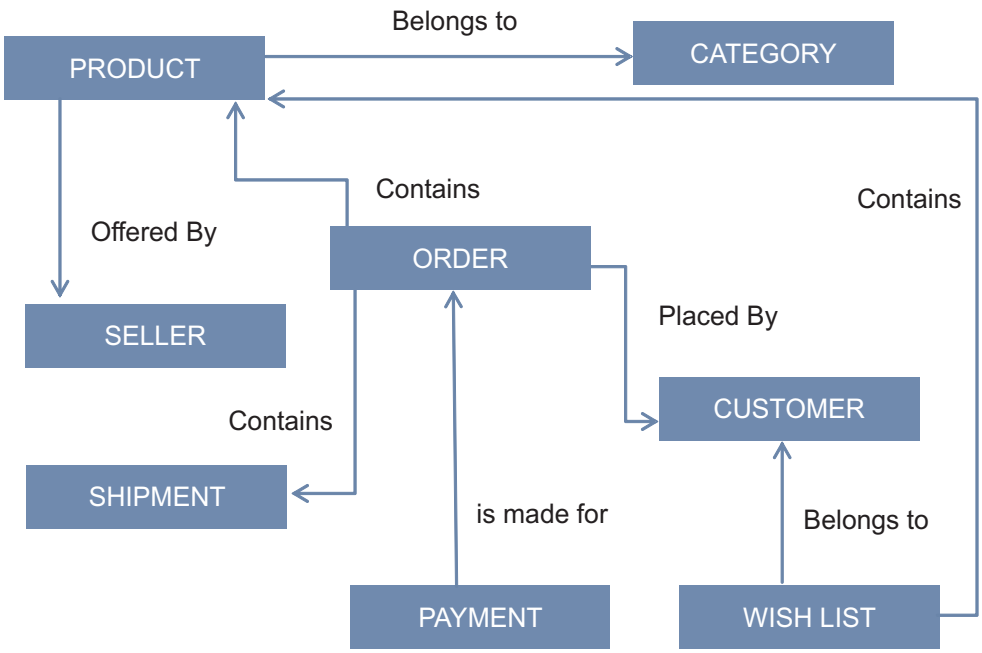


Figure 4.3(j): Relationship Between Entities

4.3.2 Functionality

Now, let us have a look at how shopping application may use and manage database. Given below are some important functions that are performed by XYZ Shopping website:

1. First, all the categories whose products are to be made available for shopping are decided by XYZ Company. For storing the finalized categories in the CATEGORY

table, function **insert_category** of class **Category** defined in Figure 4.3(k) may be used. At later point in time, if the company wishes to come up with more categories, it may do so by adding them in table using the same function. Also, the company may decide to either delete or modify the categories using functions **delete_category** and **modify_category** respectively. If the customer selects any category on the website, function **display_category** is invoked to display all the products in that category. The skeleton of class and the related functions for managing the CATEGORY table are given in Figure 4.3(k).

```
package shopping_application;
public class Category {
      public void insert_category(String categ_tuple[])
      {
            // Add functionality to insert a row of category
            // named categ_tuple.
            // SQL command to be used: insert
      }
      public void delete_category(int categ_id)
      {
            // Add functionality to delete a category with id categ_id.
            // SQL command to be used: delete
      }
      public void modify_category(int categ_id, String attr, String
      new)
      {
            // Add functionality to change value of attribute attr
            // of category with id categ_id to new.
            // SQL command to be used: update
      }
      public void display_category(int categ_id)
      {
            // Add functionality to display all the products in
            // category with id categ_id.
            // SQL command to be used: select
      }
}
```

Figure 4.3(k): Class for Table Category

QUESTION

Suppose that numbers of products belonging to a category are quite large. Customer may not be interested in some products at all. What do you think, should be the order of display of products? Should it be based on popularity, price or any other criteria? For each criterion, figure out what else is required to be stored in database.

2. Next, the list of products along with associated details are to be decided for each category and inserted in PRODUCT table using **insert_product** function of class **Product** as shown in Figure 4.3(l).Similarly, company at later point in time may add more products by calling the same function. Functions **delete_product** and

modify_product may respectively be used to delete or modify any product, as and when required. If the customer selects any product on the website, then function **display_product** may be used to display the details of product. The skeleton of class and the related functions for managing the PRODUCT table are given in Figure 4.3(l).

```
package shopping_application;
public class Product {
      public void insert_product(String prod_tuple[])
      {
            // Add functionality to insert a row of product named
            // prod_tuple.
            // SQL command to be used: insert
      }
      public void delete_product(int prod_id)
      {
            // Add functionality to delete a product with id prod_id.
            // SQL command to be used: delete
      }
      public void modify_product(int prod_id, String attr, String new)
      {
            // Add functionality to change value of attribute attr
            // of product with id prod_id to new.
            // SQL command to be used: update
      }
      public void display_product(int prod_id)
      {
            // Add functionality to display details of the products
            // with id prod_id
            // SQL command to be used: select
      }
}
```

Figure 4.3(l): Class for Table Product

QUESTION

Identify at least two other functionalities that should be supported by class Product and write routines for them.

3. The products inserted above contain **seller_id** of the Seller offering the product for selling. So, we also need to specify information regarding all the sellers in the table SELLER. This may be achieved using function **insert_seller** of class **Seller** as shown in Figure 4.3(m). Further, functions **delete_seller** and **modify_seller** may be used to delete or modify any seller's information, as and when required. If the customer selects any product on the website, then function **display_seller** may be used to display the details of seller offering the product. Suppose same product is offered by more than one seller. In such a case, you need to list all the sellers along

with their details. Here, declaring **Product_id** and **Seller_id** as the composite key will serve the purpose. The skeleton of class and the related functions for managing the SELLER table are given in Figure 4.3(m).

```
package shopping_application;
public class Seller {
      public void insert_seller(String seller_tuple[])
      {
            // Add functionality to insert a row of seller named
            // seller_tuple.
            // SQL command to be used: insert
      }
      public void delete_seller(int seller_id)
      {
            // Add functionality to delete a seller with id seller_id.
            // SQL command to be used: delete
      }
      public void modify_seller(int seller_id, String attr, String new)
      {
            // Add functionality to change value of attribute attr
            // of seller with id seller_id to new.
            // SQL command to be used: update
      }
      public void display_seller(int seller_id)
      {
            // Add functionality to display details of the seller
            // with id seller_id
            // SQL command to be used: select
      }
}
```

Figure 4.3(m): Class for Table Seller

QUESTION

Suppose a customer want to provide rating to a seller. Is there any need of special routine for that? Specify how can you achieve this.

4. Whenever a customer visits the shopping website, he can either sign in or can specify his details while placing the order. If he is the existing user, he may login using interface given in Figure 4.3(n). If the customer is not yet registered, and wants to get registered, he may use the interface provided in Figure 4.3(o). It may be noted that almost same interface (excluding password field) is used when unregistered user specifies his details during placement of order. The details specified by user are saved into CUSTOMER table using function **insert_customer** of class **Customer**. These details are used whenever customer makes any order. For example, shipping address specified can be used for delivery purpose. At any point in time, customer may change his personal details. Function **modify_customer** may be used for the same.

Figure 4.3(n): Login Interface

Figure 4.3(o): Sign Up Interface

The skeleton of class and the related functions for managing the CUSTOMER table is given in Figure 4.3(p).

```
package shopping_application;
public class Customer {
      public void insert_customer(String customer_tuple[])
      {
            // Add functionality to insert a row of customer named
            // customer_tuple.
            // SQL command to be used: insert
      }
      public void modify_customer(int customer_id, String attr, String
      new)
      {
            // Add functionality to change value of attribute attr
            // of customer with id customer_id to new.
            // SQL command to be used: update
      }
}
```

Figure 4.3(p): Class for Table Customer

QUESTION

Suppose a customer wants to delete his account. What kind of functionality needs to be added to the class Customer? Specify the routine for it.

QUESTION

Figure out what extra information regarding customer can be added to the database CUSTOMER. Specify how this information can be used. Is it possible to write any more functions associated with the newly added information?

5. While browsing products on the website, customers may wish to buy certain products at a later point in time. At this point, he may add certain products to the wish list. For achieving this, function **insert_wish** of class **WishList** may be invoked. Also, customer may want to delete certain products from the wish list. This may be achieved using function **delete_wish**. At later point in time, customer may view his wish list using function **display_wish**. The skeleton of class and the related functions for managing the WISH LIST table are given in Figure 4.3(q).

```
package shopping_application;
public class WishList{
      public void insert_wish(String wish_tuple[])
      {
            // Add functionality to insert a row of wish named
            // wish_tuple.
            // SQL command to be used: insert
      }
      public void delete_wish(int customer_id, String wish)
      {
            // Add functionality to delete a wish of customer with id
            // customer_id.
            // SQL command to be used: delete
      }

      public void display_wish(int customer_id)
      {
            // Add functionality to display wish list of customer with
            // id customer_id.
            // SQL command to be used: select
      }
}
```

Figure 4.3(q): Class for Table Wish List

6. When a customer adds products to be purchased in the shopping cart. He has to go through series of events as described in Figure 4.3(r). In case the customer is not already logged in, he is asked to either login or provide his details. After that, he may choose to pay through any of the available payment modes as specified in Figure 4.3(s). All the details of payment are stored in table PAYMENT using function **insert_payment** of class **Payment**. In case, customer choses to pay cash on

delivery, attribute **Payment_date** is set to the date on which product is delivered. The skeleton of class and the related functions for managing the PAYMENT table are given in Figure 4.3(s).

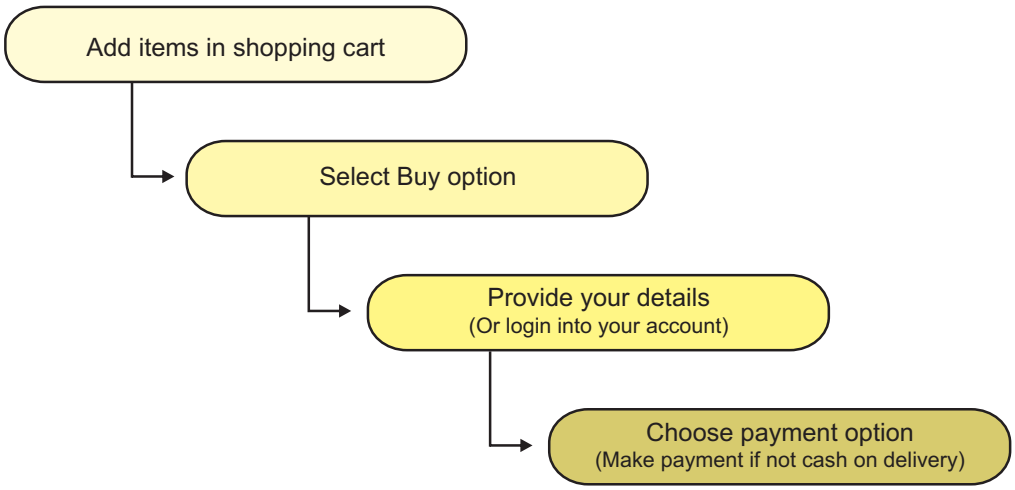


Figure 4.3(r): Sequences of events for Order Completion

```
package shopping_application;
public class Payment{
      public void insert_payment(String payment_tuple[])
      {
            // Add functionality to insert a row of payment named
            // payment_tuple.
            // SQL command to be used: insert
      }
}
```

Figure 4.3(s): Class for Table Payment

QUESTION

Suppose if the customer cancels the order, payment should be refunded to him. For achieving this, is there any need to make changes in PAYMENT table. If yes, specify those changes. Also, write a function

7. All the details of order are stored in table ORDER using function **insert_order** of class **Order** as shown in Figure 4.3(t). If user wishes to cancel order, he can do so by choosing cancel order option. In such a scenario, function **delete_order** will be invoked. If user wants to track his order, he can do so by providing the **order_id**. After doing so, function **display_order** will display all the relevant details of his order along with current status. The skeleton of class and the related functions for managing the ORDER table are defined in Figure 4.3(t).

```
package shopping_application;
public class Order{
      public void insert_order(String order_tuple[])
      {
            // Add functionality to insert a row of order named
            // order_tuple.
            // SQL command to be used: insert
      }
      public void delete_order(int order_id)
      {
            // Add functionality to delete order with id order_id.
            // SQL command to be used: delete
      }
      public void display_order(int order_id)
      {
            // Add functionality to display details of the order with
            // id order_id.
            // SQL command to be used: select
      }
}
```

Figure 4.3(t): Class for Table ORDER

QUESTION

Suppose, number of products specified by user while making order, exceeds five. How will you take care of this situation in function insert_order of class Order?

8. Finally, details of shipment of order are stored in table SHIPMENT using function **insert_shipment_details** of class **Shipment**. The skeleton of class and the related functions for managing the SHIPMENT table are defined in Figure 4.3(u).

```
package shopping_application;
public class Shipment{
      public void insert_shipment_details(String order_shipment[])
      {
            // Add functionality to insert a row of shipment named
            // order_shipment.
            // SQL command to be used: insert
      }
}
```

Figure 4.3(u): Class for Table SHIPMENT

QUESTION

Is there any need of storing shipment details in another table? Can table ORDER be used for storing the same? What if this table is deleted from the database? Can the website still function without any flaw?

Exercise:

Q1. Choose at least one more application out of the ones that are listed at the beginning of chapter. Identify the data requirements and functionality to store and manage the data. Write commands for creating the table for each identified entity. Also, describe the relationship amongst these entities. Write your own java routines for the functionality you have identified.

Appendix - A

Installing and Staring NetBeans IDE

To install NetBeans IDE, first you have to download it and then follow the instructions given below to install it.

Step 1: Visit **https://netbeans.org/downloads/** in your browser.

Step 2: Click to select the appropriate NetBeans bundle to download (Figure 1).

Figure 1: Select the Appropriate NetBeans Bundle to Download

Step 3: Once you click on the Download button the screen as per Figure 2 will appear and the download will start automatically.

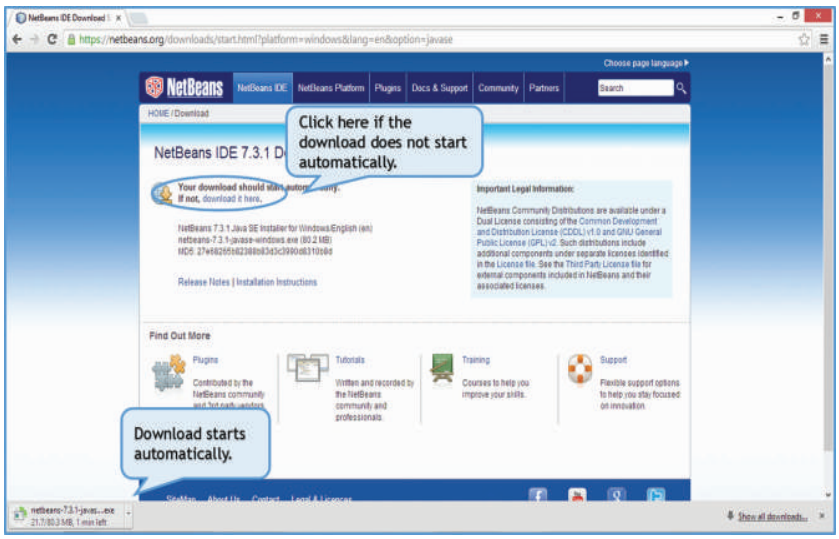


Figure 2: Download Starts Automatically

Step 4: Once the download completes, use the File Explorer to navigate to the file that was just downloaded. Double click on the file name to start the installation process (Figure 3).

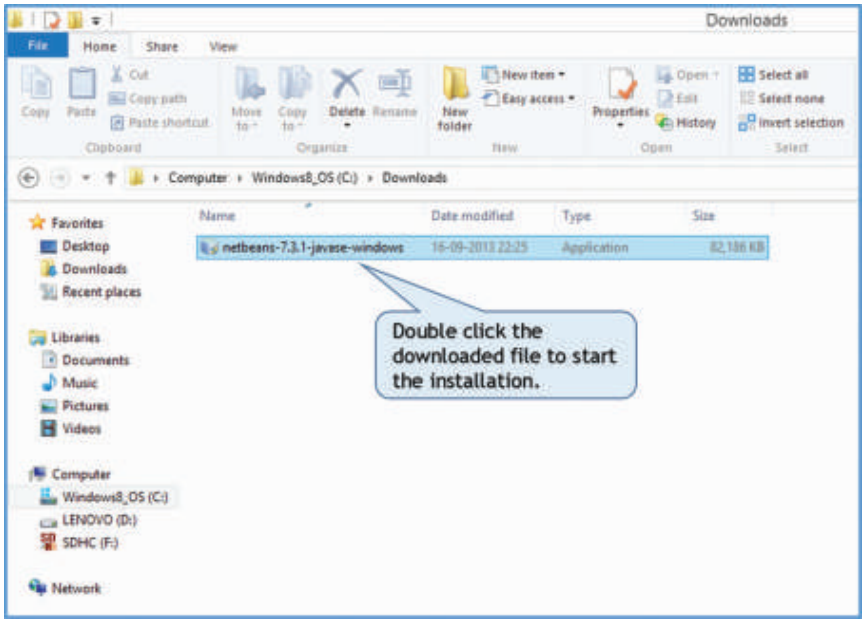


Figure 3: Double click on the file name to start the installation process

Step 5: You might see a User Account Control Window asking whether you want the program to make changes to your computer. Click on the "Yes" button. Once the installation begins, you will see the NetBeans IDE Installer window as in Figure 4.

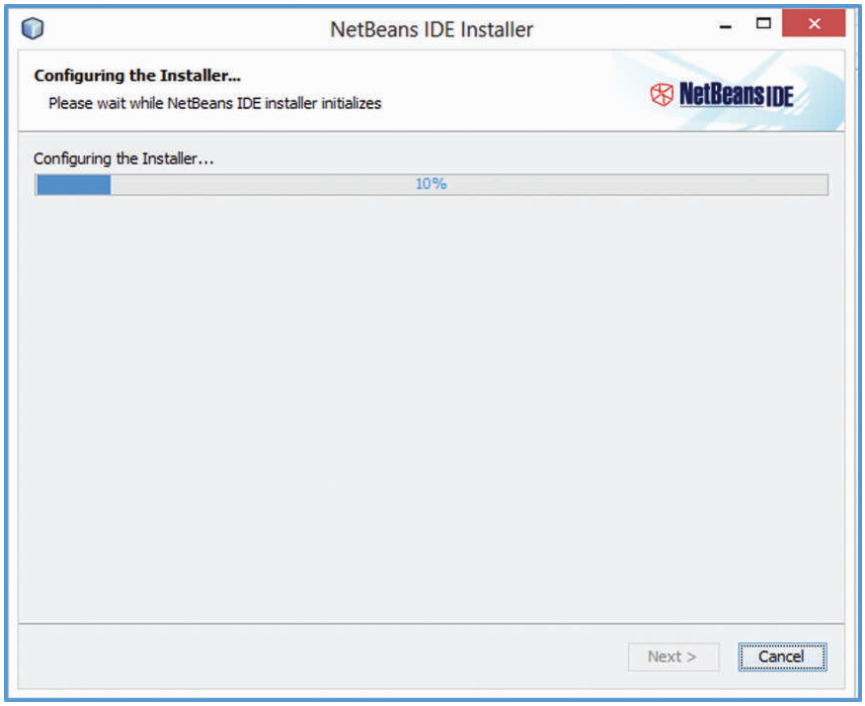


Figure 4: NetBeans IDE Installer

Step 6: Once the NetBeans IDE Installer has been configured, the Welcome screen appears, Click on Next (Figure 5).

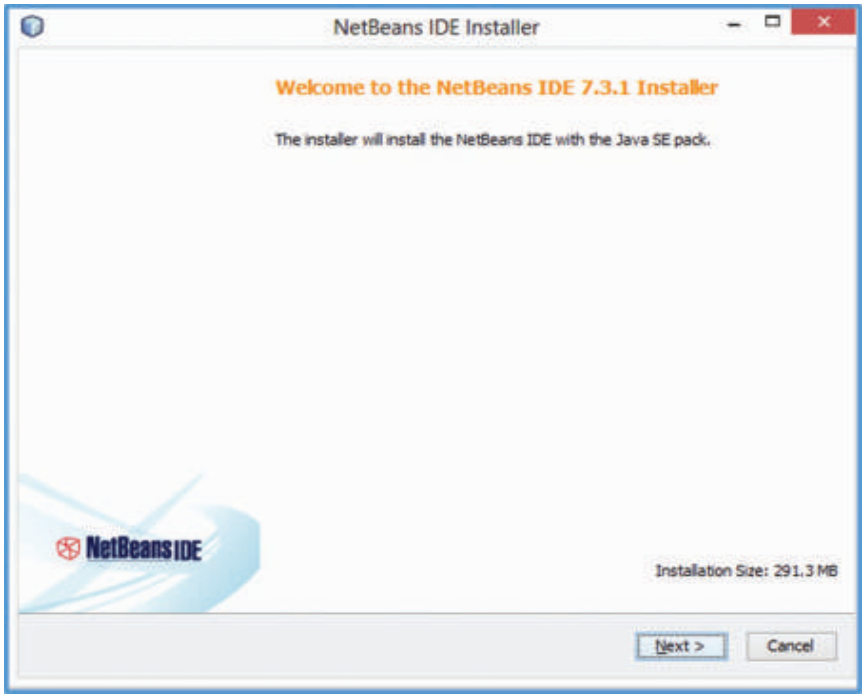


Figure 5: NetBeans IDE Installer Welcome Screen

Step 7: Check the selection box to Accept the License Agreement and click on Next (Figure 6).

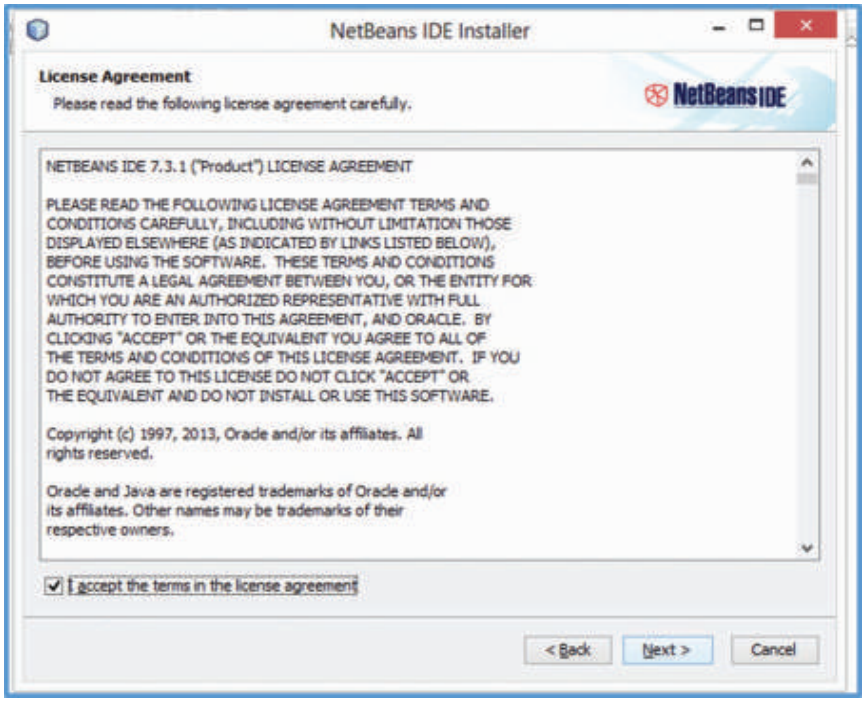


Figure 6: Accept the License Agreement

Step 8: In the JUnit License Agreement screen that appears, Click on Next (Figure 7).

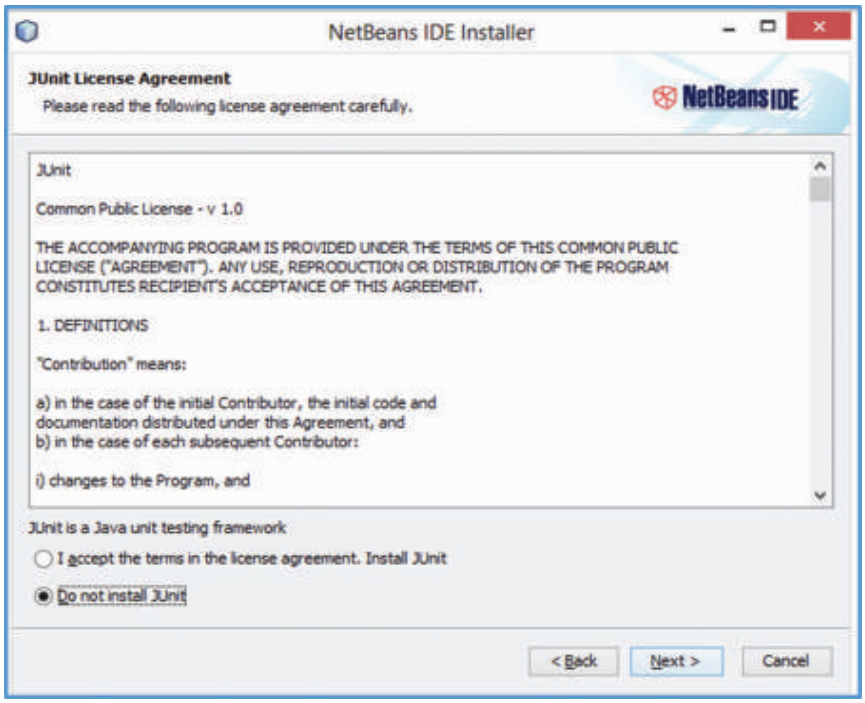


Figure 7: JUnit License Agreement

Step 9: Choose the Installation folder and Click on Next (Figure 8).

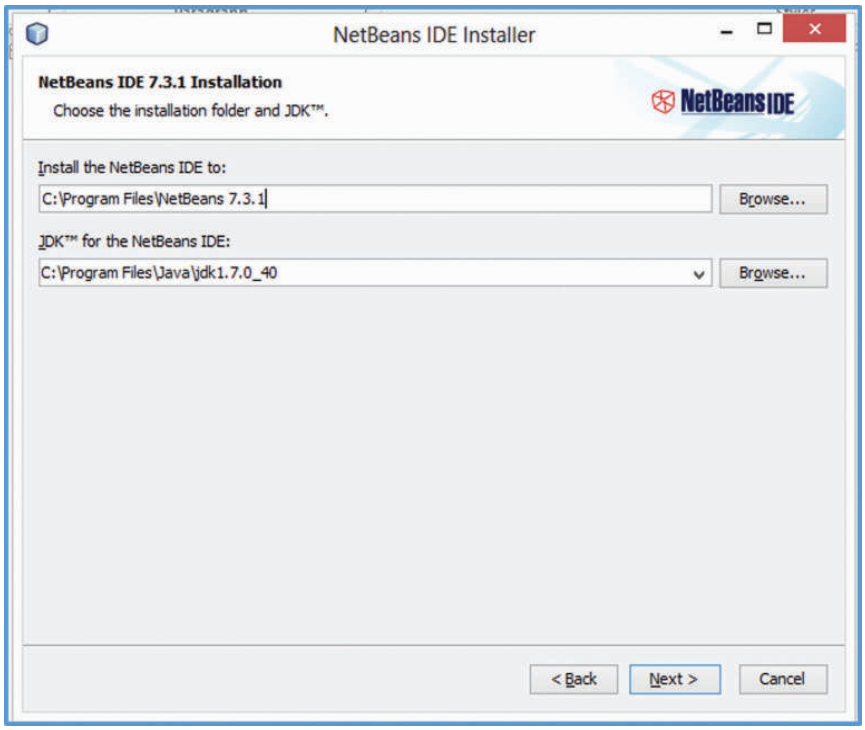


Figure 8: Choose the Installation folder

Step 10: Click the Install button to begin the Installation (Figure 9). The installation will begin as shown in Figure 10.

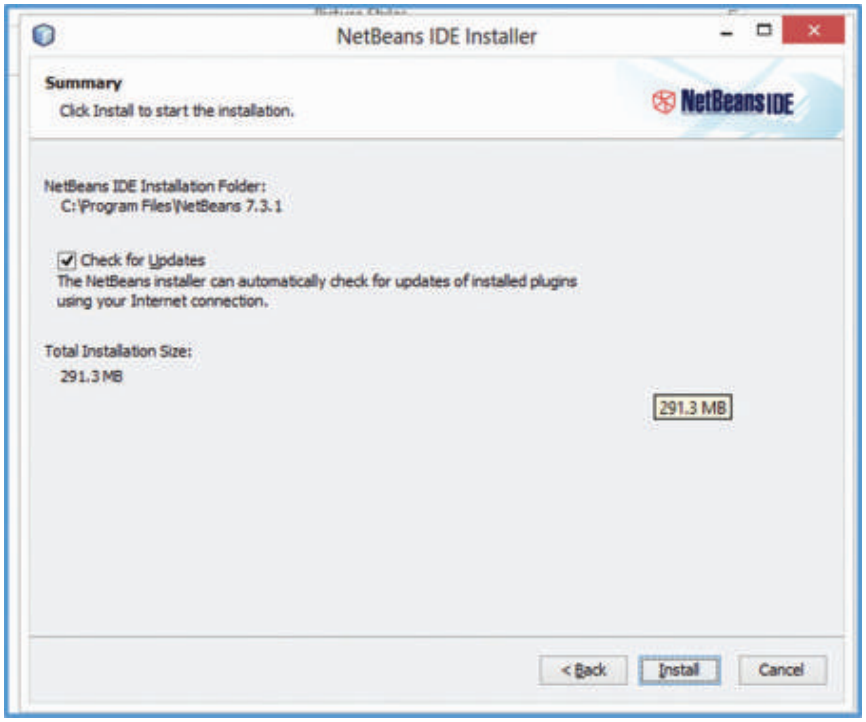


Figure 9: Begin the Installation

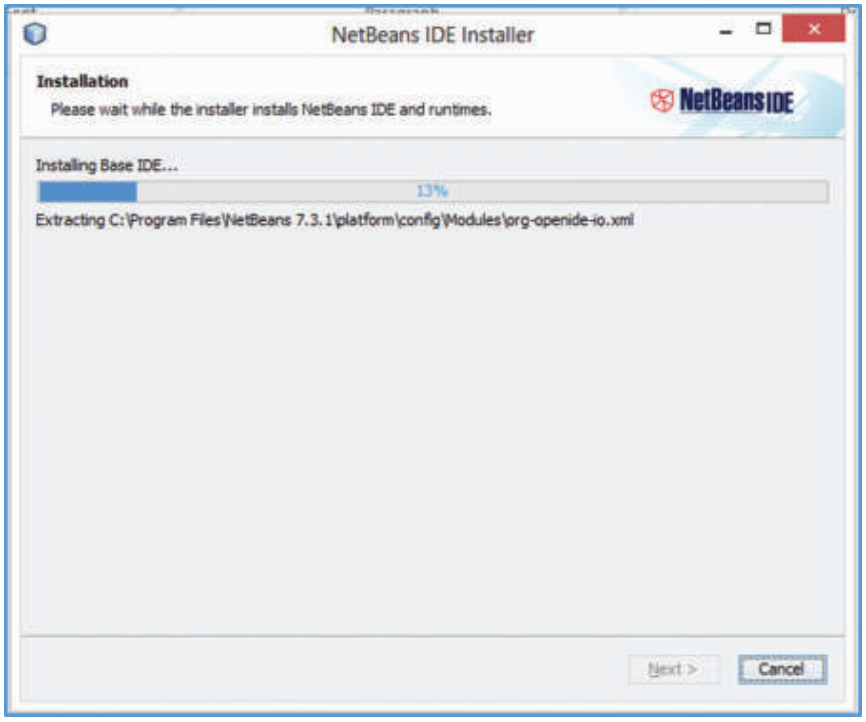


Figure 10: Installing Beans IDE

Step 11: Click on Finish to complete the Installation Process (Figure 11).

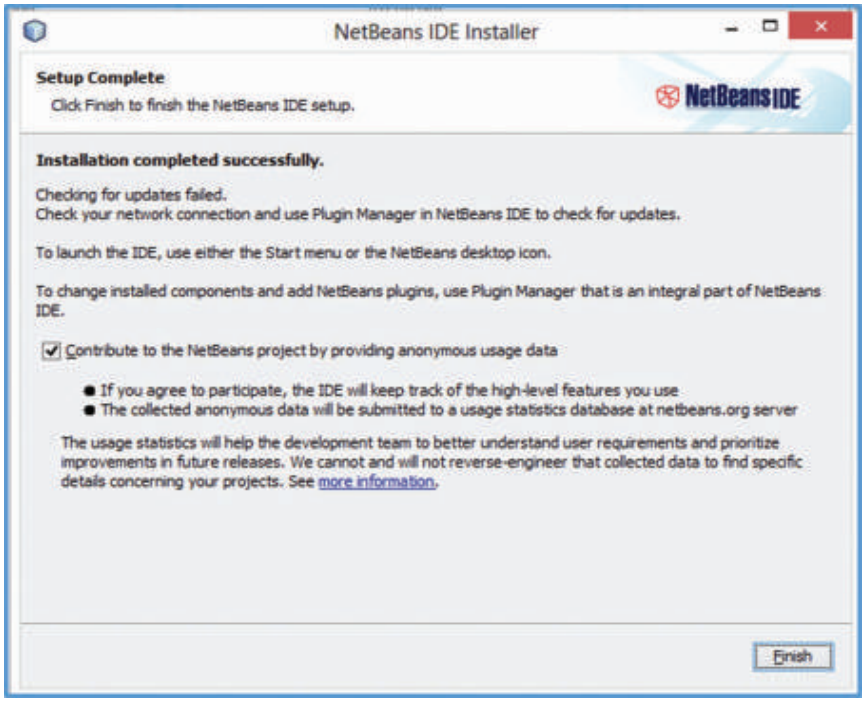


Figure 11: Finish to complete the Installation Process

Step 12: An icon to start NetBeans will be installed on your Desktop (Figure 12). Double click the icon to start the Java NetBeans IDE.

Figure 12: Double click the icon to start the Java NetBeans IDE