INTRODUCTION TO JAVA

Student Guide

NIIT

Introduction to Java

Student Guide

Trademark Acknowledgements

All products are registered trademarks of their respective organizations.
All software is used for educational purposes only.

Introduction to Java SG/18-M03-V1.0

COURSE DESIGN - STUDENT GUIDE

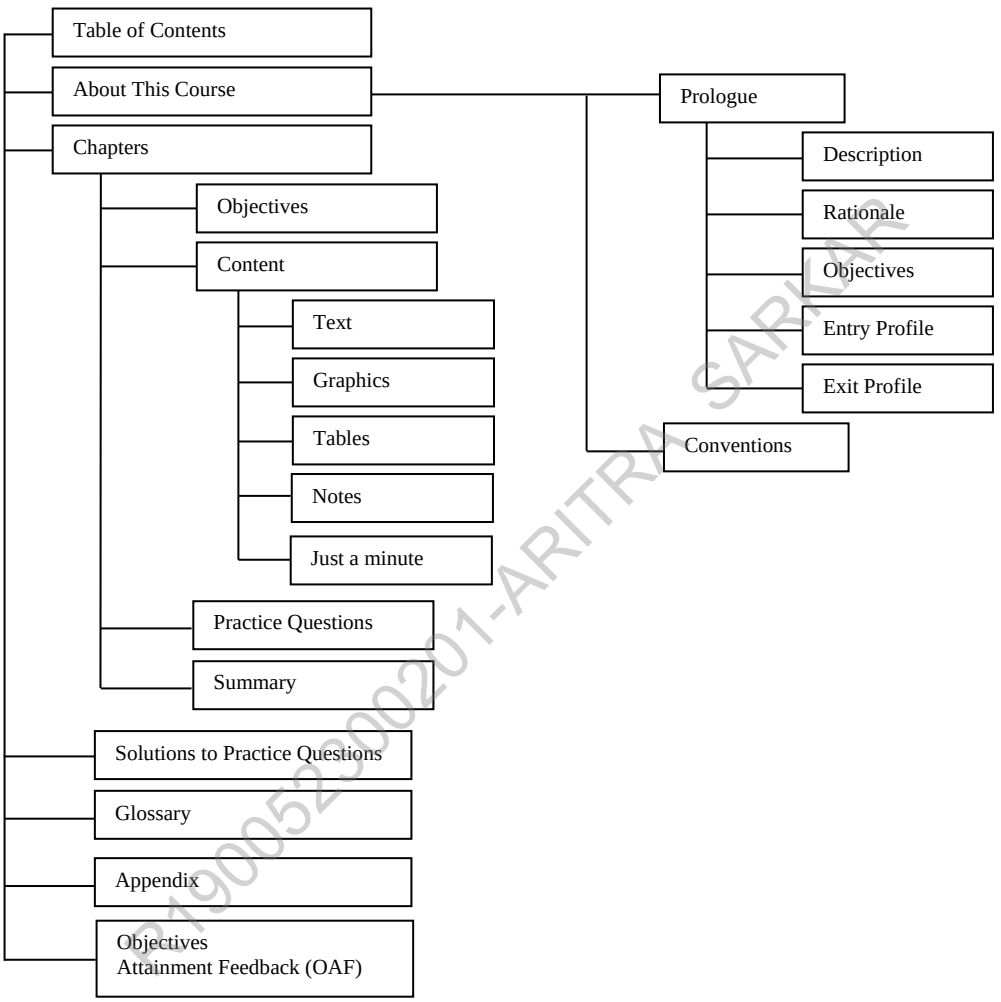


Table of Contents

About This Course

Chapter 1 – Overview of Java

Chapter 2 – Implementing Operators

Chapter 3 – Working with Conditional and Loop Constructs

Chapter 4 – Working with Arrays, Enums, and Strings

Chapter 5 – Implementing Inheritance and Polymorphism

Chapter 6 – Handling Errors

Chapter 7 – Working with Regular Expressions

Chapter 8 – Working with Streams

Solutions to Practice Questions

Glossary

Appendix

About This Course

Prologue

Description

The Introduction to Java course helps the students to get familiar with the Java programming language. It discusses the classes, objects, strings, and arrays. In addition, the course discusses how to create graphical user interfaces. Further, it discusses how to implement inheritance, polymorphism, error handling, and event handling.

The course explains how to create a Java application. It focuses on enabling the students to create both, CUI-based application and GUI-based application, by using Java. It provides details on declaring variables and literals and using arrays. In addition, the course discusses how to use conditional and looping statements. Further, the course describes how to create nested classes, override methods, and create interfaces and packages in Java.

Rationale

Today, there are varied electronic devices available in the market. To work with these electronic devices, different applications are used. These applications are developed by using different programming languages, such as C, C++, Java, and C#. However, the applications developed by using programming languages like C and C++ do not support cross-platform portability.

Java is an object oriented programming language that helps to develop real-life portable applications. We can create both, CUI-based application and GUI-based application, by using Java. The code reusability feature of Java enables software developers to upgrade the existing applications without rewriting the entire code of the application.

Objectives

After completing this course, the students will be able to:

- Get familiar with Java
- Implement operators
- Work with conditional and loop constructs
- Work with arrays, enums, and strings
- Implement inheritance and polymorphism
- Handle errors
- Work with regular expressions
- Work with streams

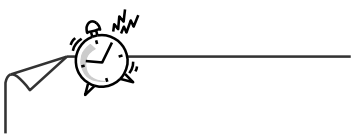
Entry Profile

The students who want to take this course should have basic knowledge of logic building and effective problem solving.

Exit Profile

After completing this course, the students will be able to develop object-based applications in Java.

Conventions

Convention	*Indicates…*
	*Note*
	*Just a minute*
	*Placeholder for an activity*

NIIT

Overview of Java

CHAPTER 1

The principles of object-oriented programming, such as modularity and reusability, are quite popular as they help in reducing the complexity of programs. In order to implement such principles in real world applications, such as library management and inventory management, you need to make use of an object-oriented language, such as Java.

This chapter focuses on the fundamentals and building blocks of the Java programming language. In addition, it explains how to access the class members.

Objectives

In this chapter, you will learn to:

- Get familiar with Java
- Identify the building blocks of a Java program
- Access class members

Introducing Java

Sam works as a programmer at DailyGames Incorporation. He has been assigned the responsibility of creating a jumbled word game named Classic Jumble Word. At the start of this game, the player should be presented with a menu that provides various options, such as playing the game or reading game instructions. When the game starts, the player should be presented with a jumbled-up word from a list of words that is provided in the application. Then, the player needs to identify the correct word corresponding to the given word. As per the requirements, the game must be compatible across various operating systems, such as Windows and Linux. Moreover, the code of the game should be efficiently written to ensure minimum complexity.

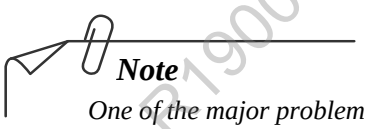
To cater to the preceding requirements, Sam uses the Java programming language. The creator of Java is the Sun Microsystems' chief programmer, James Gosling. In 1991, he and his team started a project to develop software for controlling and programming consumer electronic devices, such as televisions, by using a device, a handheld remote control. Gosling liked the basic syntax and object-oriented features of C++. Therefore, he framed his new language on the same lines as that of C++. The result was a new programming language called Oak, later renamed as Java, which could run on heterogeneous platforms.

Due to its powerful features and underlying architecture, Java is used widely as a programming language to create efficient console or Graphical User Interface (GUI) based applications that can run on a single computer or on a network of computers.

Identifying the Features of Java

Java provides powerful features that make it a popular and widely used programming language. Some of these features are:

- **Simple**: When an object of a class is created, memory needs to be allocated. Later on, when the object is no longer required, the allocated memory to the object should be released. In Java, memory allocation and deallocation happens automatically. The process of automatically deallocating memory is called *garbage collection*.

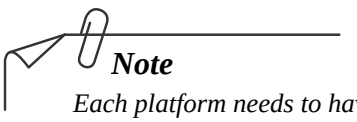


Note

One of the major problem areas in most of object-oriented languages, such as C++, is to handle memory allocation. Programmers need to handle memory in the program explicitly for optimum utilization. To handle memory allocation, they use pointers that enable a program to refer to a memory location of the computer. However, Java does not support pointers and consists of the built-in functionality to manage memory.

- **Object-oriented**: Java is an object-oriented language and it incorporates the various characteristics of an object-oriented language, such as inheritance and polymorphism. In Java, the entire program code must be encapsulated inside a class. Even the most basic program in Java must be written within a class. Furthermore, Java uses objects to refer to real world entities, such as a game or a game player..

- **Platform independence**: In Java, when you compile an error-free code, the compiler converts the program to a platform independent code, known as the *bytecode*. Thereafter, the *Java Virtual Machine (JVM)* interprets this bytecode into the machine code that can be run on that machine. Converting a Java program into bytecode makes a Java program platform independent because any computer installed with the JVM can interpret and run the bytecode.



Note

Each platform needs to have its own JVM to interpret and run bytecode for that platform.

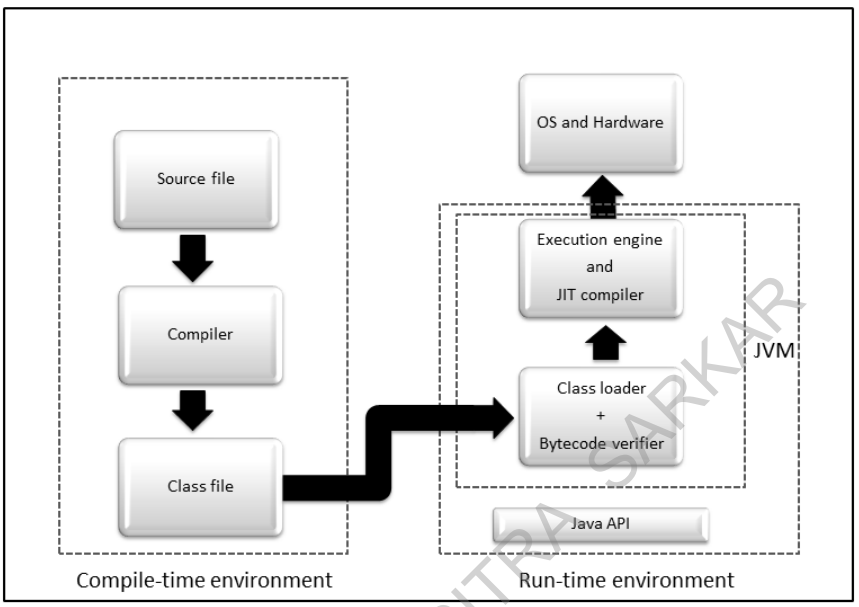
- **Portable**: Portability refers to the ability of a program to run on any platform without changing the source code of the program. The programs developed on one computer can run on another computer, which might have a different operating system. Java enables the creation of cross-platform programs by converting the programs into Java bytecode.
- **Distributed**: Java can be used for the development of those applications that can be distributed on multiple machines in a network, such as the Internet. These applications can be used by multiple users from multiple machines in a network. For this purpose, Java supports the various Internet protocols, such as Transmission Control Protocol and Internet Protocol (TCP/IP) and Hyper Text Transfer Protocol (HTTP).
- **Secure**: Java has built-in security features that ensure that a Java program gains access to only those resources of a computer that are essential for its execution. This ensures that a Java program executing on a particular machine does not harm it.
- **Robust**: Java provides various features, such as memory management and exception handling, which make it a robust programming language. These features ensure that the Java applications run securely and effectively on a machine and do not crash a machine if minor errors are encountered.
- **Multithreaded**: Java provides the concept of multithreading that allows a program to simultaneously execute multiple tasks by using threads. A *thread* is the smallest unit of execution in a Java program.

Java Architecture

The Java architecture defines the components that are essential to carry out the creation and execution of code written in the Java programming language. The various components of the Java architecture are:

- Source file
- Class file
- JVM
- Application Programming Interface (API)

The various components in the Java architecture are shown in the following figure.



Various Components in the Java Architecture

Source File

Java is a robust programming language that enables developers to build a wide variety of applications for varying purposes. In order to write a Java program or an application, you need to write certain code. A Java application contains the source code that enables an application to provide its desired functionalities. This source code is saved in a source file with the extension, .java.

Class File

Once created, a .java file is compiled to generate the .class file. A class file contains the bytecode that is generated by the compiler. Further, this class file can be executed by any machine that supports the Java Runtime Environment (JRE).

JVM

The JVM is an application that is responsible for executing Java programs on a computer. It resides in the Random Access Memory (RAM) and is specific for every platform, such as Sun and Macintosh. The bytecode generated during the compilation of a program is the machine language of the JVM. JVM is responsible for executing the bytecode and generating the machine specific code for the machine on which the Java program needs to be executed.

The major components of JVM are:

- Class loader
- Execution engine and Just-In-Time (JIT) compiler
- Bytecode verifier

Class Loader

The class loader loads the class files required by a program that needs to be executed. The classes are loaded dynamically when required by the running program.

Execution Engine and JIT Compiler

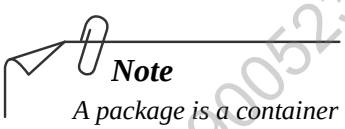
The Java execution engine acts as an interpreter that interprets the bytecode one line after another and executes it. The line-by-line interpretation is a time-consuming task and reduces the performance of a program. To enhance the performance, the *JIT compiler* has been introduced in the JVM. The JIT compiler compiles the bytecode into machine executable code, as and when needed, which optimizes the performance of a Java program.

Bytecode Verifier

Before being executed by the JVM, the bytecode is verified by using the bytecode verifier. The bytecode verifier ensures that the bytecode is executed safely and does not pose any security threat to the machine.

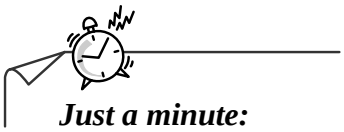
API

The Java API is a collection of software components, such as classes, interfaces, and methods, which provides various capabilities, such as GUI, Date and Time, and Calendar. The related classes and interfaces of the Java API are grouped into packages.



Note

A package is a container of classes.



Just a minute:

Which one of the following features of Java allows a program to simultaneously execute multiple tasks?

1. *Multithreading*
2. *Portability*
3. *Distributed*
4. *Garbage collection*

Answer:

1. *Multithreading*

Identifying the Building Blocks of a Java Program

Consider the scenario of DailyGames. In order to develop the Classic Jumble Word application and achieve its desired functionality, Sam needs to build a Java program. A program in Java comprises the following building blocks:

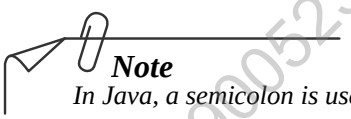
- Classes
- Data types
- Class members
- Packages

Defining a Class

A class defines the characteristics and behavior of an object. For example, if you create a gaming application, each game that a user can play from the application can be considered as an object of the `Games` class. Each game has common characteristics, such as the number of players, game category, and score. These characteristics are known as the member variables, and the behavior is specified by methods. Any concept that you need to implement in a Java program is encapsulated within a class. A class defines the member variables and methods of objects that share common characteristics. Further, all the games have common methods, such as calculating score, starting the game, and displaying game instructions.

The following code snippet shows how to declare a class:

```
class <ClassName>
{
//Declaration of member variables
//Declaration of methods
}
```



Note

In Java, a semicolon is used to mark the end of a statement.

In the preceding code snippet, the word, `class`, is a keyword in Java that is used to declare a class; and `<ClassName>` is the name given to the class.

Sam can use the following code snippet to create the `ClassicJumble` class to develop the Classic Jumble Word game:

```
class ClassicJumble {
//Declaration of member variables
//Declaration of methods
}
```

There are certain rules that should be followed to name Java classes. A program will raise error if these rules are not followed. Some of these rules are:

- The name of a class should not contain any embedded space or symbol, such as ?, !, #, @, %, &, {}, [], :, ;, “, and /.
- A class name must be unique.
- A class name must begin with a letter, an underscore (_), or the dollar symbol ($). Or, it must begin with an alphabet that can be followed by a sequence of letters or digits (0 to 9), ‘$’, or ‘_’.
- A class name should not consist of a keyword.

While naming a class, you should adhere to the following class naming conventions that improve the readability of a program:

- The class name should be a noun.
- The first letter of the class name should be capitalized.
- If the class name consists of several words, the first letter of each word should be capitalized.

Keywords are the reserved words with a special meaning for a language, which express the language features. Keywords cannot be used to name variables or classes. Java is a case-sensitive language, and the keywords should be written in lowercase only. The following table lists the Java keywords.

abstract	*boolean*	*break*	*class*
case	*catch*	*char*	*do*
const	*continue*	*default*	*final*
double	*else*	*extends*	*goto*
finally	*float*	*for*	*instanceof*
if	*implements*	*import*	*native*
int	*interface*	*long*	*protected*
new	*package*	*private*	*static*
public	*return*	*short*	*synchronized*
strictfp	*super*	*switch*	*transient*
this	*throw*	*throws*	*while*
try	*void*	*volatile*	
enum	*assert*	*byte*	

Set of Keywords in Java

After defining a class, you need to save the file before it can be executed. The following naming conventions should be followed for naming a Java file:

- A file name must be unique.
- A file name cannot be a keyword.
- If a class is specified as `public`, the file name and class name should be the same.
- If a file contains multiple classes, only one class can be declared as `public`. The file name should be the same as the class name that is declared `public` in the file.
- If a file contains multiple classes that are not declared `public`, any file name can be specified for the file.

Note

In Java, class names are case-sensitive. For example, `vehicle` is not the same as `Vehicle`.

Identifying Data Types

While working with an application, you need to store and manipulate varying data. For example, in the Classic Jumble Word game, the game score is a numeric value and the player name is a string. To handle such varying data in an application, Java supports various data types. There are eight primitive data types in Java, which are further grouped into the following categories:

- **Integer type**: Can store integer values. The four integer data types are:
  - `byte`
  - `short`
  - `int`
  - `long`

  The following table lists the integer primitive data type, its data types, size, range, and default values.

Group	*Data Type*	*Size*	*Range*	*Default Value*
Integer	*`byte`*	*1 byte*	*-2^7 to 2^7-1*	*0*
	`short`	*2 bytes*	*-2^{15} to $2^{15}-1$*	*0*
	`int`	*4 bytes*	*-2^{31} to $2^{31}-1$*	*0*
	`long`	*8 bytes*	*-2^{63} to $2^{63}-1$*	*0*

The Details of the Integer Primitive Data Type

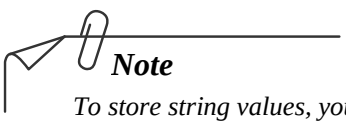
- **Floating point type**: Can store decimal numbers. The two floating point types are:
  - `float`
  - `double`

  The following table lists the floating point primitive data type, its data types, size, range, and default values.

Group	*Data Type*	*Size*	*Range*	*Default Value*
Floating point	`float`	*4 bytes*	3.4^{e-038} *to* 3.4^{e+038}	*0.0*
	`double`	*8 bytes*	1.7^{e-308} *to* 1.7^{e+308}	*0.0*

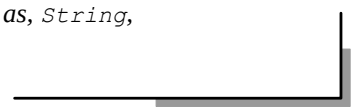
The Details of the Floating Point Data Type

- **Boolean type**: Can store only the values, `true` and `false`.
- **Character type**: Can store a single character, such as a symbol, letter, and number. In the character type, there is one data type, `char`. To a character data type, `2` bytes of memory is allocated.



Note

To store string values, you need to use built-in classes, such as, `String`, `StringBuilder`, and `StringBuffer`.



Wrapper Classes

Variables that are declared by using the primitive data types are not objects. The primitive data types, such as `int` and `char`, are not a part of the object hierarchy. Therefore, in order to use the primitive data types as objects, Java provides wrapper classes. A wrapper class acts like an object wrapper and encapsulates the primitive data types within the class so it can be treated like an object.

The various wrapper classes, such as `Boolean`, `Character`, `Integer`, `Short`, `Long`, `Double`, and `Float`, are provided by the `java.lang` package. These classes are useful when you need to manipulate primitive data types as objects.

Identifying Class Members

In Java, a class can contain the following members:

- Variables
- Methods
- Objects
- Inner classes

Variables

A variable is used essentially as a container for the storage of varying kinds of data. The main use of a variable is to store and manipulate data or values that are used in Java programs. A variable represents a name that refers to a memory location where some value is stored. You can assign varying values to a variable during program execution. However, the data type of the value must be the same as the data type of the variable. In Java, each variable that is used in a program must be declared with a data type.

A variable needs to be declared before it is accessed. The declaration of a variable informs the compiler about the variable name and the type of value stored in it. The following code snippet shows how to declare a variable:

```
<type> <variablename>; // Single variable of given type.
<type> <variable1name,variable2name.....variable_n_name>;// Multiple
variables of given type.
```

For example, in the Classic Jumble Word game, upon the start of the game, the user needs to be presented with a menu. The menu will present the user with the options to determine if the user wishes to play the game, read game instructions, or exit. The user can enter `1`, `2`, or `3` to refer to the preceding choices. In such a case, you can use a variable named `choice` to store user input to determine the course of the execution of the game. You can use the following code snippet to declare a variable:

```
int choice;
```

You can refer to a variable by using its name in a program. You can assign values to a variable during declaration or after the declaration of the variable. For example, you can use the following code snippet to declare the variable, `num1` and `num2`, to calculate the sum of two numbers:

```
int num1, num2;
```

The following code snippet shows how to assign values to a variable:

```
<type> <variablename>=<value>; // During declaration.
<variablename>=<value> // After declaration.
```

For example, you can use the following code snippet to assign values to the variables, `num1` and `num2`:

```
int num1, num2; // Declaration of variables.
num1 = 5; // Assigning values to the variables.
num2 =10;
```

The following code snippet can be used to assign a value to a variable at the time of its declaration.

```
int num1=5;
```

You can also assign the same value to more than one variable in a single statement. The following code snippet shows the assignment of the same value to more than one variable:

```
a=b=c=3;
```

In the preceding code snippet, the integer value, `3`, is first assigned to `c`, then to `b`, and finally to `a`.

Note

By default, when a member variable is declared in a class but not initialized, it is assigned a default value. For example, an integer member variable is assigned the value, `0`, by default.

Note

In Java, the rules for naming a variable are similar to the rules for naming a class.

To understand the type of values that a variable can contain, you need to understand the concept of literals. A literal is a value that is assigned to a variable or constants in a Java program. A literal contains a sequence of characters, such as digits, alphabets, or any other symbol, which represents the value to be stored. The various types of literals in Java are:

- **Integer literals**: Are non-fractional numeric values. The numerical values can be represented in the decimal, octal, and hexadecimal notation. In order to represent an octal number, it is preceded by zero. In order to represent hexadecimal numbers, they are prefixed by `0x`. For example, `n=0567` is integer literal represented in the octal notation whereas `n=0x124` represents an integer literal in hexadecimal notation.
- **Floating point literals**: Are numeric values that contain a fractional part. The floating point literals contain a decimal and an integer part. For example, `z=2.2` represents a floating point literal.
- **Character literals**: Are represented inside a pair of single quotation marks. For example, `x='k'` is a character literal.
- **String literals**: Are enclosed in a pair of double quotation marks. For example, `x="James"` is a string literal.
- **Boolean literals**: Are the literals that can contain the values, true or false. For example, `x= false` is a boolean literal.
- **Binary literals**: Are the literals that can be used to express the primitive integer data types into binary form. In order to specify these literals, you need to prefix the binary number with `0b` or `0B`. For example, `0b101`.

Note

The integer and floating point literals can also contain the '_' character.

Methods

A *method* is a set of statements that is intended to perform a specific task. For example, a `compute()` method can be used for computing the score. Moreover, methods provide encapsulation and are also essential to refer to the data members and access them. A method consists of two parts, method declaration and method body.

The syntax for defining a method is:

```
<Access specifier> <Return type> <Method name>(Parameter list) //method
declaration
{
    <Method body> // body of the method
}
```

In the preceding syntax, the first line depicts the method declaration and the contents within the curly braces depict the method body. According to the preceding syntax, the various elements of a method are:

- **Access specifier**: Determines the extent to which a method can be accessed from another class.
- **Return type**: A method can also return a value. The return type specifies the data type of the value returned by a method. Some methods do not return any value. Such methods should use the `void` return type. The value to be returned by the method is specified with the keyword, `return`.
- **Method name**: Is a unique identifier and is case-sensitive.
- **Parameter list**: A method may or may not contain a parameter list depending upon its functionality. The parameter list consists of a list of parameters separated by commas. Parameters are used to pass data to a method. The parameter list is written between parentheses after the method name. The parentheses are included even if there are no parameters.
- **Method body**: This contains the set of instructions that perform a specific task. The variables declared inside the method body are known as local variables.

While adding a method to a class, you should adhere to the following method naming conventions:

- The method names should be verb-noun pairs.
- The first letter of the method should be in lowercase.
- If the method name consists of several words, the first letter of each word, except the first word, should be capitalized.

Consider the following code snippet:

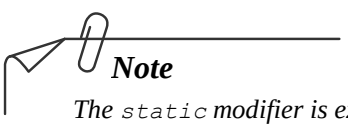
```
class ClassicJumble {
int score;
public int startGame(){
int choice;

…

        System.out.println("**********JUMBLE GAME***********");
        System.out.println("GAME MENU");
        System.out.println("Please select an option from the menu");
        System.out.println("1. Play Game");
        System.out.println("2. Instructions");
        System.out.println("3. Quit Game");
        System.out.println("\nEnter your choice: ");
```

```
            …
            …
            return choice;
            compute();
            }
}
```

In the preceding code snippet, a method named `startGame()` is defined. This method displays the game menu options for the Classic Jumble Word game application and returns the choice of the user. To get the input from a user, you can use the `Scanner` class in Java. This class is contained in the `java.util` package and must be imported in a program before it is used. The following code snippet can be used to import the `Scanner` class in a Java program:

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

The variable, `score`, is defined inside the class body, but outside the `startGame()` method. Such variables, which are defined inside the class body without the static modifier but outside the body of any method inside a class, are called instance variables. The scope of the instance variables spans across the entire class, and they can be used by the class methods. The variables defined inside a method of a class are called local variables. The scope of local variables is within the block of code in which they are defined. They are local to the block of code and are not accessible outside the method.



Note

The `static` modifier is explained later in the chapter.

To execute a method, you need to invoke the same. When a method is invoked from a Java program, the control is transferred to that method. The syntax for calling a method is:

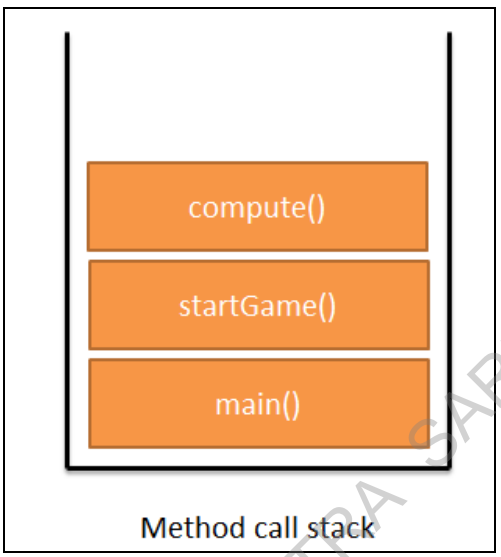
```
<Method name>(argument list);
```

The argument list is to be specified if the method uses a parameter list in its implementation. The following code snippet can be used to invoke the `startGame()` method:

```
startGame();
```

Usually, an application involves the invoking of multiple methods. Further, a method may also be invoked by another method. The invoked methods are managed by a stack. For example, in the Classic Jumble Word application, the `startGame()` method is the first to be executed after the `main()` method, and it displays the game menu to the user. The `startGame()` method will further invoke the `compute()` method that will identify the game menu input by the user and respond accordingly.

The following figure represents the stack of method calls for the `main()`, `startGame()`, and `compute()` methods.



The Stack of Method Calls

In object-oriented languages, when you instantiate a class, special methods called constructors are automatically called. *Constructors* are methods that have the same name as that of the class and have no return type. Constructors are used to construct an object of a class and initialize it.

Consider the following code snippet:

```
class ClassicJumble
{

int score;
    ClassicJumble()
    {
    score= 10 ;
    }
}
```

In the preceding code snippet, a constructor named `ClassicJumble` is created. This constructor will initialize the `score` member variable with the value, `10`, whenever an object is created. Classes can also have parameterized constructors. You can pass parameter values to such constructors. These parameter values are usually assigned to the attributes of the class before creating an object of the class. The syntax for declaring a parameterized constructor is:

```
className (datatype parameter1, datatype parameter2..)
{
    // code to construct the object
}
```

For example, you can create a parameterized constructor for the `Square` class, as shown in the following code snippet:

```
class Square
{
int length;
    Square(int l)
    {
    length = l ;
    }
    int area()
    {
    return length*length;
    }
}
```

Objects

An *object* is an instance of a class and has a unique identity. The identity of an object distinguishes it from other objects. Classes and objects are closely linked to each other. While an object has a unique identity, a class is an abstraction of the common properties of various objects. To create an object, you need to declare, and then, instantiate an object. Declaring an object creates a variable that will hold the reference to the object. The following code snippet shows how to declare an object of the class:

```
class_name object_name;
```

When you declare an object, the memory is not allocated to it. To allocate memory to the object, you need to instantiate the object by using the `new` operator. The `new` operator allocates memory to an object. It also returns a reference to that memory location in the object variable. The following code snippet shows how to create an object:

```
object_name= new class_name();
```

You can declare and instantiate an object in a single statement. You can use the following code snippet to declare and instantiate an object, `GameObject`, of the `ClassicJumble` class in a single statement:

```
ClassicJumble GameObject = new ClassicJumble();
```

You can use the following code snippet to create three game objects of the `ClassicJumble` class:

```
ClassicJumble GameObject1 = new ClassicJumble();
ClassicJumble GameObject2 = new ClassicJumble();
ClassicJumble GameObject3 = new ClassicJumble();
```

Note

After importing the `Scanner` *class, you need to create its object by using the following code snippet:*

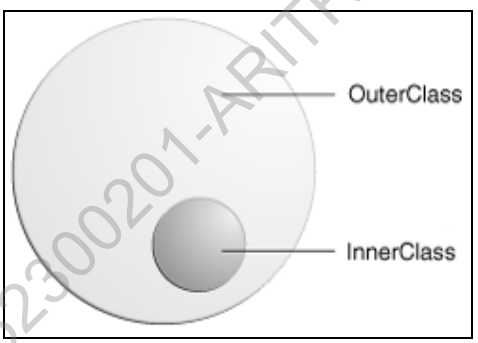
```
Scanner <objectname> = new Scanner(System.in);
```

Then, the predefined methods of the `Scanner` *class, such as* `nextInt()` *and* `nextLine()` *can be used to read an integer or a string value, respectively. The following code snippet can be used to read a string value and store it in a string reference named* `option`*:*

```
String option = <objectname>.nextLine();
```

Inner Classes

In Java, a class can also contain another class. Such a class is known as an inner class or a nested class. The class that contains the inner class is known as an outer class. The concept of an inner class is shown in the following figure.



The Concept of an Inner Class

The concept of inner classes has the following benefits:

- It improves encapsulation as a class can be hidden inside another class.
- It provides a better readability.
- It is useful for logically grouping classes.

The following code snippet demonstrates an inner class:

```
class outerclass
{
…
    class innerclass
    {
    …
    }
}
```

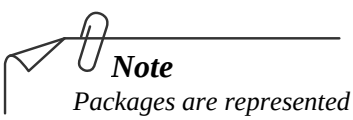
The following types of inner classes can be created in Java:

- **Regular inner class**: A regular inner class has complete access to all the members of the outer enclosing class. It is contained within the outer class.
- **Static nested class**: A static nested class is like a regular inner class. It is defined with the keyword, `static`. However, in order to create an object of the inner class, it is not necessary to instantiate the outer class.
- **Method-local inner class**: A method-local inner class is a class that is defined inside a method contained in the outer class. A method local inner class also needs to be instantiated inside the method in which it is defined.
- **Anonymous inner class**: An anonymous inner class is an inner class that has no name.

Defining a Package

A package is a collection of classes. A package provides the space essentially used to organize classes that are related to each other. You can create multiple packages to organize multiple categories of classes, as per the requirement. Moreover, packages ensure that the non-related classes with the same name do not conflict with each other. A class can belong only to one package. For example, a package named `GameConsole` can be used to create and store a class named `Game1`. This will ensure that the `Game1` class does not conflict with any that has the same name and is stored somewhere else. The following syntax is used to define a package:

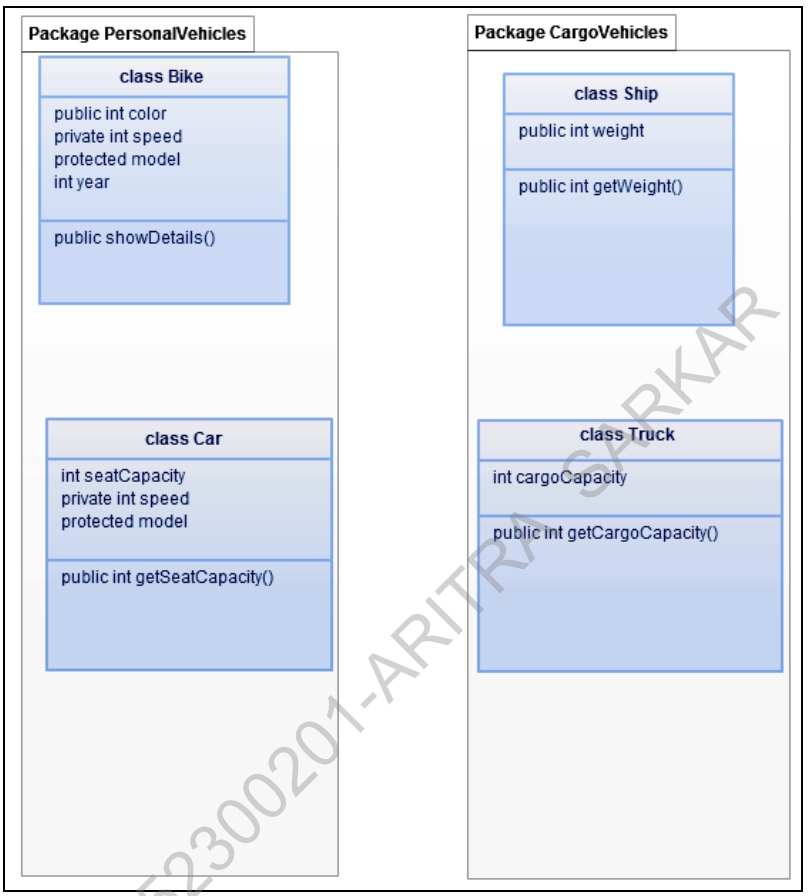
```
package <package_name>;
```



Note

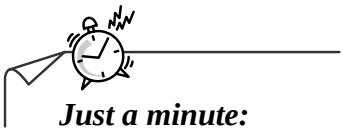
Packages are represented and stored by Java as directories in the file system.

The following figure represents the concept of packages in Java.



The Concept of Packages

In the preceding figure, the packages, `PersonalVehicles` and `CargoVehicles`, represent the category of those vehicles that are used for personal use and commercial transport, respectively. The `PersonalVehicles` package contains the Bike and Car classes. The `CargoVehicles` package contains the `Ship` and the `Truck` classes that further comprise their respective class members and methods.



Just a minute:

Which one of the following types of literals can be specified only by values, true or false?

1. *String*
2. *Boolean*
3. *Integer*
4. *Floating point*

Answer:

2. *Boolean*

Accessing Class Members

The class members describe the characteristics and behavior of an object. You may need to hide or protect certain class members from other classes in a Java application. In other words, you may want to restrict access to sensitive data that belongs to a class so that it is not modified. Moreover, restricting access to the members of a class is a way of achieving information hiding or encapsulation and ensuring that the intended functionality of a class is not modified.

For example, in the Classic Jumble Word game application, you might need to hide sensitive information outside the `ClassicJumble` class, such as the jumbled-up words to be displayed to the game user. To cater to such requirements and improve the security of a class, you can use access specifiers and modifiers in Java.

The access specifiers and modifiers provided in the Java programming language are used to identify the part of the class, such as data members and methods that needs to be accessed by other class objects. Further, it is also useful for deciding the behavior of data members when they are used in other classes and objects.

Using Objects

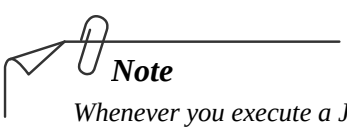
Objects are used to access the members of a class. You can access the data members of a class by specifying the object name followed by the dot operator and the data member name. The following code snippet shows how to access the data members of a class:

```
object_name.data_member_name
```

In the preceding code snippet, `object_name` refers to the name of the object and `data_member_name` refers to the name of the data variable inside the class that you want to access.

To access data members of the `ClassicJumble` class, such as `choice`, and access the member through the object, `obj1`, you can use the following code snippet:

```
class ClassicJumble {
    int choice=1;
    public static void main(String[] args) {
       ClassicJumble obj1 = new ClassicJumble();

            System.out.println(obj1.choice);
    }
}
```



Note

Whenever you execute a Java application, the `main()` method is the first to be executed by the JVM.

Similar to variables, the member method of a class is also accessed by using objects, as shown in the following code snippet:

```
class ClassicJumble {
    int choice;
    void show()
    {
        System.out.println(choice);
    }
    public static void main(String[] args) {
        ClassicJumble obj1 = new ClassicJumble();
                obj1.show();
    }
}
```

In the preceding code snippet, the `ClassicJumble` class contains the `show()` method that displays the value of the variable, `choice`. The `show()` method is called by the object, `obj1`.

Using Access Specifiers

An access specifier controls the access of class members. The various types of access specifiers in Java are:

- `private`
- `protected`
- `public`

The private Access Specifier

The `private` access specifier allows a class to hide its member variables and member methods from other classes. Therefore, the private members of a class are not visible outside a class. They are visible only to the methods of the same class. Therefore, the data remains hidden and cannot be altered by any method other than the member methods of the class. If a variable or method is declared `private`, it can be accessed only within its enclosing class. A top level class cannot be declared `private` in Java. However, an inner class can be declared `private`.

The following code snippet shows how to declare a `private` data member of a class:

```
private <data type> <variable name>;
```

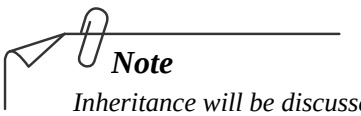
The following code snippet shows how the `private` access specifier can be implemented in the `ClassicJumble` class for the `score` variable:

```
class ClassicJumble {
   private int score;
    void show()
    {
    }
}
```

In the preceding code snippet, the `score` variable has been specified with the `private` access specifier. The `score` variable can be accessed anywhere in the `ClassicJumble` class, but it is not accessible to other classes.

The protected Access Specifier

The members of a class that are preceded with the `protected` access specifier are accessible to all the classes within the package and by the subclasses outside the package. The `protected` access specifier becomes important while implementing inheritance..



Note

Inheritance will be discussed in detail in the later chapters.

The following statement shows how to declare a member as `protected`:

```
protected <data type> <name of the variable>;
```

Consider the following code snippet:

```
class ClassicJumble {
   protected int score;
    void show()
    {
    }
}
```

In the preceding code snippet, the `score` variable is declared as `protected`. It can, therefore, be accessed within the `ClassicJumble` class from the classes that will inherit the from `ClassicJumble` class. It can also be accessed within the classes of the package that contains the `ClassicJumble` class.

The public Access Specifier

The members of a class that are preceded with the `public` access specifier are accessible by the classes present within the package or outside the package. You can access a public class, data member, or method within the class in which they are defined, from the classes defined in the same package or outside the package. The following code snippet shows how to declare a data member of a class as `public`:

```
public <data type> <variable name>;
```

The following code snippet shows how the `public` access specifier can be implemented in the `ClassicJumble` class for the `choice` variable:

```
class ClassicJumble {
   public int choice;
    void show()
    {
                System.out.println(choice);
    }
}
```

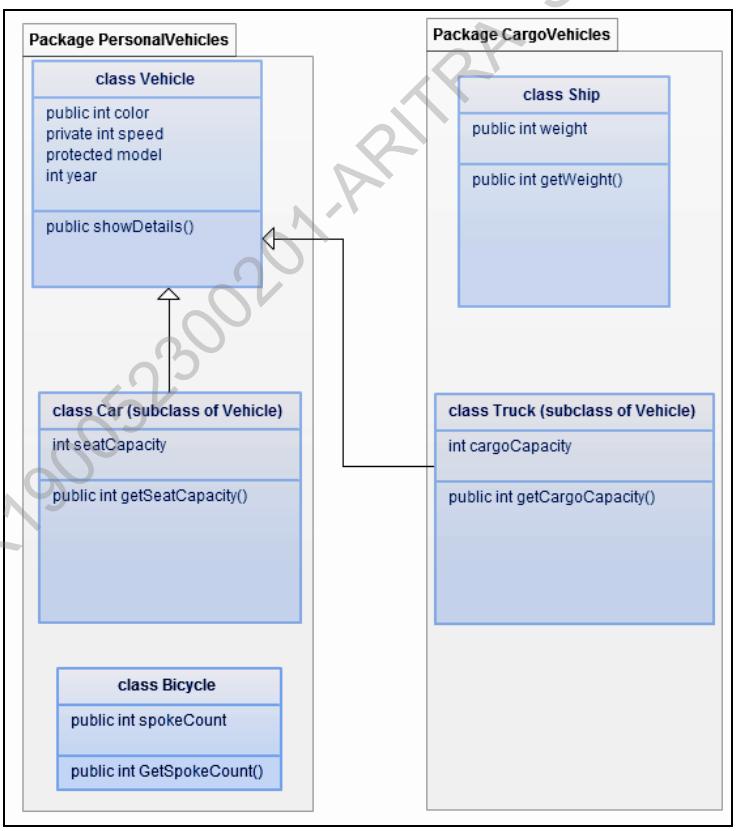
In the preceding code snippet, the `choice` variable has been specified with the `public` access specifier. The `choice` variable can be accessed anywhere in the `ClassicJumble` class. It is also accessible in the package in which the `ClassicJumble` class is created, or in a package that is different from the one where the `ClassicJumble` class exists.

In addition to the `public`, `private`, and `protected` access specifiers, the class members can have a default access. If you do not specify any of the preceding access specifiers, the scope of data members and methods is default or friendly. A class, variable, or a method with a friendly access can be accessed only by the classes that belong to the package in which they are present.

Note

The variables inside a method cannot be specified with any access specifier.

The following figure can be used to understand the usage of access specifiers in Java.



An Overview of the Usage of Access Specifiers

The accessibility of the variables mentioned in the preceding figure is described in the following table.

Variable	*Access specifier*	*Accessible by*
`color`	`public`	`Vehicle`, `Car`, `Bicycle`, `Ship`, and `Truck` classes
`speed`	`private`	`Vehicle` class
`model`	`protected`	`Vehicle`, `Car`, and `Ship` classes
`year`	`default access`	All the classes of the `PersonalVehicles` package

The Usage of Access Specifiers

Note

The hierarchy for the access specifiers arranged from the least restrictive level to the most restrictive level is `public`, `protected`, default access, and `private`.

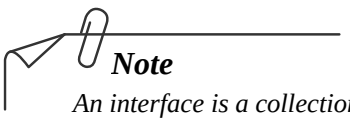
Using Access Modifiers

Access modifiers are useful to determine or define the manner in which data members and methods are utilized in other classes and objects. The access modifiers determine how the members are used and modified by other classes. In comparison to the access modifiers, access specifiers define the accessibility of the data members of a class. The various modifiers permitted in Java are::

- `final`
- `static`
- `abstract`
- `native`
- `synchronized`
- `transient`
- `volatile`
- `strictfp`

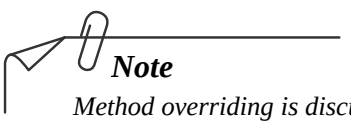
final

The `final` keyword is applicable to methods, variables, and classes. The `final` keyword cannot be used with an interface in Java. When used with a data member, the `final` keyword represents that the data member definition or value cannot be further replaced or modified. For example, consider a variable that has been assigned a specific value. If you try to modify the value of a variable that is declared as `final`, it will cause compile-time errors. Furthermore, a class that is declared as `final` cannot be used for the purpose of inheritance, and a method declared as `final` cannot be overridden.



Note

An interface is a collection of abstract methods and data members. They are covered in detail in a later chapter.



Note

Method overriding is discussed in a later chapter.

static

The `static` keyword is applicable to methods, variables, and inner classes. The `static` keyword is used to denote those class variables and methods that belong specifically to a class and not to any of its particular objects. Therefore, if any method modifies the value of a static variable, it can be seen by all the other instances. The static variables are allocated only once. Therefore, when an instance of a class is destroyed, the static variable is not destroyed and remains available to other instances of that class.

A static method associates the data members with a class and not with the objects of the class. Therefore, all the objects of a class share the same static data members and methods. A static method is a class method and can be invoked before the instance of a class is created.

For example, the `main()` method is a static method and can be invoked without creating an instance of the class to which the `main()` belongs.

abstract

The `abstract` keyword is essentially used for the declaration of those classes that define only the common properties and behavior that can be utilized in other classes. A class declared as abstract cannot be instantiated. The abstract class contains the declaration of methods. You can also use the `abstract` keyword with methods. An abstract method is declared without its implementation details. An abstract method is overridden by classes that inherit an abstract class.

For example, for the Classic Jumble word application, you can create an abstract class named `Game` with abstract methods, as shown in the following code snippet:

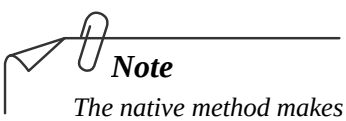
```
abstract class Game {
int score;
abstract void startGame();
     abstract void jumble();
     abstract void play_again();
          abstract void compute();

}
```

native

The `native` keyword can be specified only with methods to notify the compiler that the method implementation is not written in the Java programming language but in another language, such as C or C++. The `native` keyword with a method indicates that the method needs to be imported in the JRE and as it is present outside the JRE.

The following code snippet shows how to declare a native modifier:

```
public native void nativeMethod(var1, var2, . . .) ;
```



Note

The native method makes a program platform dependent. In addition, writing native methods must be avoided. They are used when you have an existing code in another language and do not want to rewrite the code in Java.

synchronized

The `synchronized` keyword is used for methods to control their access in a multithreaded programming environment. A thread represents the smallest unit of execution within a process. In the multi-threaded environment of Java, each thread defines a separate path of execution.

The `synchronized` keyword can be used in a multithreaded program to ensure that only a single thread is allowed access to a shared resource when two or more threads need access to that resource at the same time. A synchronized block of code restricts its execution by one or more threads at the same time. For example, when synchronization is applied to a variable, then, only one thread at a time can access the variable.

transient

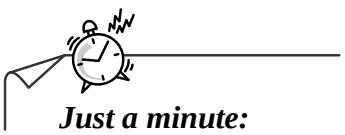
In Java, the properties and state of an object can be saved by using a technique named serialization. The keyword, `transient`, can be used to control serialization as it specifies the object properties that can be excluded from serialization. The object properties or attributes marked as `transient` will not be saved to disk.

volatile

The `volatile` keyword is used with a variable to specify that its value is updated almost every time it is accessed. Therefore, the volatile keyword prevents the Java compiler from tracking the changes that are made to a variable that is specified as `volatile`. In case of a multithreaded program, the `volatile` variable specifies that its value can be modified by multiple threads at undetermined intervals.

strictfp

The `strictfp` keyword can be used with a method or a class. This keyword specifies that the floating point calculations carried out in a class are as per the specifications of Java. This keyword can be used to ensure that the floating point computations can result precisely on varying platforms.



Just a minute:

Which one of the following options is called automatically when a class is instantiated?

1. *Constructor*
2. *Variable*
3. *Abstract method*
4. *Package*

Answer:

1. *Constructor*

Activity 1.1: Creating and Executing a Class

Practice Questions

1. Which one of the following options is the correct extension of a Java source file?
   a. .jav
   b. .java
   c. .JAVA
   d. .class

2. Which one of the following keywords controls the access to a block of code in a multithreaded programming environment?
   a. `synchronized`
   b. `native`
   c. `volatile`
   d. `abstract`

3. Which one of the following keywords is used to inform the compiler that the method has been coded in a programming language other than Java?
   a. `synchronized`
   b. `native`
   c. `volatile`
   d. `abstract`

4. Which one of the following access specifiers represents the least restrictive level of access?
   a. `private`
   b. `public`
   c. `protected`
   d. default access

5. Which one of the following literals is specified by using single quotation marks?
   a. string
   b. character
   c. float
   d. integer

Summary

In this chapter, you learned that:

- Java provides powerful features that make it a popular and widely used programming language. Some of these features are:
  - Simple
  - Object-oriented
  - Platform independence
  - Portable
  - Distributed
  - Secure
  - Robust
  - Multithreaded
- The various components of the Java architecture are:
  - Source file
  - Class file
  - JVM
  - API
- A program in Java comprises the following building blocks:
  - Classes
  - Data types
  - Class members
  - Packages
- A class defines the characteristics and behavior of an object.
- Keywords are the reserved words with a special meaning for a language, which express the language features.
- There are eight primitive data types in Java, which are further grouped into the following categories:
  - Integer type
  - Floating point type
  - Boolean type
  - Character type
- A wrapper class acts like an object wrapper and encapsulates the primitive data types within the class so it can be treated like an object.
- In Java, a class can contain the following members:
  - Variables
  - Methods
  - Objects
  - Inner classes

- A variable represents a name that refers to a memory location where some value is stored.
- A method is a set of statements that is intended to perform a specific task.
- Constructors are used to construct an object of a class and initialize it.
- An object is an instance of a class and has a unique identity.
- A package is a collection of classes.
- Objects are used to access the members of a class.
- An access specifier controls the access of class members. The various types of access specifiers in Java are:
  - `private`
  - `protected`
  - `public`
- Access modifiers determine or define how the data members and methods are used in other classes and objects.
- The various modifiers permitted in Java are:
  - `final`
  - `static`
  - `abstract`
  - `native`
  - `synchronized`
  - `transient`
  - `volatile`
  - `strictfp`

NIIT

Implementing Operators

CHAPTER 2

Operators allow you to perform various mathematical and logical calculations, such as adding and comparing numbers.

In addition, if there are multiple operations that need to be performed in an expression, each operator in an expression is evaluated in a predetermined order called operator precedence.

This chapter discusses different types of operator used in Java. In addition, it discusses operator precedence.

Objectives

In this chapter, you will learn to:

- Work with operators
- Use operator precedence

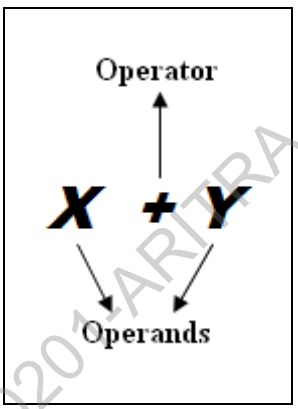
Working with Operators

In the Classic Jumble Word game, Sam needs to calculate the number of attempts in which a user gives the correct answer for a jumbled word. To implement this, the application should validate the user input and increment the counter by one on each attempt the user provides an incorrect input. For such calculations and comparisons, Java provides various types of operators. An operator is a special symbol that is combined with one or more operands to perform specific operations, and then return a result.

Consider the following expression:

```
X+Y
```

In the preceding expression, `X` and `Y` are operands, and `+` is an operator. The following figure shows the operator and operands used in the preceding expression.



The Operator and Operands

The following types of operators are supported in Java:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Unary operators
- Bitwise operators
- Shift operators
- Ternary operator

Using the Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on operands.

The following table describes the arithmetic operators.

Operator	***Operator Name***	***Description***	***Example (In the following examples, the value of Y is assumed to be 20 and the value of Z is assumed to be 2)***
`+`	*Plus*	*Used to add two numbers.*	`Y+Z;` *will return the value,* `22`.
`-`	*Minus*	*Used to subtract two numbers.*	`Y-Z;` *will return the value,* `18`.
`*`	*Multiply*	*Used to multiply two numbers.*	`Y*Z;` *will return the value,* `40`.
`/`	*Divide*	*Used to divide one number by another.*	`Y/Z;` *will return the value,* `10`.
`%`	*Modulus*	*Used to find the remainder after dividing the two numbers.*	`Y%Z;` *will return the value,* `0`.

The Arithmetic Operators

Using the Assignment Operators

The assignment operators can be categorized into:

- Simple assignment operator
- Complex assignment operator

Simple Assignment Operator

At times, during the programming, you want to assign a value to a variable. For this, you need to use the simple assignment operator, =.

Consider the following code snippet in which the variable, `x`, is assigned the value, `5`:

```
x=5;
```

Complex Assignment Operator

In Java, arithmetic operators can be combined with simple assignment operators. These operators are called complex assignment operators. The complex assignment operators are also known as arithmetic assignment operators. The benefits of these operators are that they are shorthand for their equivalent long forms. For example, in the expression, `x = x+4`, the `+` arithmetic operator adds `4` to the variable, `x`, and then assigns the result to the variable, `x`. The preceding expression is equivalent to the expression, `x += 4`.

The following table describes the complex assignment operators.

Operator	***Description***	***Example*** ***(In the following examples, the value of X is assumed to be 20 and the value of Y is assumed to be 2)***
`+=`	*Used to add two numbers and assign the result to a variable.*	`X+=Y;` *X will have the value,* `22`.
`-=`	*Used to subtract two numbers and assign the result to a variable.*	`X-=Y;` *X will have the value,* `18`.
`*=`	*Used to multiply two numbers and assign the result to a variable.*	`X*=Y;` *X will have the value,* `40`.
`/=`	*Used to divide one number by another and assign the result to a variable.*	`X/=Y;` *X will have the value,* `10`.
`%=`	*Used to find the remainder and assign the result to a variable.*	`X%=Y;` *X will have the value,* `0`.

The Complex Assignment Operators

Using the Comparison Operators

Comparison operators are used to compare two values and perform an action on the basis of the result of that comparison. Whenever you use a comparison operator, the expression results in the boolean value, `true` or `false`.

The following table describes the commonly-used comparison operators.

Operator	***Operator Name***	***Description***	***Example*** ***(In the following examples, the value X is assumed to be 20 and the value of Y is assumed to be 25)***
<	*Less than*	*Used to check whether the value of the left operand is less than the value of the right operand.*	`boolean Result;` `Result = X < Y;` `Result` *will have the value, true.*
>	*Greater than*	*Used to check whether the value of the left operand is greater than the value of the right operand.*	`boolean Result;` `Result = X > Y;` `Result` *will have the value, false.*
<=	*Less than or equal to*	*Used to check whether the value of the left operand is less than or equal to the value of the right operand.*	`boolean Result;` `Result = X <= Y;` `Result` *will have the value, true.*
>=	*Greater than or equal to*	*Used to check whether the value of the left operand is greater than or equal to the value of the right operand.*	`boolean Result;` `Result = X >= Y;` `Result` *will have the value, false.*
==	*Equal to*	*Used to check whether the value of the left operand is equal to the value of the right operand.*	`boolean Result;` `Result = X == Y;` `Result` *will have the value, false.*
!=	*Not equal to*	*Used to check whether the value of the left operand is not equal to the value of the right operand.*	`boolean Result;` `Result = X != Y;` `Result` *will have the value, true.*

The Comparison Operators

In addition to the preceding listed operator, there is one more comparison operator called the `instanceof` operator. The *`instanceof`* operator is used to test whether an object is an instance of a specific class at runtime or not.

The syntax of the `instanceof` operator is:

```
op1 instanceof op2
```

In the preceding syntax, `op1` is the name of an object and `op2` is the name of a class. The `instanceof` operator returns the `true` value if the `op1` object is an instance of the `op2` class. It returns the `false` value if the `op1` object is not an instance of the `op2` class.

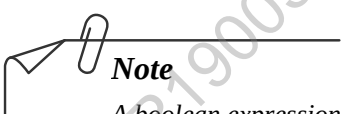
Using the Logical Operators

Logical operators are used to evaluate operands and return a boolean value.

The following table describes the logical operators.

Operator	***Operator Name***	***Description***	***Example***
`&&`	*Logical AND*	*Used to compare two boolean expressions and returns true if both the boolean expressions are true.*	`boolean Result=(5> 10&& 23<56);` `Result` *will have the value, false.*
`||`	*Logical OR*	*Used to compare two boolean expressions and returns false if both the boolean expressions are false, else returns true if any one of the boolean expression is true.*	`boolean Result=(5> 10|| 23<56);` `Result` *will have the value, true.*

The Logical Operators



Note

A boolean expression is an expression that results in a boolean value, that is, `TRUE` *or* `FALSE`*. For example, the boolean expression* `5>3` *will evaluate to* `TRUE`*.*

Using the Unary Operators

An operator that requires one operand is called a *unary operator*. The following table describes the unary operators.

Operator	***Operator Name***	***Description***	***Example***
`+`	*Unary Plus operator*	*Indicates a positive value.*	`X = +1;` *Assigns the positive value,* `1`*, to the variable,* `X`*.*
`-`	*Unary Minus operator*	*Indicates a negative value.*	`X = -1;` *Assigns the negative value,* `-1`*, to the variable,* `X`*.*
`++`	*Increment operator*	*Increments the value by 1.*	`X++;` *Increments the value of* `X` *by 1.*
`--`	*Decrement operator*	*Decrements the value by 1.*	`X--;` *Decrements the value of* `X` *by 1.*

`!`	*Logical Complement operator*	*Inverts the value.*	`boolean Y=false;`
			`boolean X=!Y;`
			Assigns the value, true, to the variable, `X`*.*

The Unary Operators

The increment and decrement operators can be applied in the prefix and postfix forms.

Prefix Form

In the prefix form, the operator precedes the operand. In this form, the value is incremented or decremented before it is assigned to the operand.

Consider the following code snippet:

```
n=5;
m=++n;
```

Once the preceding statements are executed, the values of both, `m` and `n`, will be `6`. This is because it will first increment the value of `n` by `1`, and then assign it to the variable, `m`.

Postfix Form

In the postfix form, the operator follows the operand. In this form, the value is incremented or decremented after it has been assigned to the operand.

Consider the following code snippet:

```
n=5;
m=n++;
```

Once the preceding statements are executed, the value of `n` will be `6` and the value of `m` will be `5`. This is because the value of `n` will be first assigned to `m`, and then the value of `n` will be incremented by `1`.

Using the Bitwise Operators

Bitwise operators are used for the manipulation of data at the bit level. The bitwise operators operate on the individual bits of their operands. The operands can be of various data types, such as `int`, `short`, `long`, `char`, and `byte`. When you use a bitwise operator, the operands are first converted into their binary equivalents, and then the bitwise operator operates on the bits. The result in binary form is then converted into its decimal equivalent.

Bitwise operators can be categorized into:

- Bitwise AND
- Bitwise OR
- Bitwise NOT
- Bitwise XOR

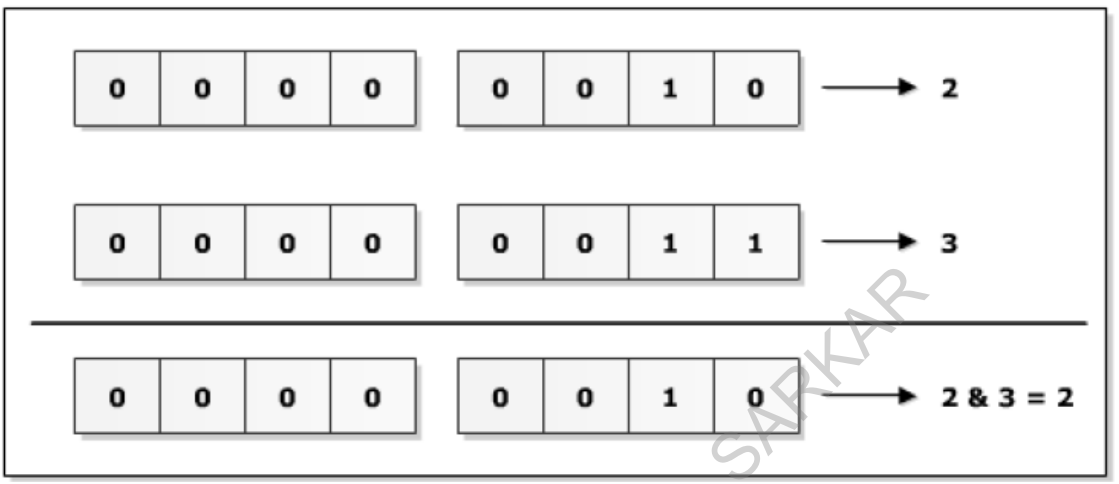
Bitwise AND Operator

The bitwise AND operator (`&`) performs an AND operation on two operands. The AND operator produces `1` if both bits are `1` else `0` in all other cases. The following table lists the bitwise AND operation results that are obtained for the different combination of operands.

Operand 1	***Operand 2***	***Operand 1 & Operand 2***
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

The Bitwise AND Operation Results

The following figure displays the bitwise AND operation for `2` and `3`.



The Bitwise AND Operation for 2 and 3

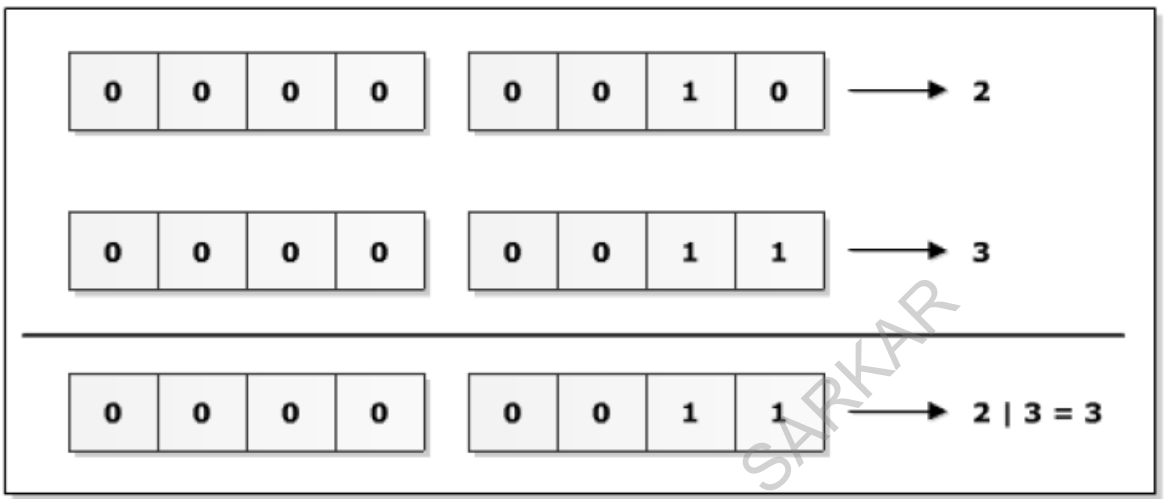
Bitwise OR Operator

The bitwise OR operator (`|`) performs the OR operation on two operands. The OR operator gives the result, `0`, if both bits are `0`, else `1` in all other cases. The following table lists the bitwise OR operation results that are obtained for the different combination of operands.

Operand 1	***Operand 2***	***Operand 1 | Operand 2***
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*1*

The Bitwise OR Operation Results

The following figure displays the bitwise OR operation for `2` and `3`.



The Bitwise OR Operation of 2 and 3

Bitwise NOT Operator

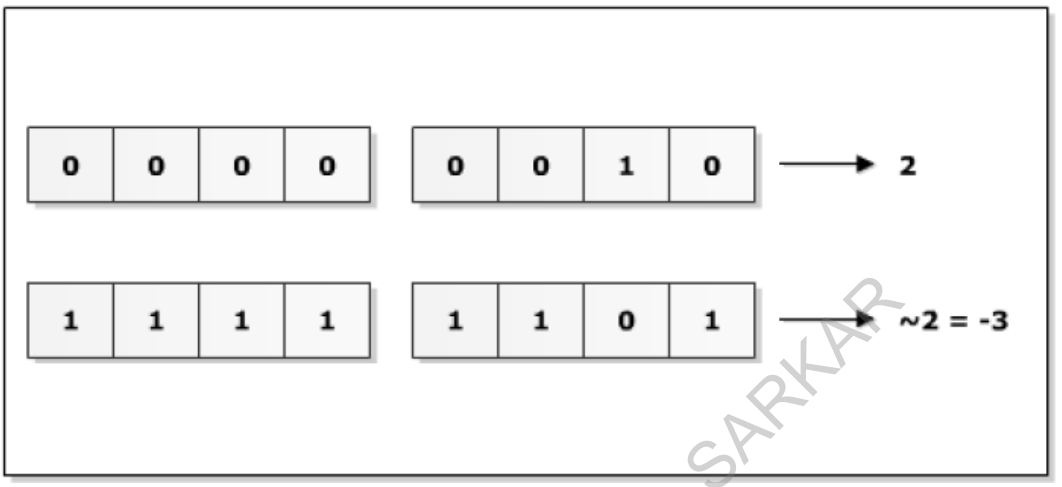
The bitwise NOT operator (~) is a unary operator and performs the NOT operation on each bit of binary number. It is also called Bitwise complement. The NOT operator inverts or complements each of the bits of a binary number.

The following table lists the bitwise NOT operation results that are obtained for different operands.

Operand 1	***~Operand 1***
0	*1*
1	*0*

The Bitwise NOT Operation Results

The following figure displays the bitwise NOT operation for `2`.



The Bitwise NOT Operation for 2

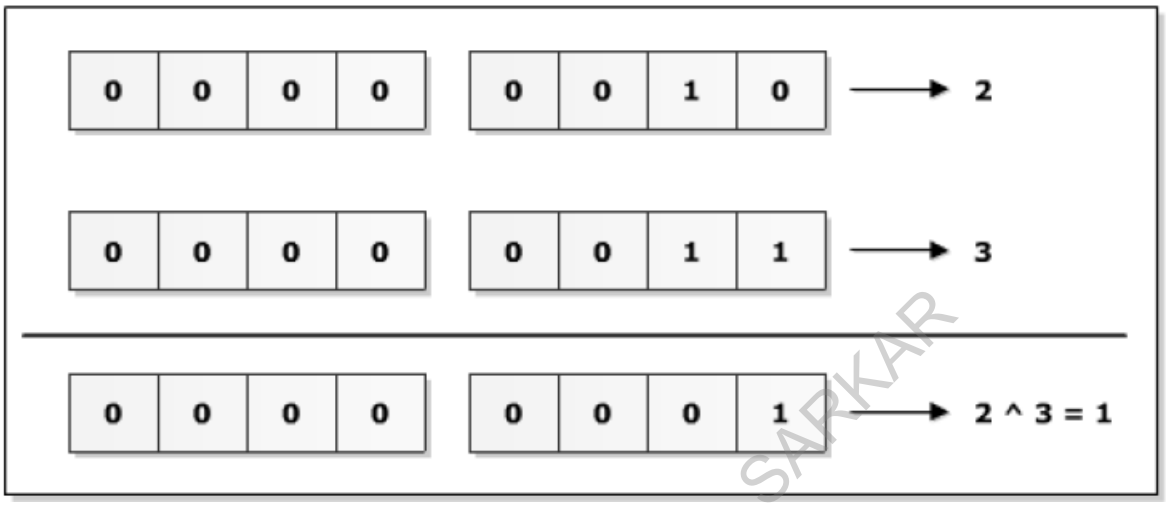
Bitwise XOR Operator

The bitwise XOR (`^`) operator performs the XOR operation on two operands. The XOR operator applied on two bits results in `1`, if exactly one bit is `1`, else `0` in all other cases. The following table lists the bitwise XOR operation results that are obtained for the different combinations of operands.

Operand 1	***Operand 2***	***Operand 1 ^ Operand 2***
0	*0*	*0*
0	*1*	*1*
1	*0*	*0*
1	*1*	*1*

The Bitwise XOR Operation Results

The following figure displays the bitwise XOR operation for `2` and `3`.



The Bitwise XOR Operation for 2 and 3

Using the Shift Operators

A *shift operator* is used to shift the bits of its operand either to the left or to the right. The various types of shift operators are:

- Right shift operator (>>)
- Left shift operator (<<)
- Unsigned right shift operator (>>>)

Right Shift Operator

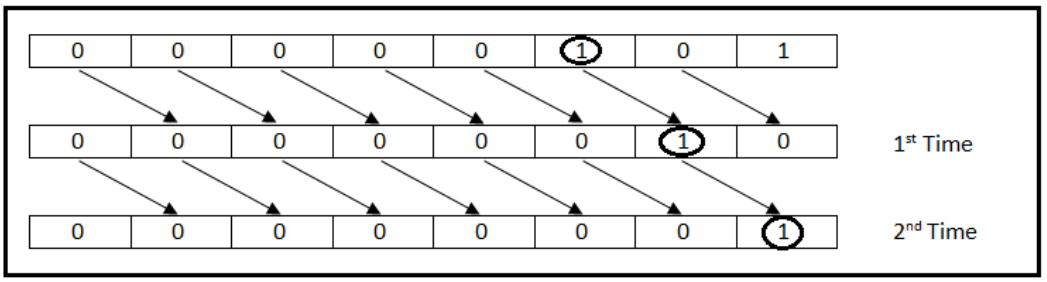
The right shift operator shifts all the bits of a binary number in the right direction. The syntax for the right shift operator is:

```
operand >> num
```

In the preceding syntax, `num` specifies the number of positions to shift the bits in the binary number. For example, in the `5>>2` expression, `2` is the number of times the bits have to be shifted to the right, and `5` is the operand in which the bits are shifted.

The binary representation of `5` in `8` bits is `00000101`. The `>>` operator shifts the bits of this binary number to the right by `2` positions. Therefore, the two rightmost bits are discarded. The result you get is `00000001`, which is the binary representation of `1`. The result is `1` in the decimal representation.

The following figure displays the right shift operation of the `5>>2` expression.



The Right Shift Operation

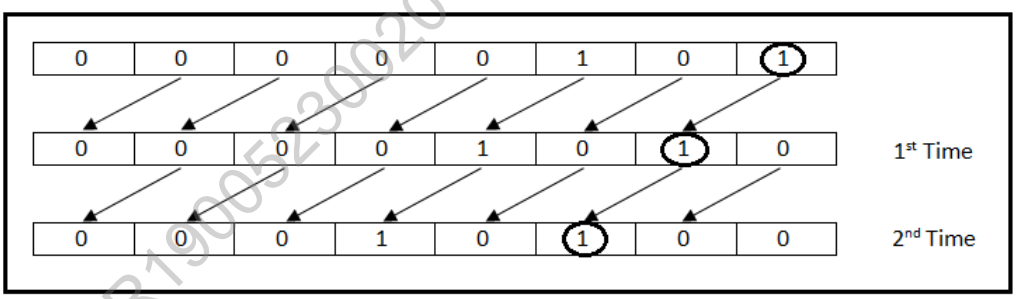
Left Shift Operator

The left shift operator, `<<`, shifts all the bits of a binary number in the left direction. The syntax for the left shift operator is:

```
operand << num
```

In the preceding syntax, `num` specifies the number of positions to shift the bits in binary number. For example, you have the expression, `5<<2`. Here, `2` is the number of times the bits have to be shifted to the left and `5` is the operand. The binary representation of `5` in `8` bits is `00000101`. The `<<` operator shifts the bits of this binary number to the left by `2` positions. Therefore, you get the result, `00010100`, which is binary of `20`.

The following figure displays the left shift operation of the `5<<2` expression.



The Left Shift Operation

Unsigned Right Shift Operator

The unsigned right shift operator (`>>>`) is used to shift the bits of a binary number to the right. The operator fills the leftmost bits of a binary value with `0`, irrespective of whether the number has `0` or `1` at the leftmost bit. The unsigned shift operator is generally used with the `32` and `64` bit binary numbers. For example, you have the number `-2`. The binary representation of `2` in 32-bit is `00000000 00000000 00000000 00000010`. Therefore, the 32-bit binary representation of `-2` is `11111111 11111111 11111111 11111110`.

Similarly, `-2 >>>24` is `00000000 00000000 00000000 11111111`. This is the binary representation of `255`. Therefore, `-2>>>24` is equal to `255`.

Using the Ternary Operator

The *ternary operator* is used to evaluate an expression. The operator works on a logical expression and two operands. It returns one of the two operands depending on the result of the expression.

The syntax of ternary operator is:

```
boolean_expression ? expression 1 : expression 2
```

In the preceding syntax, if `boolean_expression` evaluates to true, `expression 1` is returned, else `expression 2` is returned.

Consider the following code snippet:

```
int a = 1;
int b = 2;
int result;
result = a > b ? a : b;
```

In the preceding code snippet, if the condition is true, the ternary operator assigns the value of the variable, `a`, to the variable, `result`. Otherwise, it assigns the value of the variable, `b` to the variable, `result`.

Just a minute:

Which one of the following operators is used to find the remainder and assign the result to a variable?

1. `%`
2. `%=`
3. `/`
4. `/=`

Answer:

2. `%=`

Activity 2.1: Working with Operators

Using Operator Precedence

Java defines a set of rules for using operators. These rules specify the order in which the expression will be evaluated, when there are multiple operators in an expression. For example, if there are two operators in an expression, such as + and *, then the multiplicative arithmetic operator will be evaluated first as it has higher precedence than the additive arithmetic operator. However, precedence rules can be overridden by using the parentheses.

Identifying the Order of Precedence

Consider the expression, `45+90/5*6`. To compute the result of the expression, each operator in an expression needs to be evaluated in a predetermined order called operator precedence. Operators with a higher precedence are applied before operators with a lower precedence.

The following table lists the operator precedence with their categories sorted from the highest precedence to the lowest precedence.

Operators	***Category***	***Associativity***
`++expr, --expr`	*Unary postfix increment and postfix decrement operator*	*Right to Left*
`expr++, expr--,+, -, !, ~`	*Unary prefix increment, prefix decrement, plus, minus, logical NOT, and bitwise inversion*	*Right to Left*
`*, /, %`	*Multiplicative arithmetic operators*	*Left to Right*
`+, -`	*Additive arithmetic operators*	*Left to Right*
`<<, >>, >>>`	*Shift operators*	*Left to Right*
`>, <, <=, >=, instanceof`	*Comparison operators and the* `instanceof` *operator*	*Left to Right*
`==, !=`	*Comparison equality operators*	*Left to Right*
`&`	*Bitwise AND operator*	*Left to Right*
`^`	*Bitwise XOR operator*	*Left to Right*
`|`	*Bitwise OR operator*	*Left to Right*
`&&`	*Logical AND operator*	*Left to Right*
`||`	*Logical OR operator*	*Left to Right*

Operators	*Category*	*Associativity*
*=, +=, -=, *=, /=, %=*	*Assignment operators and arithmetic assignment operators*	*Right to Left*

The Operator Precedence

Operators on the same line have equal precedence. If an expression has two or more operators of equal precedence, operators are evaluated on the basis of their associativity.

Implementing Precedence Using Parentheses

Parentheses are often used to obtain the desired results as they can override the order of precedence. For example, consider the following code snippet:

```
3 * 5 + 2;
```

In the preceding code snippet, the result of the expression will be `17`, because first `3` will be multiplied by `5`, and then `2` will be added to the result. In this case, precedence of the `*` operator is higher than the `+` operator.

However, you can apply the parenthesis on the preceding code, as shown in the following code snippet:

```
3 * (5 + 2);
```

In the preceding code snippet, the result of the expression will change to `21` as first `2` will be added to `5`, and then result will be multiplied by `3`. In this case, the precedence of parenthesis is higher than the precedence of operator.

Practice Questions

1. Which one of the following operators is used to find the remainder after dividing two numbers?
   a. Plus operator
   b. Minus operator
   c. Modulus operator
   d. Multiply operator

2. Which one of the following operators is used to test whether an object is an instance of a specific class at run time?
   a. Ternary operator
   b. `instanceof` operator
   c. Unary operator
   d. Shift operator

3. Which one of the following operators is used to get output as true, if both expressions are true?
   a. Logical AND
   b. Logical OR
   c. Bitwise NOT
   d. Bitwise OR

4. Which one of the following options denotes the correct output of the expression, `7-2*16/8`?
   a. `6`
   b. `10`
   c. `3`
   d. `4`

5. Which one of the following operators is used to find the remainder after dividing the two numbers?
   a. `+`
   b. `/`
   c. `%`
   d. `&&`

Summary

In this chapter, you learned that:

- Operators allow you to perform various mathematical and logical calculations, such as adding and comparing numbers.
- Arithmetic operators are used to perform arithmetic operations on operands like addition, subtraction, multiplication, and division.
- Assignment operators can be categorized to simple assignment operator and complex assignment operator.
- The simple assignment operator is used to assign a value to the variable.
- Arithmetic operator, when combined with the simple assignment operator are called complex assignment operator. The various complex assignment operators are `+=`, `-=`, `*=`, `/=`, and `%=`.
- Comparison operators are used to compare two values and perform an action on the basis of the result of that comparison. The various relational operators are <, >, <=, >=, ==, and `!=`.
- The `instanceof` operator is used to test whether an object is an instance of a specific class at run time.
- Logical operators are used to evaluate operands and return a boolean value. The various logical operators are `&&` and `||`.
- Unary operator operates on one operand. The various unary operators are unary plus, unary minus, increment operator, decrement operator, and logical complement operator.
- Bitwise operators are used for the manipulation of data at the bit level. The various bitwise operators are bitwise AND (`&`), bitwise OR (`|`), bitwise NOT (~), and bitwise XOR (`^`).
- Shift operators are used to shift the bits of its operand either to the left or to the right. The various shift operators are right shift (>>), left shift (<<), and unsigned shift operator (>>>).
- The ternary operator is used to evaluate an expression. The operator works on a logical expression and two operands. It returns one of the two operands depending on the result of the expression.
- An expression can have multiple operators and each operator is evaluated in order of its precedence.
- Parentheses are often used to obtain the desired results as they can override the order of precedence.

NIIT

Working with Conditional and Loop Constructs

CHAPTER 3

In a Java program, the statements are executed in a sequential order. This order can be changed by using the conditional and looping constructs.

The conditional construct executes the selective block of statements according to the value of the expression. In addition, Java provides the loop construct that makes a set of statements of a program to be repeated a certain number of times. The statement execution continues till the condition set for the loop remains true. When the condition becomes false, the loop terminates and the control moves to the statements following the loop construct.

This chapter discusses the various types of the conditional constructs used in the Java programming language. In addition, it focuses on the different types of loop constructs supported by Java.

Objectives

In this chapter, you will learn to:

- Work with conditional constructs
- Work with loop constructs

Working with Conditional Constructs

In your daily life, you take various decisions that are based on certain conditions. For example, if it is raining, you will take an umbrella. In the same way, you can incorporate the decision making techniques in the Java programming language. The decision making technique can be implemented in the Java programs by using the following conditional constructs:

- The *if* construct
- The *if…else* construct
- The *switch* construct

Using the if Construct

The `if` construct executes statements based on the specified condition. For example, in the Classic Jumble Word game, you needs to display the message, `You are correct !!!!`, if a user identifies the correct word for the corresponding jumbled word that is displayed on the screen. To implement the preceding functionality, you can use the `if` construct. The syntax for the `if` construct is:

```
if(expression)
{
    //statement(s)
}
```

In the `if` construct, `statement(s)` followed by the `if` statement will be executed, when `expression` evaluates to true. However, if `expression` evaluates to false, `statement(s)` will be skipped.

The `if` construct can contain either a single statement or multiple statements. It is not mandatory to enclose the single statement within a pair of braces. However, the multiple statements must be enclosed within a pair of braces.

In the preceding example, you can use the following code snippet:

```
if(jumbleWord.equals(userWord))
{
    System.out.println("You are correct !!!!");
}
```

In the preceding code snippet, the `if` statement will compare values of two string references, `jumbleWord` and `userWord`. The `jumbleWord` reference is used to store the jumbled word and the `userWord` reference is used to store the word entered by the user. If both the objects have same values, it will execute the print statement and will display the message, `You are correct !!!!`, otherwise it will skip the print statement.

Note

The `equals()` method is a method of the `String` class that compares the values of two string objects.

Sometimes, there is a need to check the condition based on another condition. For example, in the preceding scenario of the Classic Jumble Word game, you want to check whether a user enters the word or not, and then want to compare the user's input with the jumbled word.

To achieve the preceding requirement, Java supports the nested `if` construct. The syntax of the nested `if` construct is:

```
if(condition)
{
    if(condition)
    {
            //statement(s)
    }
}
```

In the preceding example, you can use the following code snippet:

```
if(userWord.length()!=0)
{
   if(jumbleWord.equals(userWord))
   {
         System.out.println("You are correct !!!!");
   }
}
```

In the preceding code snippet, the first `if` statement compares the length of the string object stored in the `userWord` reference with `0`. If the length of the string object is not equal to `0`, the second `if` construct gets executed.

The second `if` statement compares the textual values of the `userWord` and `jumbleWord` references. If both the values are same, the message printed will be `You are correct !!!!`.

Note

The `length()` method is a method of the `String` class that returns the length of the string reference variable.

Using the if…else Construct

In the preceding example of the Classic Jumble Word game, you want to modify the existing code to display the message, `You are incorrect !!!!`, if a user enters the incorrect word for the corresponding jumbled word. For this, you can modify the existing code by using the `if…else` construct.

The `if…else` construct executes statements based on the specified condition.

The syntax for the `if…else` construct is:

```
if(expression)
{
    //statement(s)
}
else
{
    //statement(s)
}
```

In the `if…else` construct, `statement(s)` followed by the `if` statement will be executed, when `expression` evaluates to true. However, if `expression` evaluates to false, `statement(s)` followed by the `else` statement will be executed.

Both, the `if` construct and the `else` construct, can contain either a single statement or the multiple statements.

In the preceding example, you can modify the existing code by using the following code snippet:

```
if(jumbleWord.equals(userWord))
{
    System.out.println("You are correct !!!!");
}
else
{
    System.out.println("You are incorrect !!!!");
}
```

In the preceding code snippet, the `if` statement compares the textual values of the `jumbleWord` and `userWord` references. If both the textual values are same, the message printed will be `You are correct !!!!`. Otherwise, the message printed will be `You are incorrect !!!!`.

Nesting of the `if…else` constructs is possible in both, the `if` and `else` blocks. The syntax of the nested `if…else` construct is:

```
if(expression)          //Line 1
{
    if(expression)      //Line 2
    {
          //statement(s)
    }
    else
    {
          //statement(s)
    }
}
else
{
    if(expression)      //Line 3
    {
          //statement(s)
    }
```

```
    else
    {
            //statement(s)
    }
}
```

In the preceding syntax, if line 1 evaluates to true, line 2 gets executed. If line 2 evaluates to true, `statements(s)` within the `if` construct gets executed. Otherwise, `statement(s)` within the `else` construct gets executed. However, if line 1 evaluates to false, the `else` construct gets executed. In the `else` construct, if the line 3 evaluates to true, `statement(s)` within the `if` construct gets executed. Otherwise, `statement(s)` within the `else` construct gets executed.

Using the switch Construct

In the example of Classic Jumble Word game, you want to display the following menu when the game starts:

1. Play Game
2. View Instructions
3. Exit Game

Thereafter, you want a user to enter a choice, such as 1, 2, or 3. On the basis of the user's input, you want the corresponding methods, which implement the functionality for each input, to be invoked. For this, you can use the `switch` construct.

The `switch` construct evaluates an expression for multiple values. The `switch` statement is followed by an expression that tests the value of the expression against a list of values, which can be the integer, character, or string constants. The syntax for the `switch` construct is:

```
switch(expression)
{
    case Expr_1: //statement(s)
                 break;
    case Expr_2: //statement(s)
                 break;
    .
    .
    .
    case Expr_N: //statement(s)
                 break;
    default: //statement(s)
}
```

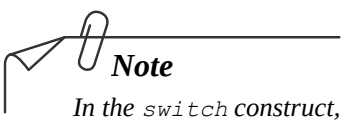
In the `switch` construct, `expression` given in the `switch` statement is compared with each `case` constant. If the `case` constant matches with `expression`, the control is moved to the statement following the matched case constant. Otherwise, the control is moved to the `default` statement.

The *`break`* statement causes the program flow to exit from the `switch` construct. This skips the execution of the remaining case structure by terminating the execution of the `switch` construct.

In the preceding example, you can use the following code snippet:

```
switch(choice)
{
          case 1: playGame();
                  break;
          case 2: instructGame();
                  break;
          case 3: exitGame();
                  break;
          default: System.out.println("Invalid option.");
 }
```

In the preceding code snippet, if the value of `choice` is `1`, the `playGame()` method will be invoked.



Note

In the `switch` construct, the case constants and the value of the `switch` expression must have the same data type.

Just a minute:

Which one of the following values can be accepted by the `switch` construct?

1. *Boolean*
2. *Float*
3. *Double*
4. *Character*

Answer:

4. *Character*

Activity 3.1: Working with Conditional Constructs

Working with Loop Constructs

A looping statement enables you to execute the same statements for a certain number of times. For this, you do not need to write the statements repeatedly. You can enclose the statements within the loop construct and the loop construct executes the statements till the specified condition is met. Java supports the following loop constructs:

- The *for* loop
- The *while* loop
- The *do…while* loop

Using the for Construct

Consider the example of the Classic Jumble Word game, you want to implement a functionality in which a user should guess the correct word in five attempts only. If the user fails to guess the word in five attempts, the game will be terminated. For this, you can use the `for` loop construct. The syntax for the `for` loop construct is:

```
for(initialization; condition; increment/decrement)
{
    //statement(s)
}
```

In the `for` loop construct, the `initialization` statement is executed first. It is executed only once at the beginning of the loop. Thereafter, the `condition` statement is executed for each iteration. If the `condition` statement evaluates to true, `statement(s)` followed by the `for` statement are executed. However, if the `condition` statement evaluates to false, the loop gets terminated. Finally, the `increment/decrement` statement is executed that increments or decrements the loop. The loop continues executing `statement(s)` until the `conditional` statement evaluates to false.

In the preceding example, you can use the following code snippet:

```
for(count=0;count<5;count++)
{
   //statement(s)
}
```

In the preceding code snippet, the `count` variable is initialized with `0`. The loop will continue executing `statement(s)` till the value of the `count` variable is less than `5`.

You can create an infinite loop by keeping all the three statements blank, as displayed in the following code snippet:

```
for ( ; ; )
{
    //statement(s)
}
```

In the preceding code snippet, `statement(s)` will continue executing infinitely.

At times, after certain iterations, you need to exit from the loop. To achieve this, Java provides the `break` statement. The `break` statement stops the execution of the remaining statements within the body of the loop. In addition, Java provides the *`continue`* statement. The `continue` statement skips all the statements following the `continue` statement and moves the control back to the loop statement.

The following code snippet depicts the use of the `break` statement within the `for` loop:

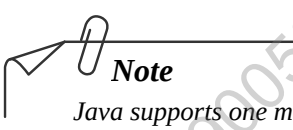
```
for(count=0;count<10;count++)
{
    if(count==7)
    {
            break;
    }
    System.out.println(count);
}
```

In the preceding code snippet, the loop will print the value of `count` from `0` to `6`. Thereafter, the loop will be terminated, when the value of `count` will be `7`.

The following code snippet depicts the use of the `continue` statement within the `for` loop:

```
for(count=0;count<10;count++)
{
    if(count==3)
    {

            continue;
    }
    System.out.println(count);
}
```

In the preceding code snippet, the loop will print the value of `count` from `0` to `9` except `3` because the control will be moved back to the `for` statement, when the value of `count` will be `3`.



Note

Java supports one more type of the `for` loop known as the `for-each` loop. This will be covered in later chapters.

Using the while Construct

In the preceding example, you can implement the similar functionality with the help of the `while` loop construct.

The `while` loop construct provides the similar functionality of the `for` loop construct. However, the syntax for the `while` loop construct is different from the `for` loop construct.

The syntax for the `while` loop construct is:

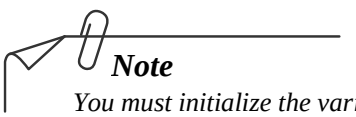
```
while(expression)
{
    //statement(s)}
```

In the `while` loop construct, `statement(s)` followed by the `while` statement will be executed, when the expression evaluates to true. The loop continues executing `statement(s)` until the expression evaluates to false.

In the preceding example, you can use the following code snippet of the `while` loop:

```
int count=0;
while(count<5)
{
    //statement(s)
    count++;
}
```

In the preceding code snippet, the `count` variable is initialized with `0`. The loop will continue executing `statement(s)` till the value of the `count` variable is less than `5`.



Note

You must initialize the variable before using it in the `expression` statement of the `while` loop.

You can also create an infinite loop by using the `while` loop construct, as displayed in the following code snippet:

```
while(true)
{
    //statement(s)
}
```

Using the do…while Construct

Consider the example of the Classic Jumble Word game, you want the menu of the game to be displayed at least once, and then it will be on the user's choice whether the user wants to display the menu again or not. For this, you should use the `do…while` loop construct.

The `do…while` loop construct places the condition at the end of the loop, which makes `statement(s)` to be executed at least once. The syntax for the `do…while` loop construct is:

```
do
{
    //statement(s)

}while(expression);
```

In the `do…while` construct, `statement(s)` are executed at least once. Thereafter, `statement(s)` are executed if the `expression` evaluates to true. The loop continues executing `statement(s)` until the `expression` evaluates to false.

In the preceding example, you can use the following code snippet:

```
do
{
    //statement(s)

}while(choice.equals("Y"));
```

In the preceding code snippet, `statement(s)` gets executed at least once. Thereafter, `statement(s)` gets executed if the textual value of string object is equal to `Y`.

You can also create an infinite loop by using the `do…while` loop construct, as displayed in the following code snippet:

```
do
{
    //statement(s)
}while(true);
```

In the preceding code snippet, `statement(s)` will continue executing infinitely.

Just a minute:

The _____ loop construct contains the condition at the end.

Answer:

do…while

Activity 3.2: Working with Loop Constructs

Practice Questions

1. State whether the following statement is true or false.
   The `if` construct can be used to check multiple conditions.

2. Predict the output of the following code snippet:

```
int num1 = 5, num2 = 5, sum;
sum = num1 + num2;
if(sum>10)
{
    System.out.println("True");
}
else
{
    System.out.println("False");
}
```

3. Which one of the following constructs is the looping construct?
   a. `if…else`
   b. `break`
   c. `switch`
   d. `for`

4. State whether the following statement is true or false.
   The `increment/decrement` expression is evaluated at each iteration of the `for` loop.

5. What will be the output of the following code snippet:

```
int i=10,j=20;
while(i++<--j)
{

}
System.out.println(i+" "+j);
```

Summary

In this chapter, you learned that:

- The decision making technique can be implemented in the Java programs by using the following conditional constructs:
  - The `if` construct
  - The `if…else` construct
  - The `switch` construct
- The `if` construct executes statements based on the specified condition.
- The `if` construct can contain either a single statement or multiple statements.
- Java supports the nested `if` construct.
- The `if…else` construct executes the statements within the `if` block if the expression evaluates to true, otherwise the `else` block gets executed.
- The `if` construct and the `else` construct can contain either a single statement or the multiple statements.
- Nesting of the `if…else` constructs is possible in both, the `if` and `else` blocks.
- The `switch` construct evaluates an expression for multiple values.
- The `switch` statement is followed by an expression that tests the value of the expression against a list of values, which can be the integer, character, or string constants.
- Java supports the following loop constructs:
  - The `for` loop
  - The `while` loop
  - The `do…while` loop
- In the `for` loop construct, the `initialization` statement is executed first.
- Thereafter, the `conditional` statement is executed for each iteration.
- Finally, the `increment/decrement` statement is executed that increments or decrements the loop.
- The loop continues executing `statement(s)` until the `conditional` statement evaluates to false.
- The `break` statement causes the program flow to exit from the construct.
- The `continue` statement skips all the statements following the `continue` statement and moves the control back to the loop statement.
- In the `while` loop construct, `statement(s)` followed by the `while` statement will be executed, when the expression evaluates to true.
- The `do…while` loop construct places the condition at the end of the loop, which makes `statement(s)` to be executed at least once.
- You can create an infinite loop by using `for`, `while`, and `do…while` loop constructs.

NIIT

Working with Arrays, Enums, and Strings

CHAPTER 4

At times, a program needs to use a number of variables. The declaration and initialization of these variables makes the program large and complex. In addition, it is difficult to keep track of so many variables. To overcome this problem, you need a single variable that can store multiple similar values. This can be achieved by using arrays. An array is a group of similar variables that are referred by a common name. Arrays can hold either primitive data type or object references. Arrays offer a convenient means of grouping the related information.

There can be a situation when you need to restrict a user to select from a fixed set of predefined values. For this, you can use the enumerated list, which contains items. To create an enumerated list, Java provides enum. An enum is used to define a fixed set of constants.

Consider another situation where you need to store a sequence of characters, such as employee name and department name. For this, you can use strings. To store the strings, Java provides various built-in classes, such as `String`, `StringBuilder`, and `StringBuffer`. In addition, these classes provide the functionality to perform various operations, such as comparing two strings, concatenating two strings, and changing the case of a string.

This chapter discusses the manipulation of arrays, enums, and strings in Java.

Objectives

In this chapter, you will learn to:

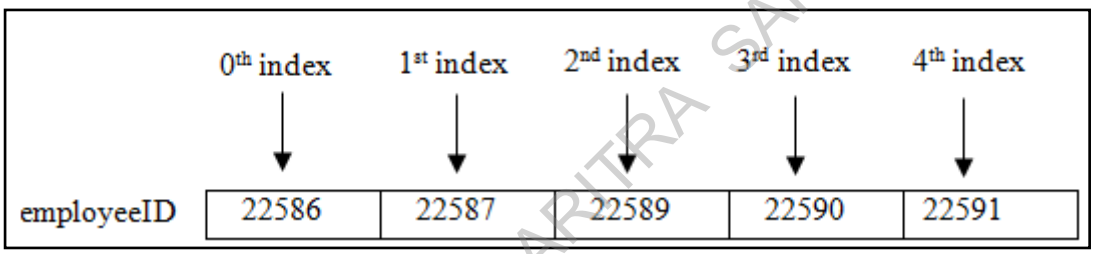
- Manipulate arrays
- Manipulate enums
- Manipulate strings

Manipulating Arrays

In the Classic Jumble Word game, you need to store 100 different words that will be used in the game. Therefore, to store these values, you need to declare 100 variables. This is because a variable can store only one value at a time. However, it is difficult to keep track of 100 variables in a program which makes the program code long and complex. Therefore, in such a situation, you need to declare a variable that can store 100 words. This can be achieved by declaring an array variable.

An *array* is a collection of elements of a single data type stored in adjacent memory locations. You can access an array element by specifying the name and the subscript number of the array. The subscript number specifies the position of an element within the array. It is also called the index of the element. The first element of an array has an index, 0, and the last element has an index one less than the size (number of elements in an array) of the array.

The following figure shows the array of employeeID.



The Array of employeeID

In the preceding figure, you can see that the first element is stored in the index number, 0, while the fifth element is stored in the index number, 4, which is one less than the size of the array.

In order to use arrays, you need to know how to create and access them.

Creating Arrays

You can create the following types of arrays:

- One-dimensional array
- Multidimensional array

One-dimensional Array

A *one-dimensional array* is a collection of elements with a single index value. A one-dimensional array can have multiple columns but only one row.

The creation of a one-dimensional array involves two steps:

1. Declare an array.
2. Assign values to the array.

Declaring an Array

An array needs to be declared before it can be used in a program. You can declare a one-dimensional array by using the following syntax:

```
arraytype arrayname[] = new arraytype[size];
```

In the preceding syntax, `arraytype` specifies the type of element to be stored in array, `arrayname` specifies the name of the array, using which the elements of the array will be initialized and manipulated, and `[size]` specifies the size of the array.

The following code snippet declares an array to store three string values:

```
String jumbledWords[] = new String[3];
```

The preceding code snippet creates an array of String, `jumbledWords`, which can store three elements with the index of elements ranging from `0` to `2`.

Assigning Values to the Array

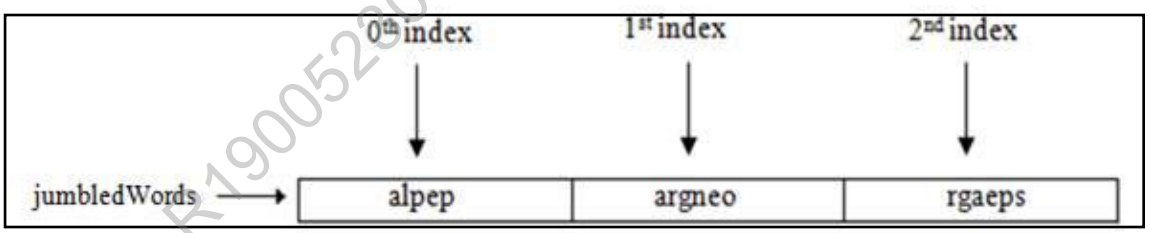
You can assign values to each element of the array by using the index number of the element. For example, to assign the value, `alpep`, to the first element of the array, you can use the following code snippet:

```
jumbledWords[0] = "alpep";
```

You can also assign values to the array at the time of declaration. For this, you are not required to specify the size of the array, as shown in the following code snippet:

```
String jumbledWords[] = {"alpep","argneo","rgaeps"};
```

In the preceding code snippet, `jumbledWords` is a one-dimensional array. The values, `alpep`, `argneo`, and `rgaeps`, are stored in the array, as shown in the following figure.

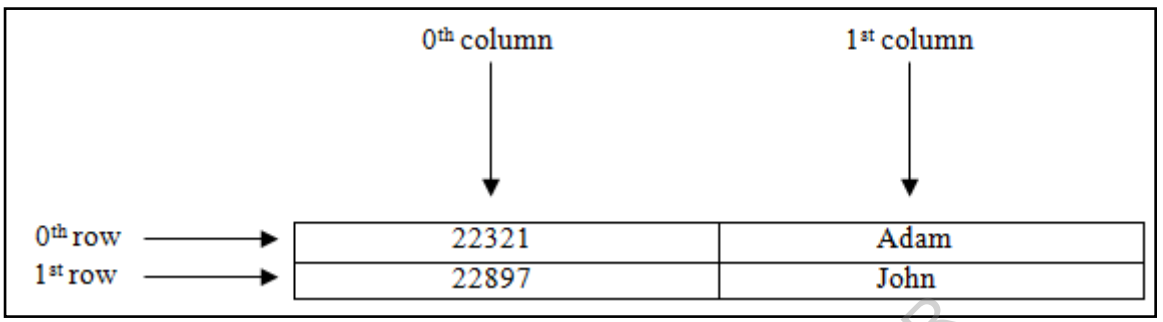


The Values Stored in the jumbledWords Array

Multidimensional Array

Multidimensional arrays are arrays of arrays. An array having more than one dimension is called a multidimensional array. The commonly used multidimensional array is a two-dimensional array where you can have multiple rows and columns. For example, to store an employee name against each employee id, you need to create a two-dimensional array.

The following figure shows a two-dimensional array.



A Two-dimensional Array

The creation of a two-dimensional array involves two steps:

1. Declare an array.
2. Assign values to the array.

Declaring an Array

You can declare a two-dimensional array by using the following syntax:

```
arraytype arrayname[][] = new arraytype[rowsize][columnsize];
```

In the preceding syntax, `arraytype` specifies the type of element to be stored in array, `arrayname` specifies the name of the array, using which the elements of the array will be initialized and manipulated, `rowsize` specifies the number of rows, and `columnsize` specifies the number of columns.

The following code snippet declares a two-dimensional array:

```
String[][] words = new String[4][2];
```

The preceding code snippet creates an array of String, `words`, which contains four rows and two columns.

Assigning Values to the Array

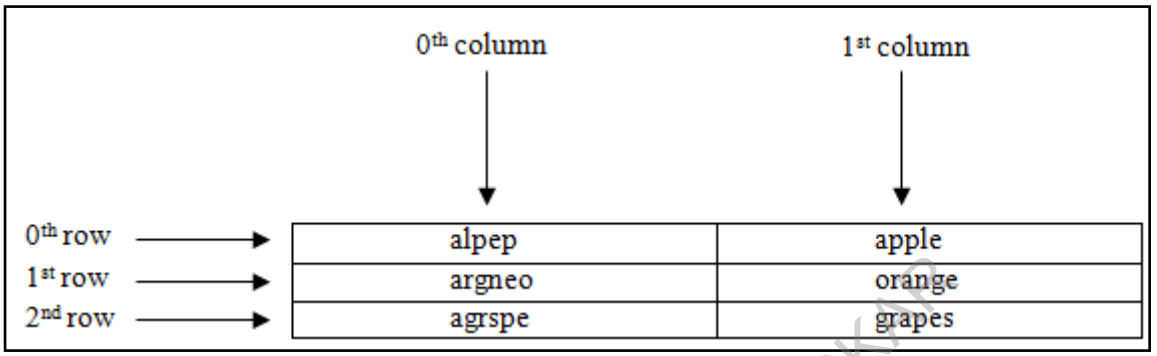
You can assign values to each element of the array by using the index number of the element. For example, to assign the value, `alpep`, to the 0th row and the 0th column and `apple` to the 0th row and the 1st column, you can use the following code snippet:

```
words[0][0] = "alpep";
words[0][1] = "apple";
```

You can also assign values to the array at the time of declaration, as shown in the following code snippet:

```
String[][] jumbledWords = new String[][] {
        {"elapp","apple"},{ "argneo","orange"},{"agrspe","grapes"}};
```

The preceding code snippet stores string values in a two-dimensional array, `jumbledWords`, as shown in the following figure.



The Values Stored in the jumbledWords Array

Just a minute:

Identify the total number of elements, if an array is declared as:

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

1. *3*
2. *4*
3. *5*
4. *6*

Answer:

3. 5

Accessing Arrays

To perform various manipulations on the array, you need to access the following types of arrays:

- One-dimensional array
- Two-dimensional array

One-dimensional Array

To access a one-dimensional array, the following syntax is used:

```
arrayname[index];
```

In the preceding syntax, `arrayname` specifies the name of the array and `index` specifies the location of the array element.

Consider the following code snippet:

```
String jumbledWords[] = {"alpep","argneo","rgaeps"};
System.out.println(jumbledWords[0]);
```

In the preceding code snippet, the `jumbledWords` array stores the various jumbled words. The statement, `System.out.println(jumbledWords[0]);`, accesses the element stored in the first index and displays it. However, if you want to display all the elements stored in the array, you can use the `for` loop, as shown in the following code snippet:

```
String jumbledWords[] = {"alpep","argneo","rgaeps"};
for(int i=0;i<3;i++)
System.out.println(jumbledWords[i]);
```

In the preceding code snippet, the `for` loop is used to traverse through all the elements in the array. For this, you need the index of each array element. Therefore, in the `for` loop, the value of `i` is initialized with `0` and iterated till one less than length of the array. However, if you do not know the total number of elements in the array, then traversing through the entire array will be difficult. This can be simplified by using the `length` property of an array. This property returns the length of an array. The following code snippet is used to traverse through the array using the `for` loop and the `length` property:

```
String jumbledWords[] = {"alpep","argneo","rgaeps"};
for(int i=0;i<jumbledWords.length;i++)
System.out.println(jumbledWords[i]);
```

In the preceding code snippet, the `jumbledwords.length` property returns `3`, which is the length of the array.

While traversing an array using the `for` loop, you need to use a variable. This variable undergoes three operations: initialization, comparison, and increment. This is an error-prone approach, if any one of the operations is not handled properly. Therefore, to avoid the occurrence of errors, Java provides the `for-each` loop to iterate through an array. This loop increases the readability and simplifies the code as you need not take care of the three operations, which are handled by the Java language implicitly. The syntax of the `for-each` loop to use in an array is:

```
for(type var: arrayobject)
```

In the preceding syntax, `type` is the data type or reference type of the `var` variable and `arrayobject` is the name of the array object. The type of the variable should be similar to the type of the data stored in `arrayobject`. The colon (`:`) within the `for-each` loop is read as 'in'. The syntax is read as type `var` in `arrayobject`.

The following code snippet is used to display all the elements stored in the array using the `for-each` loop:

```
String[] jumbledWords = {"alpep","argneo","rgaeps"};
System.out.println("Elements stored in array are: ");
for (String i : jumbledWords)
{
    System.out.println(i);
}
```

In the preceding code snippet, `jumbledWords` is an array object that holds the string object. Every time the `for` loop is executed, each element of the array is assigned to the reference variable, `i`, which further gets displayed.

Two-dimensional Array

To access the two-dimensional array, the following syntax is used:

```
arrayname[row][column];
```

In the preceding syntax, `arrayname` specifies the name of the array, and `row` and `column` specify the location of the array element. Consider the following code snippet:

```
String[][] jumbledWords = new String[][] {
        {"elapp","apple"},{"argneo","orange"},{"agrspe","grapes"}};

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

In the preceding code snippet, the `jumbledWords` array is a two-dimensional array that stores the jumbled word and their corresponding correct word in each row. The `jumbledWords[0][0]` code accesses the element stored at the row index, `0`, and the column index, `0`, and displays it. However, if you want to display all the elements, you can use the `for` loop, as shown in the following code snippet:

```
String[][] jumbledWords = new String[][]{
            {"elapp","apple"},{"argneo","orange"},{"agrspe","grapes"}};
System.out.println("Elements stored in array are: ");
for (int i=0; i<2; i++)
{
    for (int j=0; j<2; j++)
    {
           System.out.print(jumbledWords[i][j]);
    }
}
```

In the preceding code snippet, the `for` loops are used to iterate through the array, `jumbledWords`. The outer `for` loop is used to iterate the array row-wise and the inner `for` loop is used to iterate the array column-wise. The preceding code snippet can be modified to use the `length` property, as shown in the following code snippet:

```
int a[][] = {{1,2},{4,3}};
for(int i=0; i<a.length; i++)
{
for(int j=0; j<a[i].length; j++)
System.out.println(a[i][j]);
}
```

In the preceding code snippet, `a.length` is used to calculate the total number of rows and `a[i].length` is used to calculate the total number of columns in the array.

Further, you can use the following code snippet to display all the elements stored in the two-dimensional array using the `for-each` loop:

```
String[][] jumbledWords = new String{"elapp","apple"},{ "argneo","orange"},{
"agrspe","grapes"}};;

System.out.println("Fruits are: ");
for (String[] i : jumbledWords)
{
    for (String j : i)
    {
        System.out.println(j);
    }
}
```

In the preceding code snippet, the `for-each` loop is used to iterate through the array, `jumbledWords`. The array variable, `i`, is used to store the row data and the variable, `j`, is used to store the column data of each i^{th} row.

Manipulating Enums

In the Classic Jumble Word game, Sam wants the application to restrict a player from choosing the category of words from the given categories, such as fruit, country, or animal. To achieve the preceding task, Sam needs to create a predefined list of values. In Java, a list of predefined values can be created using *enum*.

An enum is a special type of a class, which can have constructors, methods, and instance variables. In order to use enum, you need to know how to declare and access it.

Declaring Enums

You need to declare an enum to define a fixed set of constant. The following syntax is used to declare an enum:

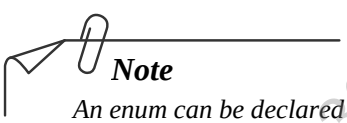
```
enum enum-name{constant 1, constant 2,. . . . ,constant n};
```

In the preceding syntax, `enum-name` is the name of the enum and `constant 1`, `constant 2`, and `constant n` are the constants in the enum.

For example, to create an enumerated list, `Mango`, you need to use the following code snippet:

```
enum Mango{Carrie, Fairchild, Haden};
```

In the preceding code snippet, `Mango` is the name of the enum and `Carrie`, `Fairchild`, and `Haden` are the enum constants, which refer to the variety of the mango.



Note

An enum can be declared either inside the class or outside the class.

In Java, enums are similar to classes. Enums can have constructors, variables, and methods. However, you cannot create an instance of an enum using the `new` keyword. The enum constructors get invoked when the enum constants are created, as these constants are treated as objects. For example, consider the following code snippet:

```
class MangoVarities{
enum Mango{Carrie(10), Fairchild(9), Haden(12);
private int price;
Mango(int p)
{
   price = p;
}
int getPrice()
{
   return price;
}
}
}
```

In the preceding code snippet, the constructor accepts the price of mango and assigns the value to the instance variable, `price`. The `getPrice()` method returns the price of the mango.

Accessing Enums

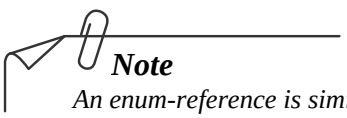
Once have declared an enum, you can access it. You can either use an enum name or enum reference to access enum constants. An enum reference creation is similar to a variable creation. You need to use the following syntax to access an enum:

```
enum-name.enum-constant
```

or

```
enum-refernce.enum-constant
```

In the preceding syntax, `enum-name` is the name of the enum and `enum-constant` is the enum constant. `enum-reference` is the enum reference variable.



Note

An enum-reference is similar to class reference.

Consider the following code snippet to access an enum constant and store it in an enum reference:

```
Mango p = Mango.Carrie;
```

In the preceding code snippet, an enum constant, `Carrie`, is accessed using the enum name, `Mango`. The accessed enum constant is assigned to an enum reference, `p`.

To access all the values stored in an enum, Java provides the `values()` method. This method returns all the enum constants stored in an enum. For example, consider the following code snippet:

```
enum Mango{Carrie, Fairchild, Haden};
class EnumTest{
    public static void main(String args[]){
        for(Mango p:Mango.values())
            System.out.println(p);
    }
}
```

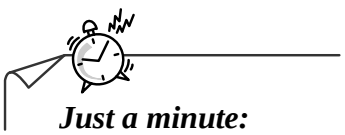
In the preceding code snippet, the `for-each` loop is used to traverse through the enum, `Mango`. The `values()` method is used to return the enum constants stored in the enum.

To access the methods defined in an enum, you can use the following code snippet:

```
enum Mango{Carrie(10), Fairchild(9), Haden(12);
private int price;
Mango(int p)
{
price = p;
}
```

```
int getPrice()
{
return price;
}
}
class EnumTest{
    public static void main(String args[]){
        Mango p = Mango.Carrie;
        System.out.println(p.getPrice());
    }
}
```

In the preceding code snippet, when the statement, `Mango p = Mango.Carrie`, is executed, the enum constructor is invoked and the value, `10`, is assigned to the member variable, `price`. Therefore, when the `getPrice()` method is called, the value, `10`, is returned.



Just a minute:

Which method is used to return all the values stored in an enum?

1. `allValues()`
2. `getValues()`
3. `valueOf()`
4. `values()`

Answer:

4. `values()`

Manipulating Strings

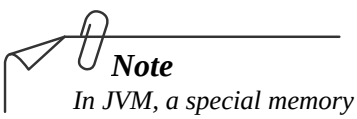
In the Classic Jumble Word game, Sam wants a user to enter the name before playing the game. Thereafter, the name should appear with the message, “Welcome [user name]”. To implement this functionality, there is a need to store the name of the user, and then append the name with the welcome message. Here, the name and welcome message are string values. To manipulate string, Sam can use the classes, such as `String`, `StringBuilder`, and `StringBuffer`, provided by Java. These classes will enable a user to store string literals. In addition, these classes provide various methods for string manipulations.

Using String Class

To store string literals, you can use the `String` class in the `java.lang` package. The following code snippet is used to create a string object:

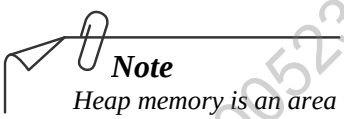
```
String s1 = new String(“Hello”);
```

The preceding code snippet creates a new string object in the heap memory, with a value, `Hello`, and assigns it to reference variable, `s1`. In addition, it creates another string object with the value, `Hello`, in the string constant pool.



Note

In JVM, a special memory named string constant pool is used to store string literals.



Note

Heap memory is an area within the JVM where objects are stored.

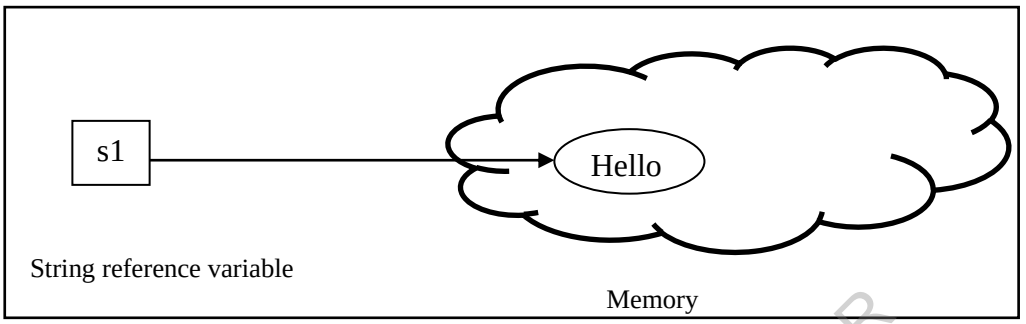
You can also create a string object by using the following code snippet:

```
String s1 = “Hello”;
```

The preceding code snippet creates a new string object with a value, `Hello`, in the string constant pool and assigns it to the reference variable, `s1`.

In Java, `String` class is an immutable class. This means that once a string object is created, you cannot change its value. However, the reference variables of the `String` class are mutable.

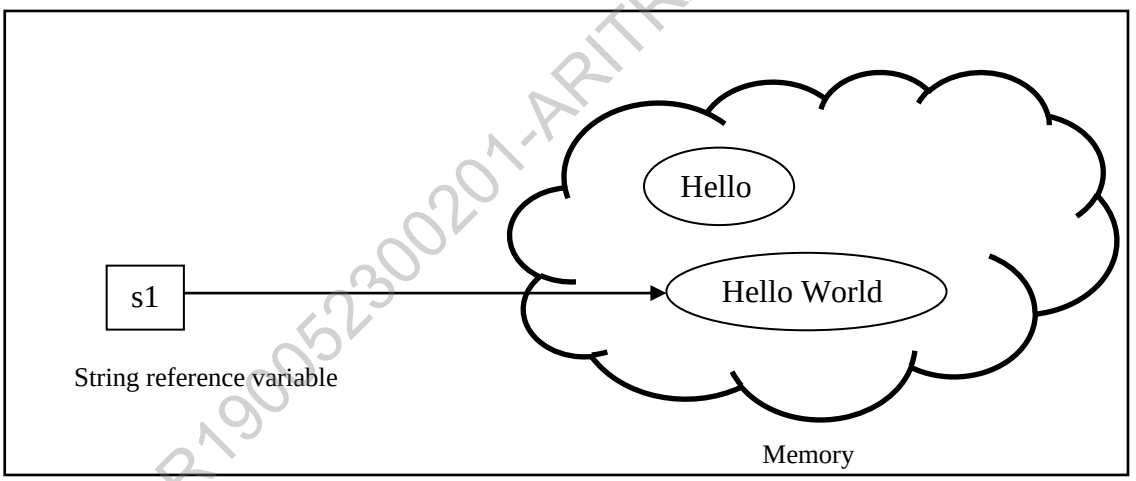
For example, consider the following figure.



The String Reference Variable Referring to the String Object

In the preceding figure, `s1` is a string reference variable, which refers to the string object with the value, `Hello`.

If you append the string literal, `World`, to the string reference variable, `s1`, then a new string object, `Hello World`, is created in the memory and the variable, `s1`, will refer to the new string object. However, the string object, `Hello`, still exists in the memory but has no reference, as shown in the following figure.



The String Reference Variable Referring to the New String Object

Thus, every time you manipulate a string object, a new string object is created in the memory. Therefore the `String` class is called as an immutable class.

The following table lists the some of the most commonly used methods of the `String` class.

Method	***Description***	***Example***

`int length()`	*Returns the length of a string object.*	`String str = "newstring";`
		`int len = str.length();`
		`System.out.println(len);`
		Output:
		`9`

`char charAt(int index)`	*Returns the character at the specified index, which ranges from* `0` *to the length of string object* `-1`*.*	`String str = "Fruit";`
		`char result = str.charAt(3);`
		`System.out.println(result);`
		Output:
		`i`

`void getChars(int srcBegin,int srcEnd, char[] dst, int dstBegin)`	*Copies characters from a source string object into the destination character array. The first character to be copied is at index* `srcBegin`*, the last character to be copied is at index* `srcEnd-1`*. The character is copied into* `dst`*, starting at index,* `dstbegin`*.*	`String Str1 = new String("Welcome to java");`
		`char[] Str2 = new char[6];`
		`Str1.getChars(8, 10, Str2, 0);`
		`System.out.print("Value Copied = ");`
		`System.out.println(Str2);`
		Output:
		`Value Copied = to`

`boolean equals(object obj)`	*Compares the current string object with the other string and returns a boolean value.*	`String str1 = "Fruit";`
		`String str2 = "Fruit";`
		`boolean result = str1.equals(str2);`
		`System.out.println(result);`
		Output:
		`true`

Method	*Description*	*Example*

`int compareTo(String str)`	*Compares the current string object with another string. If the strings are same, the return value is* `0`*, else the return value is non-zero.*	`String str1 = "fruits";`
		`String str2 = "fruits are good";`
		`int result = str1.compareTo( str2 );`
		`System.out.println(result);`
		Output:
		`-9`

`boolean startsWith(String prefix)`	*Tests whether a string starts with the specified prefix or not. If the character sequence represented by the argument is a prefix of the string, the return value is* `true`*. Otherwise, it is* `false`*.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Value returned :");`
		`System.out.println(Str.startsWith( "Welcome"));`
		Output:
		`Value returned :true`

`boolean endsWith(String suffix)`	*Tests if the string object ends with the specified suffix. Returns* `true` *if the character sequence represented by the argument is a suffix of the character sequence represented by this object. Otherwise, it is* `false`*.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Value Returned :");`
		`System.out.println(Str.endsWith("Java"));`
		Output:
		`Value Returned: true`

`int indexOf(int ch)`	*Returns the index of the first occurrence of the specified character within a string. If the character is not found, the method returns* `-1`*.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Index Found at:" );`
		`System.out.println(Str.indexOf( 'o' ));`
		Output:
		`Index Found at:4`

Method	*Description*	*Example*

`int lastIndexOf(int ch)`	*Returns the index of the specified character occurring last in the string. If the character is not found, the method,* `indexOf()`*, returns* `-1`*.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Last Index Found at :" );`
		`System.out.println(Str.lastIndexOf( 'o' ));`
		Output:
		`Last Index Found at :9`

`String subString(int beginindex)`	*Returns a substring of a string. The substring begins with the character at the specified index and extends to the end of the main string.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Value Returned:" );`
		`System.out.println(Str.substring( 10) );`
		Output:
		`Value Returned: Java`

`String concat(String str)`	*Concatenates the specified string to the end of the string object.*	`String str1 = "Hello ";`
		`String str2 = "Everybody";`
		`System.out.println(str1.concat(str2));`
		Output:
		`Hello Everybody`

`String replace(char oldChar,char newChar)`	*Replaces the occurrence of a specified character by a new specified character. Returns the string derived from the string by replacing every occurrence of* `oldChar` *with* `newChar`*.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Value Returned:" );`
		`System.out.println(Str.replace('o', 'T'));`
		Output:
		`Value Returned:WelcTme tT Java`

Method	*Description*	*Example*

`String toUpperCase()`	*Converts the string to uppercase and returns it.*	`String Str = new String("Welcome to Java");`
		`System.out.print("Value Returned:");`
		`System.out.println(Str.toUpperCase());`
		Output:
		`Value Returned: WELCOME TO JAVA`

`String toLowerCase()`	*Converts the string into lowercase and returns it.*	`String Str = new String("WELCOME TO JAVA");`
		`System.out.print("Value Returned:" );`
		`System.out.println(Str.toLowerCase());`
		Output:
		`Value Returned: welcome to java`

`String trim()`	*Removes white space from both ends of a string object and returns the trimmed string.*	`String Str = new String(" WELCOME TO JAVA");`
		`System.out.print("Value Returned :");`
		`System.out.println(Str.trim() );`
		Output:
		`Value Returned :WELCOME TO JAVA`

`char[] toCharArray()`	*Returns a newly allocated array whose length is the length of this string and whose content are initialized to contain the character sequence represented by this string.*	`String Str = new String("WELCOME TO JAVA");`
		`System.out.print("Value Returned: " );`
		`char ch[] = Str.toCharArray();`
		`System.out.println(ch);`
		Output:
		`Value Returned: WELCOME TO JAVA`

Method	*Description*	*Example*

`String valueOf(Object obj)`	*Returns the string representation of the specified argument. Returns* `null` *if the argument is* `null`*. The* `valueOf()` *method is a static method.*	`int a = 10;`
		`Double b = 2.00;`
		`char[] arr = {'a', 'b'};`
		`System.out.println("Return Value : " + String.valueOf(a) );`
		`System.out.println("Return Value : " + String.valueOf(b) );`
		Output:
		`Return Value : 10`
		`Return Value : 2.0`

`boolean equalsIgnoreCase(String anotherString)`	*Compares this string with another string and ignores case considerations.*	`String Str1 = new String("fruits");`
		`String Str2 = new String("FRUITS");`
		`System.out.println("Return = "+ Str1.equalsIgnoreCase(Str2));`
		Output:
		`Return = true`

The String Class Methods

Using StringBuilder and StringBuffer Classes

You can also use the `StringBuilder` and `StringBuffer` classes to work with strings. These classes are mutable classes as they do not create any new string object when manipulated. Therefore, when you need to do various manipulations, such as appending, concatenating, and deleting with string literals, you should use `StringBuilder` and `StringBuffer`.

The following code snippet initializes a string object to a `StringBuilder` reference:

```
StringBuilder s1= new StringBuilder("Hello");
```

The following table lists the some of the most commonly used methods of the `StringBuilder` class.

Method	***Description***	***Example***

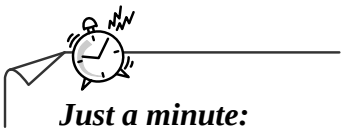
`StringBuilder append(String obj)`	*Appends the argument to the string builder.*	`StringBuilder sb = new StringBuilder("Fruits ");`
		`sb.append("are good for health");`
		`System.out.println(sb);`
		Output:
		`Fruits are good for health`

`StringBuilder delete(int start, int end)`	*Deletes the sequence from start to end in the char sequence.*	`StringBuilder str = new StringBuilder("fruits are very good");`
		`str.delete(10, 15);`
		`System.out.println("After deletion = " + str);`
		Output:
		`After deletion = fruits are good`

`StringBuilder insert(int offset, String obj)`	*Inserts the second argument into the string builder. The first argument indicates the index before which the data is to be inserted.*	`StringBuilder str = new StringBuilder("fruitsgood");`
		`str.insert(6, " are ");`
		`System.out.print("After insertion = ");`
		`System.out.println(str.toString());`
		Output:
		`After insertion = fruits are good`

`StringBuilder reverse()`	*The sequence of characters in the string builder is reversed.*	`StringBuilder str = new StringBuilder("fruits");`
		`System.out.println("reverse = " + str.reverse());`
		Output:
		`reverse = stiurf`

The StringBuilder Class Methods



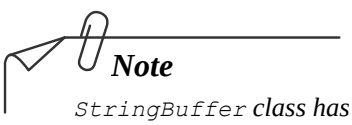
Just a minute:

Which one of the following `String` *class methods is used to copy characters from a source string object into the destination character array?*

1. `charAt()`
2. `getChars()`
3. `toCharArray()`
4. `substring()`

Answer:

2. `getChars()`



Note

`StringBuffer` *class has the same methods as that of the* `StringBuilder` *class. However, the methods in* `StringBuffer` *class are synchronized.*

Activity 4.1: Manipulating Arrays and Strings

Practice Questions

1. Which one of the following options is the correct declaration of an array?

   a. `int [10] list;`
   b. `int list[10];`
   c. `int list = new int[10];`
   d. `int[ ] list = new int[10];`

2. Consider the following code snippet:

```
int Number1 = 0;
int Number2 = 0;
int[] Array1 = new int [] {2,3,4,5,6,7,8,9,10,11};
for (int Ctr : Array1)
{
    if (Ctr%2 == 1)
    {
        Number1++;
    }
        else
        {
            Number2++;
            System.out.print(Ctr);
        }
    }
```

   What will be the output of the preceding code snippet?

   a. `3,5,7,9,11`
   b. `2,4,6,8,10`
   c. `2,5,7,9,10,11`
   d. `3,4,7,9,10,11`

3. Which one of the following `String` class methods returns the output as `true` if both strings are equal?

   a. `equals()`
   b. `compareTo()`
   c. `append()`
   d. `subString()`

4. Which one of the following `String` class methods is used to convert the calling `String` object to a new character array?

   a. `trim()`
   b. `toCharArray()`
   c. `valueOf()`
   d. `getChars()`

Summary

In this chapter, you learned that:

- An array is a collection of elements of a single data type stored in adjacent memory locations. You can access an array element by specifying the name and the subscript number of the array.
- The subscript number specifies the position of an element within the array. It is also called the index of the element.
- The various types of array are one-dimensional array and multidimensional array.
- A one-dimensional array is a collection of elements with a single index value. A one-dimensional array can have multiple columns but only one row.
- Multidimensional arrays are arrays of arrays. A multidimensional array can have multiple columns and rows.
- To display all the elements stored in the array, you can use the `for` loop.
- To know the total number of elements in the array, you can use the `length` property.
- Java provides the `for-each` loop to iterate through an array.
- In Java, a list of predefined values can be created using enum.
- An enum is a special type of a class, which can have constructors, methods, and instance variables.
- An enum can be declared either inside the class or outside the class.
- In Java, enums are similar to classes.
- To access enum constants, you can either use an enum name or enum reference.
- To store string literals, you can use the `String` class in the `java.lang` package.
- In Java, `String` class is an immutable class. This means that once a string object is created, you cannot change its value.
- Every time you manipulate a string object, a new string object is created in the memory.
- `StringBuilder` and `StringBuffer` classes are mutable classes as they do not create any new string object when manipulated.

NIIT

Implementing Inheritance and Polymorphism

CHAPTER 5

In Java, classes can be related to each other and reused in various scenarios within an application. The process of creating a new class by acquiring some features from an existing class is known as *inheritance*.

In addition, *polymorphism* is the ability to redefine a function in more than one form. In OOPs, you can implement polymorphism by creating more than one function of the same name within a class. The difference between the functions lies in the number, types, or sequence of parameters passed to each function.

In this chapter, you will learn how to implement inheritance and polymorphism in Java.

Objectives

In this chapter, you will learn to:

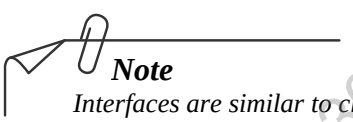
- Implement inheritance
- Implement polymorphism

Implementing Inheritance

In an object oriented programming language, such as Java, you can reuse or extend the functionalities and capabilities of an existing class in a new class if both the classes have similarities amongst themselves. A class can inherit the features of a related class and add new features, as per the requirement. This allows the sharing and reuse of code and helps in making the code of an application more organized and efficient by reducing redundancy.

Consider the scenario of the Classic Jumble Word game. Ricky, the Project Manager, has asked Sam to implement different levels to the game. These levels need to be based upon the maximum number of attempts that can be availed by a player. Further, with each subsequent level, more sets of functionality will be added to the game, such as a timer and score. The timer will specify a time limit within which the player will be required to identify the correct word. As the player will play the game, the value of the timer will be decremented. If the player is unable to identify the correct word in the time specified by the timer, the game will terminate. Furthermore, with the change in the complexity level of the game, there will be a variation in how the player scores are computed.

In order to enhance the game functionality, Sam can recode the entire application. However, this will introduce redundancy in the code and consume a large amount of time. As some of the required sets of functionality are common to all levels, such as displaying the menu and instructions, Sam should reuse the existing code. Thereafter, he should add the new desired functionality to the existing functionalities. For this, Sam decides to implement the inheritance feature of Java. In Java, inheritance can be implemented through classes and interfaces. In inheritance, the class that inherits the data members and methods from another class is known as the *subclass* or derived class. The class from which the subclass inherits the features is known as the *superclass* or base class.



Note

Interfaces are similar to classes, but they contain only constant variables and abstract methods.

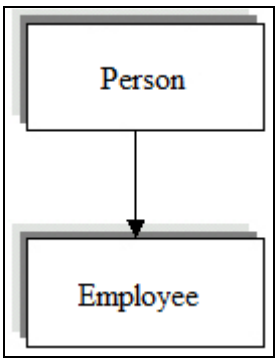
Identifying the Various Types of Inheritance

Java supports the following types of inheritance:

- Single level inheritance
- Multilevel inheritance
- Hierarchical inheritance

Single Level Inheritance

In single level inheritance, a single subclass derives the functionality of an existing superclass. The following figure explains the concept of single level inheritance.

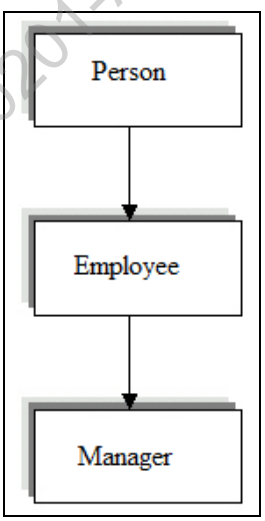


The Concept of Single Level Inheritance

In the preceding figure, there are two classes, `Person` and `Employee`. The class, `Employee`, is the subclass that is inherited from the superclass, `Person`.

Multilevel Inheritance

In multilevel inheritance, a subclass inherits the properties of another subclass. The following figure explains the concept of multilevel inheritance.

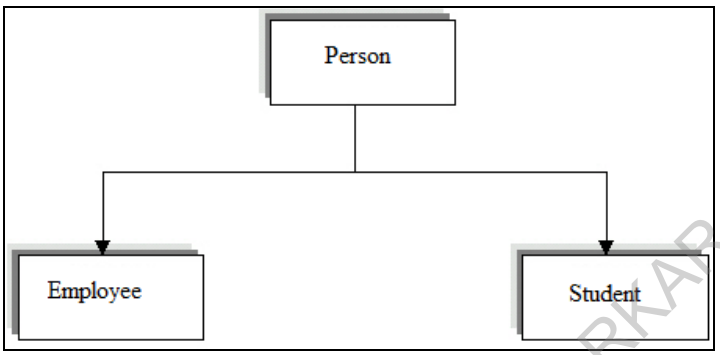


The Concept of Multilevel Inheritance

In the preceding figure, the class, `Person`, is a superclass for the class, `Employee`, and the class, `Employee`, is a superclass of the class, `Manager`. You can include any number of levels in multilevel inheritance.

Hierarchical Inheritance

In hierarchical inheritance, one or more subclasses are derived from a single superclass. The following figure shows the concept of hierarchical inheritance.



The Concept of Hierarchical Inheritance

In the preceding figure, the subclasses, `Employee` and `Student`, inherit the properties of a single superclass, `Person`.

Inheriting a Class

Inheritance signifies the relationship between a superclass and its subclass. In order to inherit a class, you need to use the `extends` keyword. For example, if you have a superclass named `Books` and a subclass named `PaperBooks` that share common functionalities and properties with the superclass, such as printing the book details and author name, total number of pages, and price. In order to implement single level inheritance for the `Books` class, you can use the following code:

```
class Books
{
    int page_num;
    String authorname, name;
    float price;

    public Books()
    {
        page_num = 50;
        authorname = "Andrew Jones";
        name = "The Living Ideas";
        price = 15.78f;
    }

    public void displayInfo()
    {
        System.out.println("The name of the book is " + name);
        System.out.println("The price of the book is " + price);
        System.out.println("The author name is " + authorname);
        System.out.println("The total number of pages is " + page_num);
```

```
    }
}

class PaperBooks extends Books
{

    int shippingcharges = 10;

    public void printInfo()
    {
        displayInfo();//Calling the method of Book class.
        System.out.println("The total shipping charges are" +
shippingcharges);
    }

    public static void main(String[] args)
    {

        PaperBooks pb = new PaperBooks();
        pb.printInfo();

    }
}
```

In the preceding code, `PaperBooks` is a subclass of the superclass class, `Books`. The subclass uses the various properties, such as author name and price. In addition, it uses the `displayInfo()` method. The output of the preceding code is:

```
The name of the book is The Living Ideas
The price of the book is 15.78
The author name is Andrew Jones
The total number of pages is 50
The total shipping charges are $10
```

Inheritance can also be implemented by using abstract classes. An *abstract class* is a class that contains one or more abstract methods. An abstract class cannot be instantiated but can be inherited by other classes by using the `extends` keyword. When an abstract class is inherited by a subclass, the subclass must provide the implementation of all the abstract methods defined in the abstract class. The abstract class acts as a blueprint for other classes. You can define an abstract class by using the following syntax:

```
<access_specifier> abstract class <abstract_class_name>
{
//variables
// abstract and concrete methods
}
```

In the preceding syntax, `<access_specifier>` specifies the access specifiers, such as `public` and `private`. `abstract` and `class` are keywords. `<abstract_class_name>` specifies the name of the abstract class.

You can declare an abstract method by using the following syntax:

```
abstract return-type methodname(parameter-list);
```

In the preceding syntax, `abstract` is a keyword. `return-type` specifies the type of value returned by the method. `methodname` specifies the name of the method and `parameter-list` specifies the list of the parameters that can be accepted by the method.

Consider the scenario of a game console. Each of the games in the game console must offer functionalities to play the game, compute the score, display the score, and exit the game. However, some of these functionalities, such as play the game and compute the score, will vary for every game. However, some of the functionalities, such as display the score and exit the game, will remain the same for all the games. For example, the Badminton game will have a different mechanism to compute the score in comparison to the TableTennis game. However, both the games will display the score in a similar manner. Therefore, for the game console, you can create an abstract class named `GameConsole`. This class will contain concrete methods for the functionalities that are common to all the games, such as `displayScore()`. The functionalities that need to be implemented differently by every game, such as `computeScore()`, will be declared as abstract methods. Therefore, the game classes, such as `Badminton`, which will inherit the `GameConsole` class, will provide the implementation of these abstract methods.

Consider the following code:

```
abstract class GameConsole
{
    int score;
    void displayScore()
    {
        System.out.println("The displayScore method.");
    }
    abstract void computeScore();
    abstract void playGame();
}

class Badminton extends GameConsole
{
    void playGame()
    {
        System.out.println("Starting the Badminton Game...");
    }
    void computeScore()
    {
        System.out.println("Implementing the abstract method of the
Gameconsole class.");
    }
}

class GameDemo
{
    public static void main(String args[])
    {
        Badminton obj1 = new Badminton();
        obj1.playGame();
        obj1.computeScore();
        obj1.displayScore();

    }
}
```

In the preceding code, the `GameConsole` class is an abstract class that declares the abstract methods, `playGame()` and `computeScore()`, which will be used to start the game and compute the score. The `Badminton` class inherits the `GameConsole` abstract class and provides its specific implementation for the `computeScore()` and `playGame()` abstract methods. Further, in the `main()` method, an object, `obj1`, of the `Badminton` class is created and the methods are invoked.

Sometimes, within a method or a constructor, there may be a need to refer to the object that has invoked it. Moreover, in your program, sometimes you may have a requirement where the class variables and the method or constructor variables have the same name. In order to cater to the preceding requirements, Java provides the `this` keyword. In addition, it helps to differentiate between the member and the local variables of a method or constructor in case the member and the local variables of a method have the same name. For example, consider the following code:

```
class Vehicle
{
    int max_speed = 210;
    Vehicle(int max_speed)
    {
        max_speed = max_speed;
    }
    public void showmaxspeed()
    {
        System.out.println("The top speed is " + max_speed);
    }
}
class MainClass {
    public static void main(String args[])
    {
        Vehicle a = new Vehicle(250);
        a.showmaxspeed();
    }
}
```

In the preceding code, the member variable and the parameter for the constructor of `Vehicle` share the same name, `max_speed`. In such a scenario, when the object, `a`, is created, the value, `250`, is assigned to it. However, when the `showmaxspeed()` method is invoked, the value of the class variable, `max_speed`, is displayed as `210`. This is because the member variable is hidden by the parameter of the constructor. To overcome this problem, you can access the member variable by using the following code snippet:

```
Vehicle (int max_speed)
{
 this.max_speed = max_speed;
}
```

In addition to `this` keyword, Java provides the `super` keyword to refer to the member variables and methods of the superclass in a subclass. For example, consider the following code:

```
class Vehicle
{
    int max_speed = 210;
    public void showmaxspeed()
    {
        System.out.println("The top speed is " + max_speed);
```

```
    }
}
class Car extends Vehicle
{
    int max_speed = 180;
    public void showmaxspeed_vehicle()
    {
        System.out.println("The top speed of CAR is " + max_speed);
        System.out.println("The top speed of VEHICLE is " + super.max_speed);
    }
}
class MainClass
{
    public static void main(String args[])
    {
        Vehicle a = new Vehicle();
        Car b = new Car();
        b.showmaxspeed_vehicle();
    }
}
```

In the preceding code, both, `Vehicle` and `Car`, have a member variable named `max_speed`. In order to access the value of the `max_speed` variable of the `Vehicle` class in the `Car` class, the `super` keyword is used.

Note

The `super` keyword followed by `()` can also be used to invoke the constructor of the superclass from the constructor of a subclass. However, in such a scenario, `super();` should be the first statement.

Inheriting an Interface

Consider a scenario of the Classic Jumble Word game where each of the games in the console can be played at various levels. With each subsequent level, the game will become more challenging for a player. However, for each game and level, it is necessary to offer some common functionality, such as taking input from the player and calculating the score. However, the implementation will vary as per the level at which the game will be played. For example, while playing the game at the first level, the player has no time restrictions and no score is displayed. At the second level, there will be a specific number of attempts to identify the correct word. At this level, the score will depend upon the number of attempts in which the player identifies the jumbled word correctly.

Furthermore, at the next level, there will be a restriction regarding the time in which the player has to guess a jumbled word correctly. In this case, the score will depend upon how quickly the player gives the correct answer.

Therefore, at all these levels, the implementation of how the score is calculated varies. In such a scenario, you can create a blueprint that outlines the necessary functionalities to be implemented at all the levels but does not specify their actual implementation. The classes that use the blueprint must provide the implementation of the functionalities listed in the blueprint. In Java, such a blueprint is known as an

interface. The interface will serve as a service contract that all the classes that are implementing the interface will have to follow. If the classes do not follow the contract, the code will not compile.

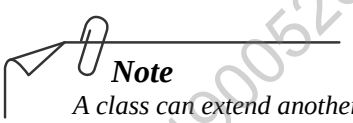
Interfaces contain a set of abstract methods and static data members. By default, all the methods specified in an interface are `public` and `abstract`, and all the variables are `public`, `static`, and `final`. You can define an interface by using the following syntax:

```
interface <interfacename>
{
//interface body
static final <data type> <variablename>;
public return-type methodname(parameter-list);
}
```

In the preceding syntax, `interface`, `static`, `final`, and `public` are keywords. `<interfacename>` specifies the name of the interface. `return-type` specifies the type of value returned by the method. `methodname` specifies the name of the method and `parameter-list` specifies the list of the parameters that can be accepted by the method. `<data type>` and `<variablename>` specify the type of data and the name of the variable, respectively.

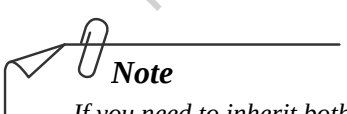
You can implement an interface in one or more class. A class that implements an interface must provide the implementation of all the methods declared in that interface. You can implement an interface in a class by using the following syntax:

```
class <class_name> implements <interfacename>
{
    //Defining the method declared in the interface.

    public return-type methodname(parameter-list)
    {
            //Body of the method
    }
}
```



Note

A class can extend another class and implement an interface at the same time.



Note

If you need to inherit both, a class and an interface, in another class, then, in the class definition, the `extends` keyword must precede the `implements` keyword. For example, `class TicTacToe extends Game implements GameInterface`.

In the preceding syntax, `class`, `public`, and `implements` are keywords. `return-type` specifies the type of value returned by the method. `methodname` specifies the name of the method and `parameter-list` specifies the list of the parameters that can be accepted by the method.

While working with interfaces, the following points should be considered:

- An interface cannot be instantiated.
- An interface cannot have a constructor.
- An interface can itself extend multiple interfaces.
- An interface contains only abstract methods and `static` and `final` variables.

Consider the scenario of the Classic Jumble Word game. You can create an interface named `Games` to declare the common functionalities for each of the games in the console, as shown in the following code snippet:

```
public interface Games
{
    String msg = "Welcome ";
    public void score();
}
class ClassicJumble implements Games
{
    void gamelist()
    {
        // Display the main list of games
    }

    void userinput()
    {
        //Accept player input
    }
    public void score()
    {

       /* Some code to be added */
    }
       /* Some code to be added */
}
```

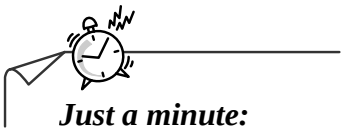
In the preceding code snippet, the interface, `Games`, declares a final variable, `msg`, and abstract method, `score()`. The `ClassicJumble` class implements the interface and provides the implementation of all the abstract methods declared in the interface, `Games`.

Further, an interface can also extend an interface, by using the following syntax:

```
interface <interface_name> extends <interface1, interface2,…interfaceN>
```

In the preceding syntax, an interface is extending multiple interfaces by using the `extends` keyword.

When an interface extends another interface, the class that is implementing the interfaces must define all the methods that are declared in both the interfaces.



Just a minute:

Which one of the following keywords is used to represent an object of the current class?

1. `super`
2. `this`
3. `interface`
4. `abstract`

Answer:

2. `this`

Implementing Polymorphism

Polymorphism is an OOP feature that enables an entity to exist in multiple forms. In Java, polymorphism has the following two types:

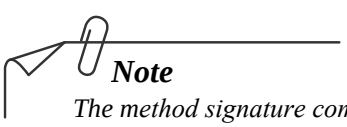
- Static polymorphism
- Dynamic polymorphism

Static Polymorphism

In case of static polymorphism, an entity, such as a method, can exist in multiple forms. This means that one or more methods can exist with the same name but with a different argument list. This type of polymorphism, when exhibited by methods, is known as method overloading. For example, a method named `calculate` can be overloaded by using the following code snippet:

```
calculate( int x, int y)
{
/* Some code to be added */
}

calculate (float x, int y, int z)
{
/* Some code to be added */
}

calculate (int x, float y)
{
/* Some code to be added */
}

calculate (float y, int x)
{
/* Some code to be added */
}
```


In the preceding code snippet, the `calculate()` method implements polymorphism by existing in multiple forms. Each of the forms of the `calculate()` method differs from the other in terms of method signature. During compilation, the compiler will bind the function call. For example, `compute (0.75,8,10)` with the `compute (float x, int y, int z)` method. Whereas, the compiler will bind the `compute (81,102)`, statement to the `compute(int x, int y)` method.

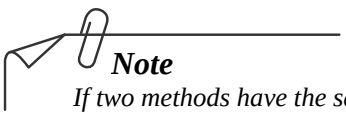


Note

The method signature comprises a method's name and parameter list.

While implementing method overloading, it is important to consider the following points about overloaded methods:

- They differ in the type and/or number of their arguments.
- They differ in the sequence of their parameters.
- They differ in the data types of their parameters.



Note

If two methods have the same signature but different return types, the methods are not overloaded.

Dynamic Polymorphism

In Java, if a superclass and subclass contain methods with the same name, the version to be invoked will be decided by the JVM at runtime. Such a decision to invoke the appropriate method is known as dynamic polymorphism. Dynamic polymorphism is implemented in Java by *method overriding*. Method overriding enables a subclass to provide its own implementation of a method that already has an implementation defined in its superclass. To override a method present in the superclass, the subclass method should have the same name, same parameters, and same return type as the method in the superclass.

Consider the following code:

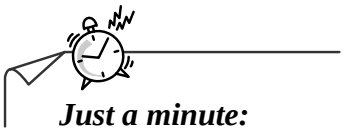
```
class Person
{
    public void showDetails()
    {
        System.out.println("In the Person class");
    }
}
class Employee extends Person
{
    public void showDetails()
    {
        System.out.println("In the Employee class");
    }
}
class Student extends Person
{
    public void showDetails()
    {
        System.out.println("In the Student class");
    }
}
class Method_Override
{
    public static void main(String args)
    {
        Person P = new Person();
```

```
        Employee E = new Employee();
        Student S = new Student();
        Person ref;
        ref = P;
        ref.showDetails(); // calls the showDetails method of the Person
class
        ref = E;
        ref.showDetails(); // calls the showDetails method of the Employee
class
        ref = S;
        ref.showDetails();// calls the showDetails method of the Student
class
    }
}
```

In the preceding code, the `Student` and `Employee` classes extend the `Person` class. The `Person` and `Employee` classes override the `showDetails()` method of the `Person` class. Further, in the `main()` method, the objects of the `Student`, `Employee`, and `Person` classes are created. Also, a reference, `ref`, has been created for the `Person` class. The reference is used to further invoke the `showDetails()` method. The invoked method depends upon the object being referred to by the reference at runtime.

It is important to consider the following points while implementing overriding:

- Private methods cannot be overridden, as they are not accessible in subclasses.
- Final methods cannot be overridden.
- An overridden method cannot be granted more restrictive access rights in a subclass than it is assigned in case of a superclass. For example, if a method is specified with the `public` access specifier in a superclass, it cannot be specified as `protected` in a subclass.



Just a minute:

What will be the output of the following code?

```
public class MethodDemo
{
    void print()
    {
       System.out.println("Print1");
    }
    void print(String a)
    {
       System.out.println("Print2");
    }
    String print()
    {
       System.out.println("Print3");
       return "Print3";
    }
public static void main(String args[])
  {
      MethodDemo obj=new MethodDemo();
      obj.print();
  }
}
```

1. `Print1`
2. *Compile-time error*
3. `Print2`
4. `Print3`

Answer:

2. *Compile-time error*

Activity 5.1: Implementing Inheritance and Polymorphism

Practice Questions

1. Which one of the following classes is used to define concrete methods and declare methods without any implementation?
   a. Subclass
   b. Superclass
   c. Abstract class
   d. Final class

2. Which one of the following keywords is used to implement an interface?
   a. `implements`
   b. `extends`
   c. `super`
   d. `final`

3. Which one of the following types of inheritance allows a subclass to inherit from another subclass?
   a. Multiple inheritance
   b. Multilevel inheritance
   c. Single level inheritance
   d. Multipath inheritance

4. In which one of the following options, one or more subclasses are derived from a single superclass?
   a. Single level inheritance
   b. Instantiation
   c. Multiple inheritance
   d. Hierarchical inheritance

5. Which one of the following options allows a method to exist in multiple forms?
   a. Polymorphism
   b. Encapsulation
   c. Inheritance
   d. Abstraction

Summary

In this chapter, you learned that:

- A class can inherit the features of a related class and add new features, as per the requirement.
- In inheritance, the class that inherits the data members and methods from another class is known as the subclass or derived class.
- The class from which the subclass inherits the features is known as the superclass or base class.
- In single level inheritance, a single subclass derives the functionality of an existing superclass.
- In multilevel inheritance, a subclass inherits the properties of another subclass.
- In hierarchical inheritance, one or more subclasses are derived from a single superclass.
- An abstract class is a class that contains one or more abstract methods.
- An abstract class cannot be instantiated but can be inherited by other classes by using the `extends` keyword.
- Interfaces contain a set of abstract methods and static and final data members.
- A class that implements an interface must provide the implementation of all the methods declared in that interface.
- In Java, polymorphism has the following two types:
  - Static polymorphism
  - Dynamic polymorphism
- In case of static polymorphism, an entity, such as a method, can exist in multiple forms.
- Dynamic polymorphism is implemented in Java by method overriding.
- To override a method present in the superclass, the subclass method should have the same name, same parameters, and same return type as the method in the superclass.

NIIT

Handling Errors

CHAPTER 6

An *error* is a condition that causes a disruption in the execution of a program. An error can be a result of circumstances, such as incorrect code or insufficient memory resources. Errors in a Java program are categorized into two types, compile-time errors and
run-time errors. The compile-time errors occur when the syntax of a programming language are not followed. For example, in Java, if you use a keyword name as a variable name, a compile-time error will be raised by the compiler.

The run-time errors occur during the execution of a program. For example, if the program runs out of memory, it results in a run-time error. Such an error is known as an exception. An *exception* can lead to the termination of an application. Dealing with such abnormal behavior is called exception handling.

In addition, to create an error free code, you need to validate the program. In order, to validate the program you need to test various assumptions. For this, Java provides assertions.

This chapter discusses the common methods that you can use to handle exceptions. In addition, it also discusses the use of assertions in Java.

Objectives

In this chapter, you will learn to:

- Handle exceptions
- Use the assert keyword

Handling Exceptions

In the Classic Jumble Word game, if the player enters an incorrect menu option, such as a string value instead of an integer value, the game terminates abnormally. The abnormal termination of an application can have severe consequences in a real-time or a business application that provides a vital functionality to an organization. Therefore, it is essential to handle and prevent the abnormal termination. In order to prevent the abnormal termination in the Classic Jumble Word game, Sam wants to implement a functionality, such that the game application displays an appropriate message whenever an abnormal termination occurs.

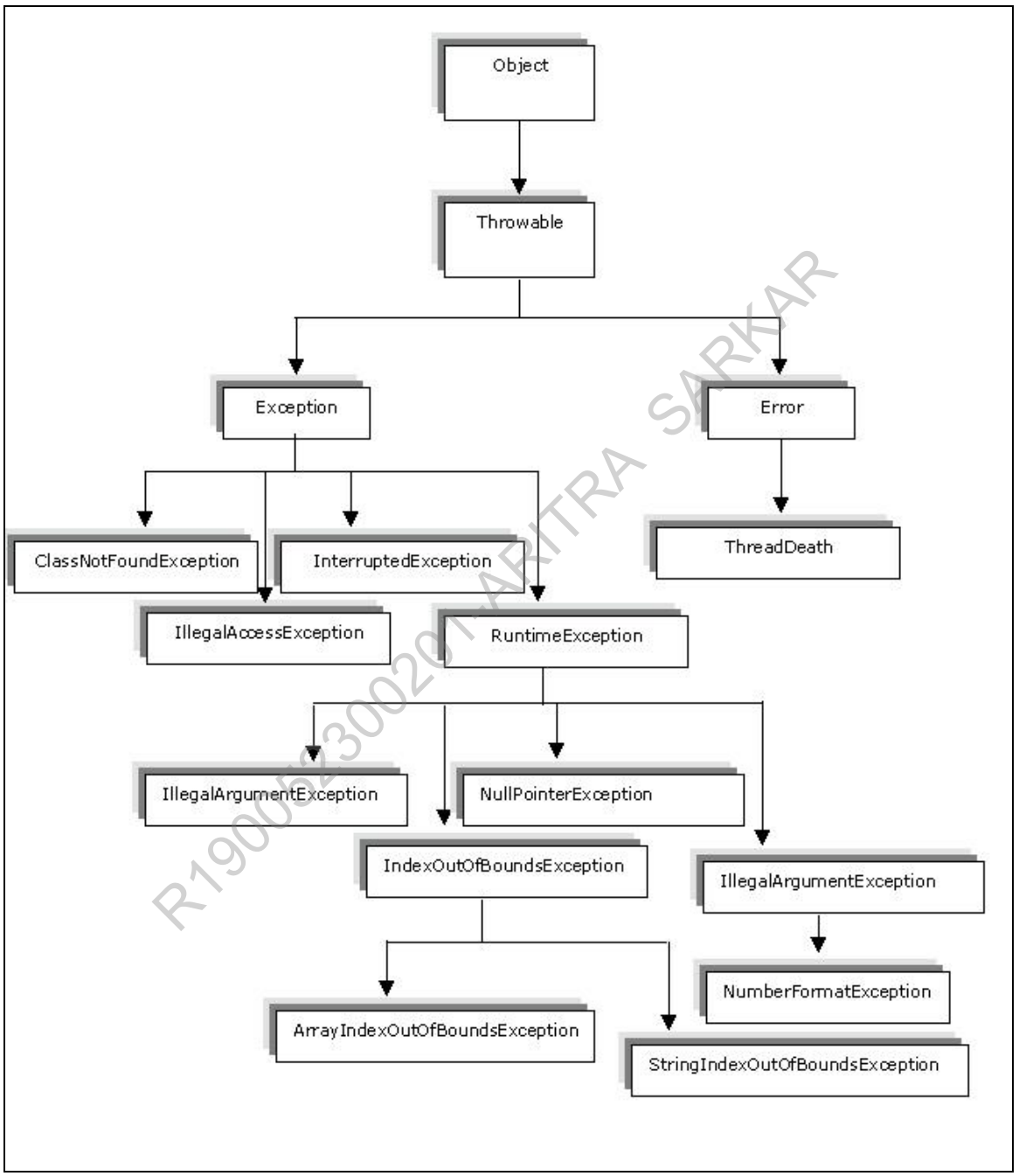
To cater to the preceding requirement, Sam should implement exception handling. The term exception in Java indicates an abnormal event that occurs during the program execution and disrupts the normal flow of instructions. For example, if you divide a number by zero or open a file that does not exist, an exception is raised. In Java, exceptions can be handled either by the Java run-time system or by a user-defined code. Java supports various built-in classes and exception handlers to deal with exceptions.

Exploring Exceptions

When a run-time error occurs, an exception is thrown by the JVM which can be handled by an appropriate exception handler. The Java run-time system proceeds with the normal execution of the program after an exception is handled. If no appropriate exception handler is found by the JVM, the program is terminated.

There are several built-in exceptions that have been identified in Java. In order to deal with these exceptions, Java has various built-in exception classes.

These built-in classes are organized in a hierarchical manner, as shown in the following figure.



The Exception Hierarchy

The Throwable Class

The `Throwable` class is the base class of exceptions in Java. You can throw only those exception objects that are derived from the `Throwable` class. The following two classes are derived for the `Throwable` class:

- `Exception`
- `Error`

The Exception Class

The `Exception` class represents the conditions that a program should handle. The `Exception` class has various subclasses, such as `classNotFoundException`, `IllegalAccessException`, and `RuntimeException`. The `ClassNotFoundException` exception is thrown when a class is being referred, but no definition for the same is found. The `IllegalAccessException` exception is thrown when a particular method is not found.

The Error Class

The `Error` class defines the exceptions related to the Java run-time environment. For example, `OutOfMemoryError` is an error that occurs when there is insufficient system memory to execute a program.

Identifying Checked and Unchecked Exceptions

Java exceptions are categorized into the following types:

- Checked exceptions
- Unchecked exceptions

Checked Exceptions

These are the invalid conditions that occur in a Java program due to the problems, such as accessing a file that does not exist or referring a class that does not exist. The *checked exceptions* are the objects of the `Exception` class or any of its subclasses excluding the `RuntimeException` and `Error` class. For example, a checked exception `IOException` is raised, when a program tries to read from a file that does not exist. The checked exceptions make it mandatory for a programmer to handle them. If a checked exception is not handled, then a compile-time error occurs.

The following table lists the most commonly used checked exceptions.

Exception	***Cause of Creation***
`ClassNotFoundException`	*Is thrown when the Java run-time system is unable to find the referred class.*
`IllegalAccessException`	*Is thrown when you refer an object, class, variable, constructor, or a method that is not accessible.*

Exception	*Cause of Creation*
`InstantiationException`	*Is thrown when you try to create an instance of a class by using the* `newInstance()` *method, but the referred class cannot be instantiated.*
`NoSuchMethodException`	*Is thrown when a particular method cannot be found.*

The Checked Exceptions in Java

Unchecked Exceptions

The *unchecked exceptions* occur because of programming errors. Therefore, the compiler does not force a programmer to handle these exceptions. Such errors should be handled by writing bug free codes. However, there are times such situations cannot be eradicated. For example in a calculator application, if you divide a number by zero, an unchecked exception is raised.

The following table lists the various unchecked exceptions and their causes of creation.

Exception	*Cause of Creation*
`ArithmeticException`	*Occurs when you make an arithmetic error, such as dividing a number by zero.*
`ArrayIndexOutOfBoundsException`	*Occurs when an attempt is made to access an array element beyond the index of the array.*
`ArrayStoreException`	*Occurs when you assign an element to an array that is not compatible with type of data that can be stored in that array.*
`ClassCastException`	*Occurs when you assign a reference variable of a class to an incompatible reference variable of another class.*
`IllegalArgumentException`	*Occurs when you pass an argument of incompatible data type to a method.*
`NegativeArraySizeException`	*Occurs when you create an array with a negative size.*
`NullPointerException`	*Occurs when an application tries to use an object without allocating memory to it or calls a method of a null object.*

Exception	*Cause of Creation*
`NumberFormatException`	*Occurs when you want to convert a string in an incorrect format to a numeric format.*

The Unchecked Exceptions in Java

Implementing Exception Handling

When an unexpected error occurs, Java creates an exception object. After creating the exception object, Java sends it to the program by throwing the exception. The exception object contains information about the type of error and the state of the program when the exception occurred. You need to handle the exception by using an exception handler and processing the exception. You can implement exception handling in a program by using the following keywords and blocks:

- `try`
- `catch`
- `throw`
- `throws`
- `finally`
- `try`-with-resources

Using try and catch Blocks

A `try` block encloses the statements that might raise an exception and defines one or more exception handlers associated with it. If an exception is raised within the `try` block, the appropriate exception handler that is associated with the `try` block processes the exception.

In Java, the `catch` block is used as an exception handler. A `try` block must have at least one `catch` block that follows the `try` block, immediately. The `catch` block specifies the exception type that you need to catch.

You can declare the `try` and the `catch` block by using the following syntax:

```
try
{
    // Statements that can cause an exception.
}

catch(exceptionname obj)
{
    // Error handling code.
}
```

In the preceding syntax, the `catch` block accepts the object of the `Throwable` class or its subclass that refers to the exception caught, as a parameter. When the exception is caught, the statements within the `catch` block are executed.

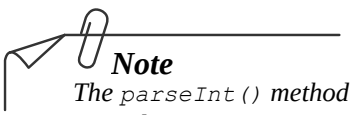
Consider a scenario in which you need to accept two integers from a user, performs their addition, and displays the result. For this, you can use the following code:

```
import java.util.Scanner;
public class Addition {

    public static void main(String[] args) {

        int num1, num2, result;
        Scanner obj1 = new Scanner(System.in);

        System.out.println("Enter the 1st number");
        num1 = obj1.nextInt();
        System.out.println("Enter the 2nd number");
        num2 = obj1.nextInt();
        result = num1+num2;
        System.out.println("The result is "+result);
    }
}
```

While executing the preceding code, if the user provides an input other than an integer value, an exception is raised and the program terminates abnormally. To overcome this, you can implement exception handling in the preceding code by using the try-catch block as shown in the following code:

```
import java.util.Scanner;
public class Addition
{
    public static void main(String[] args)
    {
          int num1, num2, result;
          String snum1, snum2;
          Scanner obj1 = new Scanner(System.in);
          try
                { // monitor the try block for exceptions
                  System.out.println("Enter the 1st number");
                  snum1 = obj1.next();
                  System.out.println("Enter the 2nd number");
                  snum2 = obj1.next();
                  num1=Integer.parseInt(snum1);
                  num2=Integer.parseInt(snum2);
                  result = num1+num2;
                  System.out.println("The result is "+result);
                }
          catch(Exception e) /*the catch block catches the exception raised
    in the try block and handles it*/
                {
                      System.out.println("Please input only numeric
values..!!");
                }

     }
}
```

In the preceding code, the `try` block is used to monitor the user input. If the user enters an input that is not an integer value, such as a char or a string, an exception is raised. However, the exception is successfully handled by the `catch` block that catches the exception raised in the `try` block. Once the exception is caught, the message, `Please input only numeric values..!!` is displayed and the abnormal termination of the program is handled, successfully.



Note

The `parseInt()` method is a static method defined inside the Integer wrapper class. It is used to convert a numeric string value, such as 12, to an integer value.

A `try` block can have multiple `catch` blocks. This is required when a code within the `try` block is capable of raising multiple exceptions. You can declare multiple `catch` blocks with a single `try` statement by using the following code snippet:

```
try
    {
    // statements
    }
catch(exceptionname1 obj1)
    {
    //statements to handle the exception
    }

catch(<exceptionname2 obj2)
    {
    //statements to handle the exception
    }
catch(exceptionnameN objN)
    {
    //statements to handle the exception
    }
```

In the preceding code snippet, when the statements in the `try` block raise an exception, the matching `catch` block corresponding to the raised exception is executed. Further, the execution continues after the try-catch block. While working with multiple `catch` statements, it is important to follow the exception hierarchy, such that the subclasses must appear prior to the superclasses. If the exception hierarchy is not followed, a compile-time error is generated.

Consider the following code:

```
public class Division
{
    public static void main(String[] args)
    {
        int num1, num2, result;
        Scanner obj1 = new Scanner(System.in);
        try
        { // monitor the try block for exceptions
            System.out.println("Enter the 1st number");
            num1 = obj1.nextInt();
```

```
            System.out.println("Enter the 2nd number");
            num2 = obj1.nextInt();
            result = num1 / num2;
            System.out.println("The result is " + result);
        } catch (Exception e)
        {
            System.out.println("Please input only numeric values..!!");
        } catch (ArithmeticException e) /*the catch block catches the
exception raised in the try block and handles it */
        {
            System.out.println("Division performed by zero...");
        }
    }
}
```

The preceding code will generate a compile-time error as the exception hierarchy is not followed. Therefore, in the preceding code `ArithmeticException` should be caught before `Exception`.

throw

You can throw an exception explicitly, by using the `throw` keyword. For example, you need to throw an exception when a user enters an incorrect login ID or a password. The `throw` keyword causes the termination of the normal flow of control of the Java code and stops the execution of the subsequent statements. The `throw` keyword transfers the control to the nearest `catch` block that handles the type of exception being thrown. If no appropriate `catch` block exists, the program terminates.

You can throw an exception by using the following syntax:

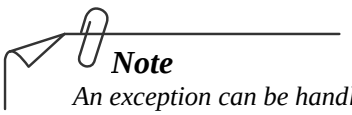
```
throw ThrowableObj
```

In the preceding syntax, `ThrowableObj` is an object of the `Throwable` class or a subclass of the `Throwable` class, such as an `Exception` class. The `ThrowableObj` object is created by using the `new` operator. The Java compiler gives an error if the `ThrowableObj` object does not belong to a valid `Exception` class.

Consider the following code that demonstrates the implementation of the `throw` statement:

```
public class ThrowDemo
{
    void display()
    {
        throw new RuntimeException();
    }

    public static void main(String[] args)
    {
        ThrowDemo obj1 = new ThrowDemo();
        try
        {
            obj1.display();
        } catch (RuntimeException e)
        {
            System.out.println("Runtime Exception raised");
        }
```

```
    }
}
```



Note

An exception can be handled either in the method that raises the exception or in the calling method.

In the preceding code, the statement, `throw new RuntimeException` is used to throw a `RuntimeException` object and the calling method, `main()` handles the exception raised in the `display()` method.

The `throw` keyword can also be used inside a `catch` block to rethrow an exception. Exceptions are rethrown if a method causing the error wants to pass the error handling responsibilities to the exception handlers in some other methods. You can use the following code snippet to rethrow an exception:

```
catch(Exception e)
{
    System.out.println("Exception Raised");
    throw e;
}
```

In the preceding code snippet, the exception goes to the `catch` block of the next higher context ignoring the `catch` blocks of the same `try` block. You can use the following code to catch the `RuntimeException` exception and rethrow the exception to the outer handler:

```
class RethrowException
{
    static void compute()
    {
            try
            {
                    throw new RuntimeException ("My Exception");
            }
            catch(RuntimeException e)
            {
                    System.out.println("Exception caught in compute() method");
                    throw e; // Rethrow the Exception.
            }
    }

    public static void main(String args[])
    {
            try
            {
                    compute();
            }
            catch(RuntimeException e)
            {
                    System.out.println("Exception caught:" + e);
```

```
            }
        }
    }
```

In the preceding code, the `catch` block in the `compute()` method rethrows the `RuntimeException` exception to the `catch` block defined in the `main()` method.

throws

The `throws` keyword is used by a method to specify the types of exceptions that the method can throw. If a method is capable of raising a checked exception that it does not handle, then the method must specify that the exception is handled by the calling method. This is done by using the `throws` keyword.

The `throws` keyword lists the checked exceptions that a method can throw. You can use the following syntax to declare a method that specifies a `throws` keyword:

```
<access_specifier> <modifier> <return_type> <method_name> (<arg_list>) throws
<exception_list>
```

In the preceding syntax, `<access_specifier> <modifier>` specifies the access specifiers and modifiers, such as `public` and `static`, respectively. `<return_type>` specifies the type of value returned by the method. `<method_name>` specifies the name of the method. `(<arg_list>)` specifies the arguments that the method can accept and `throws <exception_list>` specifies the exceptions that the method can throw.

Consider the following code that demonstrates the implementation of the `throws` statement:

```
public class ThrowsDemo
{
     void display() throws Exception
        {
        throw new Exception();
        }

     public static void main(String[] args) {

         ThrowsDemo obj1 = new ThrowsDemo();
         try
            {
            obj1.display();
            }
         catch (Exception e)
            {
            System.out.println("Runtime Exception raised");
            }
     }
}
```

In the preceding code, the `display()` method declares that an `Exception` class exception can be generated. Therefore, the try-catch blocks have been defined to catch the exception in the `main()` method.

finally

During the execution of a Java program, when an exception is raised, the rest of the statements in the `try` block are ignored. Sometimes, it is necessary to execute certain statements irrespective of whether an exception is raised. The `finally` block is used to execute these required statements. The statements specified in the `finally` block are executed after the control has left the try-catch block.

You can use the following syntax to declare the `try` and `finally` block:

```
try
{
    // Block of code
}
finally
{
// Block of code that is always executed irrespective of an exception being
raised.
}
```

In the preceding syntax, the `try` and the `finally` blocks are declared.

If there is a `catch` block associated with the `try` block, the `finally` block is written after the `catch` block. The following code snippet shows how to declare the `try`, `catch`, and `finally` blocks:

```
try
{
    // Block of code.
}
catch(execptionname1 obj1)
{
    System.out.println("Exception1 has been raised");
}
catch(exceptionname2 obj2)
{
    System.out.println("Exception2 has been raised");
}
finally
{
// Block of code that is always executed irrespective of an
 exception being raised or not.
}
```

In the preceding code snippet, the `try`, `catch`, and the `finally` block are declared.

The `finally` block executes irrespective of whether or not an exception is raised. If an exception is raised, the `finally` block executes even if none of the `catch` blocks match the exception.

try-with-resources

The `try`-with-resources statement is similar to the `try` block. However, it is essentially used to declare and automatically close the objects, such as the file streams and database connections after the execution of the `try` block finishes. Such objects are known as resources. In order to be handled by the `try`-with-resources statement, the resource must implement the `java.lang.AutoCloseable` interface.

The `try`-with-resources block ensures that one or more system resources are released when they are no longer required. Consider a scenario where you are monitoring code that deals with the opening of a resource inside a `try` block. You have also specified the instructions to execute in a `finally` block. If the `try` block raises an exception, then the statements in the `finally` block are executed. In case, if the `finally` block also encounters an exception, then the object will never be closed. In such a case, to make your application more efficient and deal with the improper resource management, the `try`-with-resources statement can be used.

You can use the following syntax to declare the `try`-with-resources statement:

```
try( [resource-declaration 1];
[resource-declaration n];
)

{
//code to be executed
}
//after the try block, the resource is closed
```

In the preceding syntax, multiple resources separated with "`;`" can be declared with the `try` statement.

The following code snippet shows how to implement the `try`-with-resources statement:

```
try (BufferedReader br = new BufferedReader(new FileReader("<file_path>")))
{
     return br.readLine();
}
```

In the preceding code snippet, `BufferedReader` is declared as a resource and is used to read data from a file provided at the location specified by `<file_path>`. The class `BufferedReader` is a built-in class that implements the `java.lang.AutoCloseable` interface. The preceding code snippet ensures that the `br` instance of `BufferedReader` is closed after the execution of the `try` block. You can also achieve the preceding result by specifying a `finally` block and including the statements to close the instance in it. However, using the `try`-with-resources statement reduces code and improves the readability of code.

User-defined Exceptions

In addition to the built-in exceptions, you can create customized exceptions, as per the application requirements. To create a user-defined exception, you need to perform the following steps:

- Create an exception class.
- Implement user-defined exception.

Creating an Exception Class

If you want to create a new user-defined exception, then the class should extend the `Throwable` class or its subclasses.

Consider the following code snippet:

```
public class AgeException extends RuntimeException
{
     public AgeException()
          {
                  System.out.println("Invalid value for age");
          }



AgeException(String msg)
          {
                  super(msg);
          }
}
```

In the preceding code snippet, the `AgeException` class is created. The `AgeException` class extends `RuntimeException` which is a subclass of the `Exception` class. The constructor of the `AgeException()` class prints the message to state that an invalid age value has been entered. In addition, it has an overloaded constructor which accepts string objects.

Implementing User-defined Exception

Consider the following code to demonstrate the implementation of a user-defined exception:

```
import java.util.*;
public class ValidateAge
{
  public static void main(String[] args)
    {
         int age;
         Scanner obj1 = new Scanner(System.in);
         System.out.println("Enter the age: ");
         age = obj1.nextInt();
          if (age <= 0){
             try
                  {
                         throw new AgeException();
                  }
           catch (AgeException e)
                  {
                         System.out.println("Exception raised");

                  }
    }
         else
                  {
                         System.out.println("Age entered is " + age);
                  }

    }

}
```

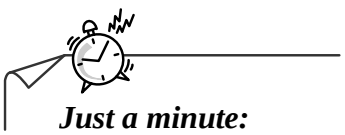
In the preceding code, if the entered age is less than or equal to `0`, the user-defined exception, `AgeException` is thrown. When the exception is thrown, the constructor defined in the class of the user-defined exception is invoked and the `Invalid value for age` message is displayed.

For example, when the age is specified as `12`, the following output is displayed:

```
Age entered is 12
```

When the age is specified as `0`, the following output is displayed:

```
Invalid value for age
Exception raised
```



Just a minute:

Which one of the following keywords lists the checked exceptions that a method can throw?

1. `throws`
2. `throw`
3. `catch`
4. `finally`

Answer:

1. `throws`

Activity 6.1: Exception Handling

Using the assert Keyword

Assertions are statements in Java that enable you to test any assumptions that you make regarding a program during its execution. The assumptions in a program can be simple facts, such as the range of a number should be between 10 and 20 or a number cannot be greater than 100. For example, in an application that computes interest, you are certain that the time period cannot be less than or equal to 0. In such a scenario, you can use an assertion to validate your code and confirm that your application does not use an incorrect value for the time period. You can implement assertions by using the `assert` keyword provided in Java.

Understanding Assertions

Assertions are used during the testing of a program. They enable you to test the assumptions made in a program during its execution. For example, a particular program may require the value being passed to a method, to be positive. You can test this by asserting that the value being passed to that particular method is greater than zero by using an `assert` keyword.

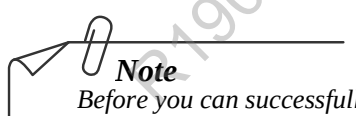
Assertions in Java provide the following benefits:

- By using assertions, data can be validated easily. For example, in a method that performs division, you can use an assertion to validate at run-time if that method is performing division by zero.
- By using assertions, you can confirm whether the program is working as expected.
- By using assertions, the task of debugging is simplified as assertions can easily indicate the source and the reason for an error.

Implementing Assertions

The `assert` keyword is used to implement assertions. You can declare an assertion by using the following syntax:

```
assert expressionA;
```



Note

Before you can successfully implement assertions in your code, you need to enable them in the Java environment.

In the preceding syntax, `expressionA` is a boolean expression. During the execution of program, `expressionA` in the assert statement is tested. If `expressionA` returns `true`, then:

- The assertion made in the program is true.
- The program execution continues uninterruptedly.
- No action takes place.

However, if `expressionA` returns `false`, then the assertion made in the program fails. The program throws an `AssertionError` object and the program execution terminates.

Consider the following code:

```
public class ValidateAge
{
   public static void main(String[] args)
    {
         int age;
         Scanner obj1 = new Scanner(System.in);
         System.out.println("Enter the age: ");
         age = obj1.nextInt();
         assert(age>0)&&(age<130);
         System.out.println("The entered age is: " +age);
    }
}
```

In the preceding code, an assertion is placed is place to check that the value of `age` is greater than `0`, but less than `130`. While executing the code, if the value of the variable `age` is less than or equal to `0` or greater than `130`, the program terminates with the following assertion error:

```
Exception in thread "main" java.lang.AssertionError
```

You can also declare assertion by using the following syntax:

```
assert expression1 : expression2;
```

In the preceding syntax, `expression1` is a boolean expression and `expression2` is an error message value that is displayed if the assertion fails. `expression2` is passed to the constructor of the `AssertionError` class, if `expression1` returns `false`. The `AssertionError` constructor converts the value in `expresion2` into a string format and displays the message if the assertion fails.

Just a minute:

Which one of the following errors is generated if an assertion does not hold true?

1. *AssertionError*
2. *IllegalAccess*
3. *TypeMismatch*
4. *IllegalAssertion*

Answer:

1. *AssertionError*

Activity 6.2: Implementing Assertions

Practice Questions

1. Which of the following options is the superclass of all the exception objects?
   a. `Error`
   b. `Throwable`
   c. `RuntimeException`
   d. `Exception`

2. Which one of the following options represents invalid conditions that occur in a Java program due to problems, such as database problems?
   a. Checked exception
   b. Unchecked exception
   c. Assertion
   d. Postconditions

3. Which one of the following checked exceptions occurs when you try to create an object of an abstract class or an interface by using the `newInstance()` method?
   a. `ClassNotFoundException`
   b. `IllegalAccessException`
   c. `InstantiationException`
   d. `NoSuchMethodException`

4. Which one of the following options is used to implement assumptions in a Java program?
   a. `assertion`
   b. `asserts`
   c. `assert`
   d. `invariant`

5. Which one of the following keywords is used to enclose the statements that might raise an exception?
   a. `try`
   b. `catch`
   c. `throw`
   d. `finally`

Summary

In this chapter, you learned that:

- When a run-time error occurs, an exception is thrown by the JVM which can be handled by an appropriate exception handler.
- To deal with these exceptions, Java provides various built-in exception classes.
- The `Throwable` class is the base class of exceptions in Java.
- The `Exception` class represents the conditions that a program should handle.
- The `Error` class defines the exceptions related to the Java run-time environment.
- Java exceptions are categorized into the following types:
  - Checked exceptions
  - Unchecked exceptions
- You can implement exception handling in a program by using the following keywords and blocks:
  - `try`
  - `catch`
  - `throw`
  - `throws`
  - `finally`
  - `try`-with-resources
- A `try` block encloses the statements that might raise an exception and defines one or more exception handlers associated with it.
- In Java, the `catch` block is used as an exception handler.
- A `try` block must have at least one `catch` block that follows the `try` block, immediately.
- You can throw an exception explicitly, by using the `throw` keyword.
- The `throws` keyword is used by a method to specify the types of exceptions that the method can throw.
- The statements specified in the `finally` block are executed after the control has left the try-catch block.
- The `try`-with-resources block ensures that one or more system resources are released when they are no longer required.
- In addition to the built-in exceptions, you can create customized exceptions, as per the application requirements.
- Assertions are statements in Java that enable you to test any assumptions that you make regarding a program during its execution.
- You can implement assertions by using the `assert` keyword provided in Java.

NIIT

Working with Regular Expressions

CHAPTER 7

At times, you need to develop applications that provide a functionality, such as search or filter, which requires processing of strings. To develop such applications, you need to write lengthy code, which is a time-consuming task. To solve this issue, Java supports regular expressions also known as regex. *Regular expressions* allow you to match a character sequence with another string.

At times, you want to develop applications that can be used worldwide. These applications should be able to display various pieces of information, such as text, currency, and date, according to the region where they are deployed. To meet this requirement, Java supports localization.

In this chapter, you will learn about processing strings using regular expressions and implementing localization.

Objectives

In this chapter, you will learn to:

- Process strings using regex

Processing Strings Using Regex

Consider a scenario where you need to develop a Java application that provides the functionality to search the text, hello, in a particular file. To accomplish this task, Java provides a support for regular expressions. A regular expression is a string that defines a character sequence called pattern. This pattern can be matched against another string. To work with regular expressions, Java provides two classes, `Pattern` and `Matcher`, in the `java.util.regex` package.

In addition, Java provides character classes to work with a set of characters. Further, to identify the number of occurrences of a pattern in a string, Java supports quantifiers.

Note

In regular expressions, a group of characters enclosed within square brackets is called a character class.

Working with the Pattern and Matcher Classes

The `Pattern` class represents a compiled regular expression. As the `Pattern` class does not define any constructor, the `Pattern` class reference is obtained by using the static method, `compile()`, of the `Pattern` class.

The following code snippet is used to create a reference of the `Pattern` class:

```
Pattern myPattern = Pattern.compile("Expression");
```

In the preceding code snippet, the regular expression, `Expression`, is passed to the `compile()` method, which returns the object of the `Pattern` class. Then, the `Pattern` object is used to create a `Matcher` object, as shown in the following code snippet:

```
Matcher myMatcher = myPattern.matcher("Expression");
```

To compare the two expressions, the `Matcher` class provides the `matches()` method. The `matches()` method returns the value, `true`, if the expression in the matcher object matches the pattern specified in the pattern object. Otherwise, it returns the value, `false`. The following code snippet shows the usage of the `matches()` method:

```
boolean myBoolean = myMatcher.matches();
```

The following table lists the most commonly used methods of the `Pattern` class.

Method	***Description***	***Example***

`String pattern()`	*Is used to return the compiled regular expression.*	`String s1 = "pattern";`
		`Pattern myPattern = Pattern.compile(s1);`
		`String s2 = myPattern.pattern();`
		`System.out.println(s2);`
		Output:
		`pattern`

`String[] split(CharSequence input, int limit)`	*Is used to split the given input sequence based on the pattern and the limit.*	`Pattern pattern = Pattern.compile(":");`
		`String[] split = pattern.split("One:two:three",2);`
		`for (String element : split)`
		`{`
		`System.out.println("element = " + element);`
		`}`
		Output:
		`element = One`
		`element = two:three`

`static boolean matches(String regex,CharSequence input)`	*Is used to compile the given regular expression and attempt to match the given input against it.*	`boolean matches = Pattern.matches("is", "is");`
		`System.out.println("matches = " + matches);`
		Output:
		`matches = true`

The Methods of the Pattern Class

The following table lists the most commonly used methods of the `Matcher` class.

Method	***Description***	***Example***

`Matcher appendReplacement(StringBuffer sb, String replacement)`	*Is used to process the input character sequence specified in a matcher by adding the input character sequence to the `StringBuffer` variable. If the match specified in `Pattern` is found in `Matcher`, then it is replaced by the replacement string and is appended to the `StringBuffer` variable. Further, the remaining character sequence is truncated.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`StringBuffer s3 = new StringBuffer();`
		`while(matcher.find())`
		`{`
		`matcher.appendReplacement(s3, "sam");`
		`System.out.println(s3.toString());`
		`}`
		Output:
		`sam`
		`sam does this, and sam`

`StringBuffer appendTail(StringBuffer sb)`	*Is used to process the input character sequence specified in `Matcher` by adding the input character sequence to the `StringBuffer` variable. If the match specified in `Pattern` is found in `Matcher`, then it is replaced by the replacement string and is appended to the `StringBuffer` variable. Further, the remaining character sequence is appended.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`StringBuffer s3 = new StringBuffer();`
		`while(matcher.find())`
		`{`
		`matcher.appendReplacement(s3, "sam");`
		`System.out.println(s3.toString());`
		`}`
		`matcher.appendTail(s3);`
		`System.out.println(s3.toString());`
		Output:
		`sam`
		`sam does this, and sam`
		`sam does this, and sam does that`

Method	*Description*	*Example*

`String replaceAll(String replacement)`	*Is used to replace every subsequence of the input sequence that matches the pattern with the given replacement string.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`String s2 = matcher.replaceAll("sam");`
		`System.out.println("replaceAll = " + s2);`
		Output:
		`replaceAll = sam does this, and sam does that`

`String replaceFirst(String replacement)`	*Is used to replace the first subsequence of the input sequence that matches the pattern with the given replacement string.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`String s2 = matcher.replaceFirst("sam");`
		`System.out.println("replaceFirst = " + s2);`
		Output:
		`replaceFirst = sam does this, and John does that`

`int start(int group)`	*Is used to return the start index of the subsequence captured by the given group during the match operation.*	`Pattern pattern = Pattern.compile("a");`
		`Matcher matcher = pattern.matcher("This is a text");`
		`while(matcher.find())`
		`{`
		`System.out.println("Match started at:"+ matcher.start(0));`
		`}`
		Output:
		`Match started at:8`

Method	Description	Example

`int end(int group)`	*Is used to return the offset after the last character of the subsequence is captured by the given group during the match operation.*	`Pattern pattern = Pattern.compile("a");`
		`Matcher matcher = pattern.matcher("This is a text");`
		`while(matcher.find())`
		`{`
		`System.out.println("Match ended at:"+ matcher.end(0));`
		`}`
		Output:
		`Match ended at:9`

The Methods of the Matcher Class

Working with Character Classes

Consider a scenario where you need to search certain words in a file that can have the prefix " a", "b", or "c" and suffix "at", such as, bat and cat. For this, regular expressions provide character classes. The character class specifies the characters that will successfully match a single character from a given input string.

The following table lists the constructs of the character class.

Construct	Description
`[def]`	*Match succeeds if the given input string starts with the characters: d, e, or f.*
`[^def]`	*Match succeeds if the given input string starts with any character except d, e, or f (negation).*
`[a-zA-Z]`	*Match succeeds if the given input string starts with any character between a to z or A to Z, inclusive (range).*
`[b-e[n-q]]`	*Match succeeds if the given input string starts with any character between b to e or n to q.*
`[a-z&&[abc]]`	*Match succeeds if the given input string starts with characters: a, b, or c (intersection).*
`[a-z&&[^bcd]]`	*Match succeeds if the given input string starts with any character between a to z, except b, c, and d :* `[ae-z]` *(subtraction).*

Construct	*Description*
`[a-z&&[^n-p]]`	*Match succeeds if the given input string starts with any character between a to z and not n to p: [a-mq-z] (subtraction).*

The Constructs of the Character Class

Consider the following code that demonstrates the use of the character class:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class TestRegx
{
    public static void main(String[] args)
    {
        Pattern myPattern = Pattern.compile("[abc]at");
        Matcher myMatcher = myPattern.matcher("bat");
        boolean myBoolean = myMatcher.matches();
        if(myBoolean)
        System.out.println("Expression Matched");
        else
        System.out.println("Expression Not Matched");
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Expression Matched
```

In the preceding code, `myBoolean`, will have the value, `true`, only when the first letter matches one of the characters defined by the character class.

Working with Quantifiers

Consider a scenario where you want to specify the number of times a character or a sequence of characters appears in an expression. For example, you want to find the occurrence of the characters, Ro, in the expression, "When in Rome, do as the Romans". To accomplish this task, a regular expression provides quantifiers that help you specify the number of occurrences of a string matched with the specified string. In addition, quantifiers enable you to easily select a range of characters in the files. Quantifiers are of the following three types:

- **Greedy**: Is used to match with the longest possible string that matches the pattern.
- **Reluctant**: Is used to match with the shortest possible string that matches the pattern.
- **Possessive**: Is used to match the regular expression with the entire string. It matches only when the whole string satisfies the criteria.

The following table describes the different types of quantifiers.

Greedy	*Reluctant*	*Possessive*	*Description*
`X?`	`X??`	`X?+`	*`X`, once or not at all*
`X*`	`X*?`	`X*+`	*`X`, zero or more times*
`X+`	`X+?`	`X++`	*`X`, one or more times*
`X{n}`	`X{n}?`	`X{n}+`	*`X`, exactly `n` times*
`X{n,}`	`X{n,}?`	`X{n,}+`	*`X`, at least `n` times*
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	*`X`, at least `n` times but not more than `m` times*

The Types of Quantifiers

Consider the following code snippet to test the occurrence of the word beginning with Ro in the expression, When in Rome, do as the Romans:

```
String text="When in Rome, do as the Romans ";
String textSplit[]=text.split(" ");
Pattern myPattern=Pattern.compile("Ro.+");
for(int i=0;i<textSplit.length;i++)
{
    Matcher myMatcher=myPattern.matcher(textSplit[i]);
    boolean myBoolean =myMatcher.matches();
    System.out.println(myBoolean);
}
```

In the preceding code snippet, the `split()` method splits the expression "`When in Rome, do as the Romans`". Thereafter, the pattern, `Ro`, is compared with each split expression. The `System.out.println(myBoolean);` statement prints the value, `true`, if the first two characters of the word are "`Ro`".

Consider the following code that accepts the pattern and matcher from the user:

```
import java.util.regex.*;
import java.util.*;
public class PatternMethods
{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the desired pattern: ");
        String pattern = input.nextLine();
        System.out.println("Enter the text: ");
        String matcher = input.nextLine();
        Pattern myPattern2 = Pattern.compile(pattern);
```

```
            Matcher myMatcher2 = myPattern2.matcher(matcher);
            Boolean myBoolean2 = myMatcher2.matches();
            boolean b = myBoolean2;
            if (b == true)
            {

               System.out.println("I found the text:" + myMatcher2.group() + "\n"
                        + "Starting at:" + myMatcher2.start() + "\n"
                        + "Ending at index:" + myMatcher2.end());

            }
            else if (b == false)
            {
                System.out.println("No match found");
            }

    }
}
```

While executing the preceding code, if you provide the pattern as `.*Friday` and matcher as `XFriday`, then the following output will be displayed:

```
I found the text:XFriday
Starting at:0
Ending at index:7
```

The preceding output is generated as the pattern uses the greedy quantifier. Since the quantifier is greedy, initially, the entire input string is consumed. Because the entire input string does not match with the pattern, the matcher traverses back by one letter at a time, until the rightmost occurrence of "`Friday`" has been found. At this point, the match succeeds and the search ends.

While executing the preceding code, if you provide the pattern as `.*?Friday` and matcher as `XFriday`, then the following output will be displayed:

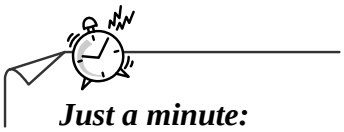
```
I found the text:XFriday
Starting at:0
Ending at index:7
```

The preceding output is generated as the pattern uses the reluctant quantifier. Since the quantifier is reluctant, initially, no character of the input string is consumed. Thereafter, the characters of the string are consumed one by one.

While executing the preceding code, if you provide the pattern as `.*+Friday` and matcher as `XFriday`, then the following output will be displayed:

```
No match found
```

The preceding output is generated as the pattern uses the possessive quantifier. Since the quantifier is possessive, initially, the entire input string is consumed. Because this quantifier does not traverse back, it fails to find a match.



Just a minute:

Which one the following methods is used for compiling the given regular expression and attempting to match the given input against the specified pattern?

1. `static boolean matches(String regex,CharSequence input)`
2. `Pattern pattern()`
3. `static Pattern compile(String regex,int flags)`
4. `Matcher appendReplacement(StringBuffer sb,String replacement)`

Answer:

1. `static boolean matches(String regex,CharSequence input)`

Activity 7.1: Processing Strings Using Regex

Practice Questions

1. Which one of the following pattern class methods is used to return the regular expression from which the pattern was compiled?
   a. `String pattern()`
   b. `static boolean matches(String regex,CharSequence input)`
   c. `static Pattern compile (String regex)`

2. Which one of the following quantifiers is used to match with the shortest possible string that matches the pattern?
   a. Greedy
   b. Reluctant
   c. Possessive

3. Identify the correct option by using which the match succeeds if the given input string starts with any character between a and z, except b, c, and d.
   a. `[a-z&&[abc]]`
   b. `[b-z&&[^bcd]]`
   c. `[a-z&&[^n-p]]`
   d. `[abc]`

4. Identify the correct option that is used to match the pattern with the longest possible string.
   a. Greedy
   b. Reluctant
   c. Possessive

Summary

In this chapter, you learned that:

- Regular expressions allow you to match a character sequence with another string.
- A regular expression is a string that defines a character sequence called pattern.
- To work with regular expressions, Java provides two classes, `Pattern` and `Matcher`, in the `java.util.regex` package.
- Java provides character classes to work with a set of characters.
- To identify the number of occurrences of a pattern in a string, Java supports quantifiers.
- The `Pattern` class represents a compiled regular expression.
- As the `Pattern` class does not define any constructor, the `Pattern` class reference is obtained by using the static method, `compile()`, of the `Pattern` class.
- The character class specifies the characters that will successfully match a single character from a given input string.
- A regular expression provides quantifiers that help you specify the number of occurrences of a string matched with the specified string.
- Quantifiers are of the following three types:
  - Greedy
  - Reluctant
  - Possessive

NIIT

Working with Streams

CHAPTER 8

Most programs accept input from the user, process the same, and produce the output. Therefore, all programming languages support the reading of input from an input stream, such as a file, and display the output from an output stream, such as a console. Java handles all the input and output operations in the form of streams that act as a sequence of bytes or characters traveling from a source to a destination. When a stream of data is being sent, it is said to be written; and when a stream of data is being received, it is said to be read.

This chapter focusses on reading the data from an input stream and writing the same to an output stream.

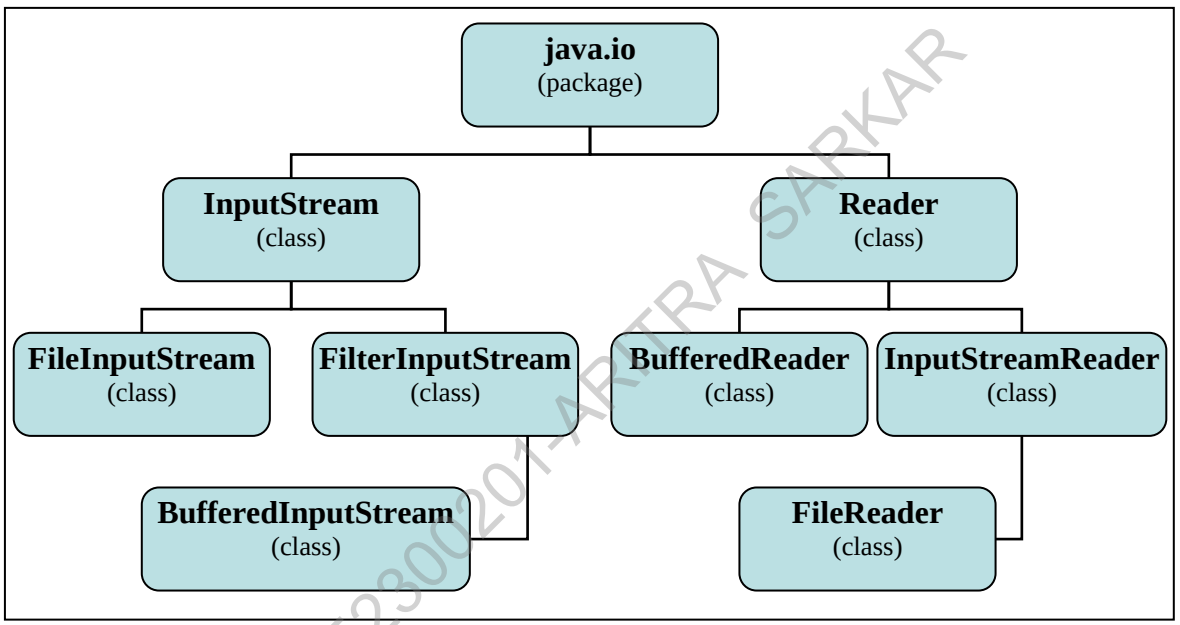
Objectives

In this chapter, you will learn to:

- Work with input stream
- Work with output stream

Working with Input Stream

Consider a scenario where you need to develop a Java application that converts the documents from one format into another, such as .txt into .pdf. To develop such an application, you need to read data from an input stream, such as a file. The data can be read in the form of bytes or characters from these streams. For this, Java provides the `InputStream` class and its subclasses. To read data in the form of characters, Java provides the `Reader` classes inside the `java.io` package. The following figure shows the class hierarchy of the `java.io` package.



The Input Stream Class Hierarchy

The `FileInputStream` and `FilterInputStream` classes are subclasses of the `InputStream` class. The `BufferedInputStream` class is a subclass of the `FilterInputStream` class. The `BufferedReader` and `InputStreamReader` classes are the subclasses of the `Reader` class. The `FileReader` class is the subclass of the `InputStreamReader` class.

Using the FileInputStream Class

The *`FileInputStream` class* is used to read data and the steams of bytes from the file. The `FileInputStream` class provides various constructors that can be used to create an instance.

The following table lists the constructors of the `FileInputStream` class.

Constructor	***Description***
`FileInputStream(File file)`	*It creates an instance of `FileInputStream` by opening a connection to an actual file, `file`, in the file system.*
`FileInputStream(FileDescriptor fdObj)`	*It creates an instance of `FileInputStream` by using the file descriptor, `fdObj`, which represents an existing connection to an actual file in the file system.*
`FileInputStream(String name)`	*It creates `FileInputStream` by opening a connection to an actual file specified in the `name` parameter.*

The Constructors of the FileInputStream Class

The following table lists the methods of the `FileInputStream` class with their description.

Method	***Description***
`int read()`	*Is used to read a byte of data from the input stream.*
`FileDescriptor getFD()`	*Is used to return the `FileDescriptor` object that represents the connection to the actual file in the file system.*

The Methods of the FileInputStream Class

Consider the following code to read the data from a file using the `FileInputStream` class:

```
import java.io.FileInputStream;
import java.io.IOException;
public class FileInputStreamDemo
{

    public static void main(String[] args)
    {
        int i;
        char c;
        try (FileInputStream f = new FileInputStream("D:\\Files\\File.txt"))
        {
            while ((i = f.read()) != -1)
            {
                c = (char) i;
                System.out.print(c);
            }
        }
```

```
            catch (IOException ex)
            {
                System.out.println(ex);
            }
        }
    }
```

In the preceding code, the `FileInputStreamDemo` class is created that is reading data from the `File.txt` file by using the `read()` method. The `System.out.print(c);` statement prints the bytes of data present in the file.

Using the BufferedInputStream Class

The *`BufferedInputStream` class* is used to perform the read operations by using a temporary storage, buffer, in the memory. A buffer is used to store the retrieved data when the first system call to read data is made. In addition, for every subsequent request to read data, the buffer is used to retrieve data by a program instead of making system calls. The retrieval of data by using a buffer instead of a system call is much faster; and hence, it improves the efficiency of the program. The `BufferedInputStream` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `BufferedInputStream` class.

Constructor	***Description***
`BufferedInputStream(InputStream in)`	*It creates an instance of `BufferedInputStream` and saves the specified `InputStream` arguments in the `in` parameter for later use.*
`BufferedInputStream(InputStream in, int size)`	*It creates an instance of `BufferedInputStream` with the specified buffer size, and saves the `InputStream` argument in the `in` parameter for later use.*

The Constructors of the BufferedInputStream Class

The following table lists the methods of the `BufferedInputStream` class with their description.

Method	***Description***
`int available()`	*Is used to return an estimate of the number of bytes that can be read from this input stream without blocking the next invocation of a method for this input stream.*
`void mark(int readlimit)`	*Is used to mark the present position in the stream. Subsequent calls to `reset()` will attempt to reposition the stream to this point.*

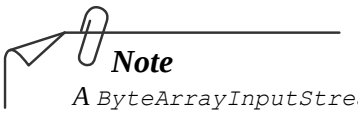
Method	*Description*
void reset()	*The reset() method returns the stream to the marked point.*
boolean markSupported()	*Is used to test if this input stream supports the mark() and reset() methods.*
void close()	*Is used to close the input stream and release any system resource associated with the stream. Once the stream is closed, further read(), available(), reset(), or skip() invocation will throw IOException.*
int read(byte[] b,int off,int len)	*Is used to read bytes from the input stream into the specified byte array, starting from the given offset.*
long skip(long n)	*Is used to skip over n bytes of data from this input stream.*

The Methods of the BufferedInputStream Class

Consider the following code of BufferedInputStream that prints the output on the console:

```
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;

public class BufferedInputStreamDemo
{
    public static void main(String[] args)
    {
        String s = "This is a BufferedInputStream Demo Program";
        byte buf[] = s.getBytes();

        try(ByteArrayInputStream in = new ByteArrayInputStream(buf);
              BufferedInputStream f = new BufferedInputStream(in)){

        int c;
        while ((c = f.read()) != -1)
        {
            System.out.print((char) c);
        }
    }
 catch(IOException e)
{
        System.out.println(e);
    }
}
}
```

In the preceding code, the `BufferedInputStreamDemo` class has been created. The `System.out.print((char) c);` statement prints the output on the console.



Note

A `ByteArrayInputStream` class allows you to create a buffer in memory that will be used as `InputStream`. The input source is a byte array.

Using the FileReader Class

The *`FileReader` class* is used for reading characters from a file, but it does not define any method of its own. It derives all the methods from its base classes, such as the `Reader` and `InputStreamReader` classes. The `FileReader` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `FileReader` class.

Constructor	***Description***
`FileReader(File file)`	*It creates an instance of `FileReader` that reads from the file specified in the `file` parameter.*
`FileReader(FileDescriptor fd)`	*It creates an instance of `FileReader` that reads from the specified `FileDescriptor` class.*
`FileReader(String fileName)`	*It creates an instance of `FileReader` that reads from the file with the name specified by the `fileName` parameter.*

The Constructors of the FileReader Class

Consider the following code that reads data from the file:

```
import java.io.FileReader;
import java.io.IOException;

class FileReaderDemo {
    public static void main(String args[])
    {
        try (FileReader f = new FileReader("D:\\Files\\file.txt "))
        {
            char[] a = new char[50];
            f.read(a);
            for (char c : a)
            {
                System.out.print(c);
            }
        }
```

```
            catch (IOException e)
            {
                System.out.println(e);
            }
        }
    }
```

In the preceding code, the `FileReaderDemo` class has been created that reads the data from the file, `file.txt`. The `System.out.print(c);` statement prints the output on the console.

Using the BufferedReader Class

The *`BufferedReader` class* is used to read the text from a character-input stream, such as a file, console, and array, while buffering characters. As the data is buffered, the read operation becomes more efficient using the `BufferedReader` class. There are various constructors provided by the `BufferedReader` class that can be used to create an instance. The following table lists the constructors of the `BufferedReader` class.

Constructor	***Description***
`BufferedReader(Reader in)`	*It creates an instance of `BufferedReader` that uses a default-sized input buffer.*
`BufferedReader(Reader in, int sz)`	*It creates an instance of `BufferedReader` that uses an input buffer of the specified size.*

The Constructors of the BufferedReader Class

The following table lists the commonly used methods of the `BufferedReader` class with their description.

Method	***Description***
`void mark(int readAheadLimit)`	*Is used to mark the present position in the stream. Subsequent calls to `reset()` will attempt to reposition the stream to this point.*
`boolean markSupported()`	*Is used to tell whether this stream supports the `mark()` operation.*
`String readLine()`	*Is used to read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.*

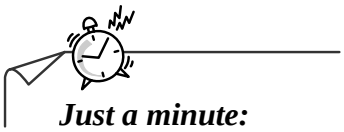
Method	*Description*
boolean ready()	*Is used to tell whether this stream is ready to be read. A buffered character stream is ready if the buffer is not empty, or if the underlying character stream is ready.*

The Methods of the BufferedReader Class

Consider the following code that accepts two numbers from the user by using the `BufferedReader` class and prints the sum of these two numbers:

```
import java.io.*;

public class BufferedReaderDemo
{
    public static void main(String args[]) throws IOException
    {
        try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))){

        System.out.println("Enter First number");
        String s = br.readLine();
        System.out.println("Enter Second number");
        String s1 = br.readLine();
        int i = Integer.parseInt(s);
        int i1 = Integer.parseInt(s1);
        int i3 = i + i1;
        System.out.println("Sum=" + i3);
        }
    }
}
```

In the preceding code, an object of the `BufferedReaderDemo` class has been created that wraps an `InputStreamReader` class to read data from the console. The `readLine()` method accepts the input from the user and stores it in the string variables, `s` and `s1`, respectively. Thereafter, the string value is converted into the integer value by using the `parseInt()` method, which is the static method of the `Integer` class. Further, the `System.out.println("Sum=" + i3);` statement prints the output.



Just a minute:

Which one of the following methods is used to ensure that the `close()` *method of the* `FileInpuStream` *class is called when there are no more references to it?*

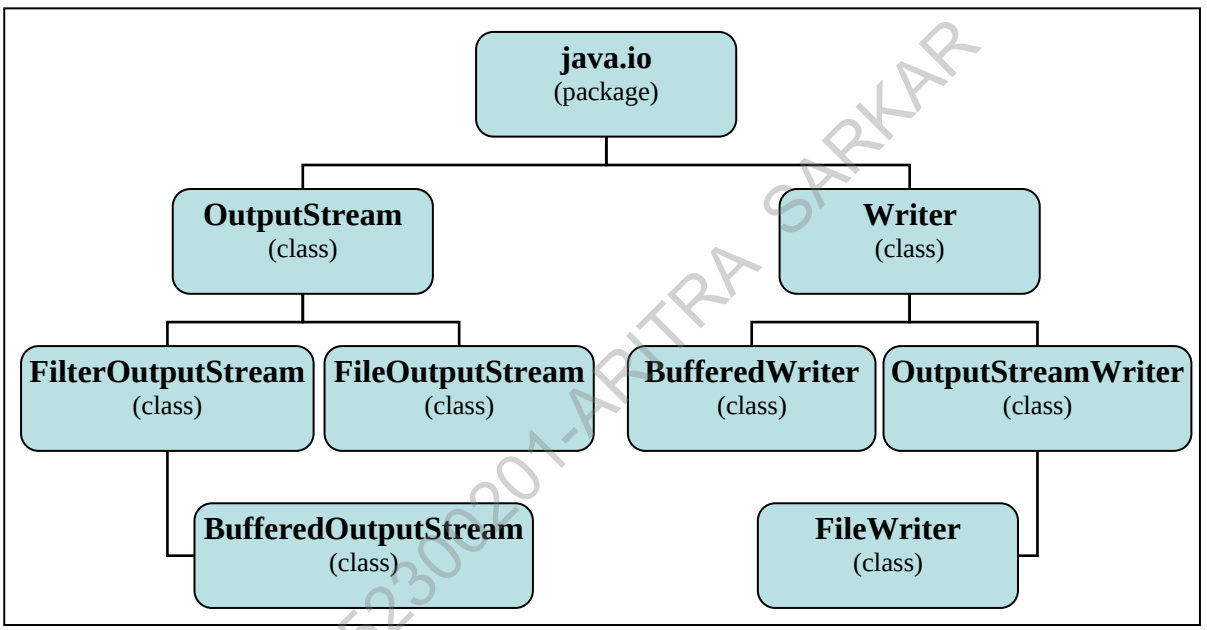
1. `finalize()`
2. `ready()`
3. `reset()`
4. `close()`

Answer:

1. `finalize()`

Working with Output Stream

Consider a scenario where you need to develop a Banking application. In this application, you need to implement a set of functionality to maintain a log for every transaction performed in the application. For this, you need to write the data to an output stream, such as a file. The data can be written in the form of bytes or characters to these streams. To write the data in the form of bytes, Java provides the `OutputStream` classes. To write the data in the form of characters, Java provides the `Writer` classes inside the `java.io` package. The following figure shows the class hierarchy of the `java.io` package.



The Output Stream Class Hierarchy

The `FileOutputStream` and `FilterOutputStream` classes are the subclasses of the `OutputStream` class. The `BufferedOutputStream` class is subclass of the `FilterOutputStream` class. The `BufferedWriter` and `OutputStreamWriter` classes are the subclasses of the `Writer` class. The `FileWriter` class is the subclass of the `OutputStreamWriter` class.

Using the FileOutputStream Class

The *`FileOutputStream` class* is used for writing data, byte by byte, to a file. The instance creation of `FileOutputStream` class is not dependent on a file on which the writing needs to be done. If the file does not exist, then it will create a file before opening it for the output operation. If the file is in the read-only mode, then it will throw an error. The `FileOutputStream` class provides various constructors that can be used to create an instance.

The following table lists the constructors of the `FileOutputStream` class.

Constructor	***Description***
`FileOutputStream(File file, boolean append)`	*It creates an instance of `FileOutputStream` that writes the file represented by the specified `File` object. If the `append` parameter is `true`, then bytes will be written at the end of the file. Otherwise, bytes will be written at the beginning.*
`FileOutputStream(FileDescriptor fdObj)`	*It creates an instance of `FileOutputStream` that writes to the file descriptor, which represents an existing connection with the actual file.*
`FileOutputStream(String name)`	*It creates an instance of `FileOutputStream` that writes to the file with the name specified by the `name` parameter.*
`FileOutputStream(String name, boolean append)`	*It creates an instance of `FileOutputStream` that writes to the file with the name specified in the `name` parameter. If the `append` parameter is `true`, then bytes will be written at the end of the file. Otherwise, bytes will be written at the beginning.*

The Constructors of the FileOutputStream Class

The following table lists the commonly used methods of the `FileOutputStream` class with their description.

Method	***Description***
`FileDescriptor getFD()`	*Is used to return the file descriptor associated with this stream.*
`void write(int b)`	*Is used to write the specified byte to this file output stream.*

The Methods of the FileOutputStream Class

Consider the following code to write the data to a file using the `FileOutputStream` class:

```
import java.io.*;
public class FileOutputStreamDemo
{
   public static void main(String[] args) throws IOException
   {
          boolean bool;
          long pos;
```

```
            String s = "This is a FileOutputStream Program";
            byte buf[] = s.getBytes();
            try (FileOutputStream fos = new
        FileOutputStream("D:\\Files\\File.txt "))
            {
                    for (int i = 0; i < buf.length; i++)
                    {
                            fos.write(buf[i]);
                    }
            }
            catch (Exception e)
            {
                    System.out.println(e);
            }
    }
}
```

In the preceding code, the `FileOutputStreamDemo` class has been created. The `fos.write(buf[i]);` statement writes the data into the file.

Using the BufferedOutputStream Class

The *`BufferedOutputStream` class* writes bytes to an output stream using a buffer for increased efficiency. The `BufferedOutputStream` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `BufferedOutputStream` class.

Constructor	***Description***
`BufferedOutputStream(OutputStream out)`	*It creates an instance of the buffered output stream that writes data to the specified output stream.*
`BufferedOutputStream(OutputStream out, int size)`	*It creates an instance of the buffered output stream to write data to the specified output stream with the specified buffer size.*

The Constructors of the BufferedOutputStream Class

The following table lists the commonly used methods of the `BufferedOutputStream` class with their description.

Method	***Description***
`void flush()`	*Is used to flush the buffered output stream. This forces any buffered output bytes to be written out to the output stream.*
`void write(byte[] b, int off, int len)`	*Is used to write `len` bytes from the byte array. It starts writing from the offset value, `off`, to the buffered output stream.*

The Methods of the BufferedOutputStream Class

Consider the following code of the `BufferedOutputStream` class that prints the output on the console:

```
import java.io.BufferedOutputStream;
import java.io.IOException;

public class BufferedOutputStream
{
    public static void main(String[] args)
    {
        try (BufferedOutputStream b = new BufferedOutputStream(System.out))
        {
            String s = "This is a BufferedOutputStream Demo Program";
            byte buf[] = s.getBytes();

            b.write(buf);
            b.flush();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the `BufferedOutputStream` class has been created. The `b.write(buf);` statement prints the output on the console.

Using the BufferedWriter Class

The *BufferedWriter class* can be used to write text to an output stream. This class writes relatively large chunks of data to an output stream at once. The `BufferedWriter` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `BufferedWriter` class.

Constructor	***Description***
`BufferedWriter(Writer out)`	*It creates an instance of `BufferedWriter` that uses a default-sized output buffer.*
`BufferedWriter(Writer out, int size)`	*It creates an instance of `BufferedWriter` that uses an output buffer of the given size.*

The Constructors of the BufferedWriter Class

The following table lists the commonly used methods of the `BufferedWriter` class with their description.

Method	***Description***
`void newLine()`	*Is used to write a line separator that is defined by the system property, `line.separator`.*
`void write(char[] cbuf, int off, int len)`	*Is used to write a portion of an array of characters.*

The Methods of the BufferedWriter Class

Consider the following code to write the data to the console using the `BufferedWriter` class:

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class BufferedWriterDemo
{
    public static void main(String args[])
    {
        try (BufferedWriter b = new BufferedWriter(new
OutputStreamWriter(System.out)))
        {
            String fruit[] = {"Apple", "Banana", "Grapes"};
            b.write("Different types of fruit are:" + "\n");
            for (int i = 0; i < 3; i++)
            {
                b.write(fruit[i] + "\n");
                b.flush();
            }
        }
```

```
            catch (IOException e)
            {
                System.out.println(e);
            }
        }
    }
```

The output of the preceding code is:

```
Different types of fruit are:
Apple
Banana
Grapes
```

In the preceding code, an array of fruits has been created. Then, an object of `BufferedWriter` has been created that wraps the `OutputStreamWriter` class to write data to the console output stream. The `write()` method writes the string to the console.

Using the FileWriter Class

The *`FileWriter` class* writes character data to a file. This class does not define any methods of its own. It derives all methods from its base classes, such as the `Writer` and the `OutputStreamWriter` classes.

The `FileWriter` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `FileWriter` class.

Constructor	***Description***
`FileWriter(File file)`	*It creates an instance of the `FileWriter` object from a `File` object.*
`FileWriter(File file,boolean append)`	*It creates an instance of the `FileWriter` object from a `File` object. If the second argument is `true`, then characters will be written at the end of the file. Otherwise, characters will be written at the beginning of the file.*
`FileWriter(FileDescriptor fd)`	*It creates an instance of the `FileWriter` object associated with a file descriptor.*
`FileWriter(String fileName,boolean append)`	*It creates an instance of the `FileWriter` object with a given file name. If the second argument is `true`, then characters will be written at the end of the file. Otherwise, characters will be written at the beginning of the file.*

The Constructors of the FileWriter Class

Consider the following code that writes the data to the file:

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

class FileWriterDemo
{
    public static void main(String args[])
    {
        try (FileWriter f = new FileWriter("D:\\Files\\file.txt "))
        {
            String source = "This is FileWriter Program";
            char buffer[] = new char[source.length()];
            source.getChars(0, source.length(), buffer, 0);
            f.write(buffer);
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the `FileWriterDemo` class has been created that writes data to the `file.txt` file. The `f.write(buffer);` statement writes the data to the file.

Just a minute:

Which one of the following `FileWriter` constructors will you use if there is a requirement to append data at the end of a new file?

1. `FileWriter(File file)`
2. `FileWriter(File file,boolean append)`
3. `FileWriter(FileDescriptor fd)`
4. `FileWriter(String fileName,boolean append)`

Answer:

4. `FileWriter(String fileName,boolean append)`

Activity 8.1: Working with Input Stream and Output Stream

Practice Questions

1. Which one of the following constructors of the `FileOutputStream` class is used to write bytes at the end of a given file?
   a. `FileOutputStream(File file,boolean append)`
   b. `FileOutputStream(FileDescriptor fdObj)`
   c. `FileOutputStream(File file)`
   d. `FileOutputStream(String name,boolean append)`

2. State whether the following statement is true or false.
   The `markSupported()` method is used to test whether the input stream supports the `mark()` and `reset()` methods.

3. Which one of the following methods is used to return the estimate number of bytes that can be read from an input stream?
   a. `available()`
   b. `reset()`
   c. `read()`
   d. `ready()`

4. Which one of the following classes does not have its own methods and inherits the methods of the `InputStreamWriter` class?
   a. `FileWriter`
   b. `BufferedWriter`
   c. `BufferedOutputStream`
   d. `FileOutputStream`

5. Which one of the following classes does not have its own methods and inherits the methods of the `InputStreamReader` class?
   a. `FileReader`
   b. `BufferedReader`
   c. `BufferedInputStream`
   d. `FileInputStream`

Summary

In this chapter, you learned that:

- Java handles all the input and output operations in the form of streams that act as a sequence of bytes or characters traveling from a source to a destination.
- When a stream of data is being sent, it is said to be written; and when a stream of data is being received, it is said to be read.
- To read data in the form of characters, Java provides the `Reader` classes inside the `java.io` package.
- The `FileInputStream` class is used to read data and the steams of bytes from the file.
- The `BufferedInputStream` class is used to perform the read operations by using a temporary storage, buffer, in the memory.
- The `FileReader` class is used for reading characters from a file, but it does not define any method of its own.
- The `BufferedReader` class is used to read the text from a character-input stream, such as a file, console, and array, while buffering characters.
- To write the data in the form of bytes, Java provides the `OutputStream` classes.
- To write the data in the form of characters, Java provides the `Writer` classes inside the `java.io` package.
- `FileOutputStream` is used for writing data, byte by byte, to a file.
- The `BufferedOutputStream` class writes bytes to an output stream using a buffer for increased efficiency.
- The `BufferedWriter` class can be used to write text to an output stream.
- The `FileWriter` class writes character data to a file. This class does not define any methods of its own.

Solutions to Practice Questions

Chapter 1

1. b. .java
2. a. `synchronized`
3. b. `native`
4. b. `public`
5. b. character

Chapter 2

1. c. Modulus operator
2. b. `instanceof` operator
3. a. Logical AND
4. c. `3`
5. c. `%`

Chapter 3

1. True
2. `False`
3. d. `for`
4. True
5. `16 14`

Chapter 4

1. d. `int[ ] list = new int[10];`
2. b. `2,4,6,8,10`
3. a. `equals()`
4. b. `toCharArray()`

Chapter 5

1. c. Abstract class
2. a. `implements`
3. b. Multilevel inheritance
4. d. Hierarchical inheritance
5. a. Polymorphism

Chapter 6

1. b. `Throwable`
2. a. Checked exception
3. c. `InstantiationException`
4. c. `assert`
5. a. `try`

Chapter 7

1. True
2. a. `String pattern()`
3. b. Reluctant
4. b. `[b-z&&[^bcd]]`
5. a. Greedy

Chapter 8

1. a. `FileOutputStream(File file,boolean append)`
2. True
3. a. `available()`
4. a. `FileWriter`
5. a. `FileReader`

Glossary

A

Abstract class

An abstract class is a class that contains one or more abstract methods.

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations on operands.

Array

An array is a collection of elements of a single data type stored in adjacent memory locations.

Assertions

Assertions are statements in Java that enable you to test any assumptions that you make regarding a program during its execution.

B

Bitwise operators

Bitwise operators are used for the manipulation of data at the bit level.

break

The break statement stops the execution of the remaining statements.

Bytecode

In Java, when you compile an error-free code, the compiler converts the program to a platform independent code, known as the bytecode.

C

Checked exceptions

These are the invalid conditions that occur in a Java program due to the problems, such as accessing a file that does not exist or referring a class that does not exist.

Comparison operators

Comparison operators are used to compare two values and perform an action on the basis of the result of that comparison.

Constructors

Constructors are methods that have the same name as that of the class and have no return type.

continue

The `continue` statement skips all the statements following the continue statement and moves the control back to the loop statement.

E

Enum

An enum is used to define a fixed set of constants.

Error

An error is a condition that causes disruption in the execution of a program.

Event handler

An event handler is a method that is invoked by an event listener when the event is generated.

Event listener

An event listener listens for a specific event and gets notified when the specific event occurs.

Event source

An event source is an object that generates an event.

Exception

The term, exception, in Java indicates an abnormal event that occurs during the program execution and disrupts the normal flow of instructions.

G

Garbage collection

The process of automatically deallocating memory is called garbage collection.

I

if

The `if` construct executes statements based on the specified condition.

if-else

The `if-else` construct executes statements based on the specified condition.

Inheritance

The process of creating a new class by acquiring some features from an existing class is known as inheritance.

instanceof

The `instanceof` operator is used to test whether an object is an instance of a specific class at runtime or not.

J

Java Virtual Machine

Java Virtual Machine interprets the bytecode into the machine code that can be run on that machine.

JIT compiler

The JIT compiler compiles the bytecode into machine executable code, as and when needed, which optimizes the performance of a Java program.

K

Keywords

Keywords are the reserved words with a special meaning for a language, which express the language features.

L

Logical operators

Logical operators are used to evaluate operands and return a boolean value.

M

Method

A method is a set of statements that is intended to perform a specific task.

Method overriding

Method overriding enables a subclass to provide its own implementation of a method that already has an implementation defined in its superclass.

Multidimensional arrays

Multidimensional arrays are arrays of arrays.

O

Object

An object is an instance of a class and has a unique identity.

One-dimensional array

A one-dimensional array is a collection of elements with a single index value.

Operators

Operators allow you to perform various mathematical and logical calculations, such as adding and comparing numbers.

P

Polymorphism

Polymorphism is the ability to redefine a function in more than one form.

S

Shift operator

A shift operator is used to shift the bits of its operand either to the left or to the right.

Subclass

In inheritance, the class that inherits the data members and methods from another class is known as the subclass or derived class.

Superclass

The class from which the subclass inherits the features is known as the superclass or base class.

switch

The `switch` construct evaluates an expression for multiple values.

T

Ternary operator

The ternary operator works on a logical expression and two operands. It returns one of the two operands depending on the result of the expression.

Thread

A thread is the smallest unit of execution in a Java program.

U

Unary operator

An operator that requires one operand is called a unary operator.

Unchecked exceptions

The unchecked exceptions occur because of programming.

NIIT

Appendix

Case Study 1: Hangman Game

ProGame Inc. is an emerging game development organization based in the US. The office of the organization is located in Santa Monica, California. The total employee count of the organization is 55. Recently, the business has seen growth and the organization has gained new clientele. As a new client deal, the organization has gained a project to develop a game console, which has games, such as Hangman and Scramble. The management has assigned the responsibility of developing the game console to Peter, the Senior Programmer. The management wants the game code to be simple. In addition, the game must be robust and compatible across various operating systems, such as Windows and Linux.

Initially, Peter decides to develop the game console interface and the Hangman game. The game console interface should have the options to play different games and exit the game console. The Hangman game is a single player game. At the start of this game, the player should be presented with a menu. Using the menu, a player can decide whether to play the game, read the game instructions, or exit the game. If the player decides to play the game, a series of blank dashes needs to be displayed. The player will then have to correctly guess an alphabet for each blank displayed. After guessing all the blanks correctly, the player needs to be notified about the result and should also be asked whether to continue playing the game or not. If the player decides to continue, then another series of blank dashes need to be displayed, and this process should continue until the player decides to exit.

Case Study 2: EmployeeBook Application

Alchem Inc. is an emerging company in Singapore. The company provides software development services to its clients. The company has 35 diligent employees. Recently, the company has seen a huge growth in its clientele. Due to a large number of projects, the company has decided to hire around 100 employees in the present quarter and 50 employees in the next quarter. In order to organize the details of an ever rising count of employees in an efficient manner, the management has asked Shawn, the Head of the IT department, to create an application to store and update the employee details. The application should be robust and platform-independent as the company utilizes a Windows and Linux-based infrastructure.

The application should be menu-based. It must provide the functionality to store employee data that includes the employee ID, employee name, department, designation, date of joining, date of birth, and marital status. If the employee is married, the date of anniversary must also be stored. In addition, the application should provide the functionality to view the records.

Objectives Attainment Feedback

Introduction to Java

Name: _____ Batch: _____ Date: _____

The objectives of this course are listed below. Please tick whether the objectives were achieved by you or not. Calculate the percentage at the end, fill in your name and batch details, and return the form to your coordinator.

S. No.	*Objectives*	*Yes*	*No*
1.	Get familiar with Java	☐	☐
2.	Identify the building blocks of a Java program	☐	☐
3.	Access class members	☐	☐
4.	Work with operators	☐	☐
5.	Use operator precedence	☐	☐
6.	Work with conditional constructs	☐	☐
7.	Work with loop constructs	☐	☐
8.	Manipulate arrays	☐	☐
9.	Manipulate enums	☐	☐
10.	Manipulate strings	☐	☐
11.	Implement inheritance	☐	☐

S. No.	*Objectives*	*Yes*	*No*
12.	Implement polymorphism		
13.	Handle exceptions		
14.	Use the assert keyword		
15.	Process strings using regex		
16.	Work with input stream		
17.	Work with output stream		
	Percentage: (# of Yes/17) * 100		

NIIT