

**MASTER OF COMPUTER APPLICATIONS**

**MCA-13**

**JAVA PROGRAMMING**



**Directorate of Distance Education  
Guru Jambheshwar University of  
Science & Technology  
Hisar - 125001**

## CONTENTS

<b>1.</b>	<b>Introduction to JAVA</b>	<b>1-30</b>
<b>2.</b>	<b>Data Types and Operators</b>	<b>31-56</b>
<b>3.</b>	<b>Control Structures and Looping</b>	<b>57-74</b>
<b>4.</b>	<b>Inheritance and Polymorphism</b>	<b>75-107</b>
<b>5.</b>	<b>Multithreaded Programming</b>	<b>108-142</b>
<b>6.</b>	<b>Exception Handling</b>	<b>143-168</b>
<b>7.</b>	<b>File Handling</b>	<b>169-187</b>
<b>8.</b>	<b>GUI Programming</b>	<b>188-206</b>

SUBJECT: JAVA PROGRAMMING	
COURSE CODE: MCA-13	AUTHOR: AYUSH SHARMA
LESSON NO. 1	
INTRODUCTION TO JAVA	

## **STRUCTURE**

- 1.0 Learning Objective
- 1.1 Introduction
- 1.2 Object-Oriented Paradigm
- 1.3 Basic Concepts of Object-Oriented Programming
  - 1.3.1 Objects and Classes
  - 1.3.2 Data Abstraction and Encapsulation
  - 1.3.3 Inheritance
  - 1.3.4 Polymorphism
  - 1.3.5 Dynamic Binding
  - 1.3.6 Message Communication
- 1.4 Benefits of Object-Oriented Programming
- 1.5 Applications of Object-Oriented Programming
- 1.6 Java Evolution
  - 1.6.1 Java History
- 1.7 Features of Java
  - 1.7.1 Simple
  - 1.7.2 Object-Oriented
  - 1.7.3 Robust
  - 1.7.4 Secure
  - 1.7.5 Multithreaded
  - 1.7.6 Architecture-Neutral
  - 1.7.7 Portable
  - 1.7.8 Distributed
  - 1.7.9 Dynamic

- 1.7.10 Interpreted and High Performance
- 1.8 Java Run Time Environment
  - 1.8.1 Hardware and Software Requirements
  - 1.8.2 Java Support Systems
  - 1.8.3 Java Environment
    - 1.8.3.1 Java Development Kit
    - 1.8.3.2 Application Programming Interface
- 1.9 Comparison of Java and C++
- 1.10 Basic Java Program
  - 1.10.1 Entering the Program
  - 1.10.2 Compiling the Program
  - 1.10.3 One Close look at sample program
- 1.11 Check Your Progress
- 1.12 Summary
- 1.13 Keywords
- 1.14 Self-Assessment Test
- 1.15 Answers to check your progress
- 1.16 References / Suggested Readings

---

## **1.0 LEARNING OBJECTIVE**

---

After going through this unit, you will be able to:

- Gain the concept of Java
- Differentiate Java as Object Oriented Language.
- Illustrate the basic features of Java.
- Know the Java environment.
- Know how to install Java SDK.

---

## 1.1 INTRODUCTION

---

The greatest challenges and most exciting opportunities for software developers today lie in tackling the power of networks. Most of the applications created today, will almost certainly be run on machines connected to the global networks i.e. Internet.

We know that by the mid 1990s, the World Wide Web had transformed the online world. Through a system of hypertext, users of the Web were able to select and view information from all over the world. However, while this system of hypertext gave users a high degree of selectivity over the information they chose to view, their level of interactivity with that information was low. Moreover, the Web lacked true interactivity—real-time, dynamic, and visual interaction between the user and application.

Sun Microsystems, a company best known for its high-end Unix workstations, developed a programming language named **Java** to create software that can run on many different kinds of devices. Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level.

Java brings this missing interactivity to the Web. With a Java-enabled Web browser, you can encounter animations and interactive applications. Java programmers can make customized media formats and information protocols that can be displayed in any Java-enabled browser. Java's features enrich the communication, information, and interaction on the Web by enabling users to distribute executable content—rather than just HTML pages and multimedia files—to users. This ability to distribute executable content is the power of Java.

In this unit, we will introduce you to the Java programming language. We will discuss the basic features of Java and how to install the Java Development Kit. We will also discuss how to write a Java program and the procedure for compiling and running a Java program.

---

## 1.2 Object-Oriented Paradigm

---

Java is a true object-oriented language. Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input

and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions.



**Fig. 1.1 Object = Data + Methods**

C was developed to meet general system programming needs. It was quickly adapted for general use and found widespread acceptance. C is a high-level procedural language that has many low-level features. These features help to make it versatile and efficient. However, many of these features give it a reputation for being cryptic and hard to maintain. C++ extends the C language to provide object-oriented features. The language is backward compatible with C, and code from the two languages can be used with each other with little difficulty. C++ has found quick acceptance and is supported by a number of pre-built specialized classes.

Java can be considered the third generation of the C/C++ family. It is not backward compatible with C/C++ but was designed to be very similar to these languages. The creators of Java intentionally left out some of the features of C/C++ that have been problematic for programmers. Java is strongly object-oriented. In fact, one cannot create Java code that is not object-oriented. Java's portability is a key advantage and is the reason why Java is often used for Web development.

---

### **1.3 Basic Concepts of Object-Oriented Programming**

---

Java, the programming language, was introduced by Sun Microsystems. This work was initiated by James Gosling and the final version of Java was released in the year 1995. However, initially Java was released as a component of the core Sun Microsystem platform for Java called J2SE or Java 1.0. The latest release of Java or J2SE is Java Standard Version 6.

The rising popularity of Java, as a programming platform and language has led to the development of several tools and configurations, which are made keeping Java in mind. For instance, the J2ME and J2EE are two such configurations. The latest versions of Java are called Java SE and Java EE or Java ME instead of J2SE, J2EE and J2ME. The biggest advantage of using the Java platform is the fact that it allows you to run your code at any machine. So, you just need to write your code once and expect it to run everywhere.

As far as the features of Java are concerned, they are as follows:

### 1.3.1 Objects and Classes

**Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

**Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

#### Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

#### Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

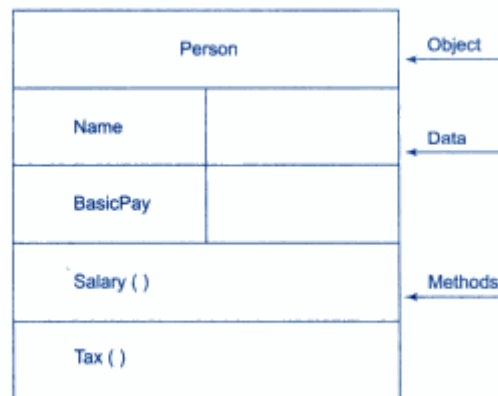
Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;
```

```

void barking() {
}
void hungry() {
}
void sleeping() {
}
}

```



**Figure 1.2 Representation of an Object**

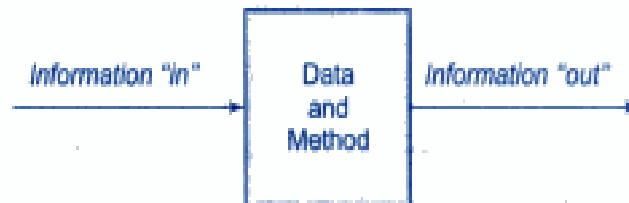
### 1.3.2 Data Abstraction and Encapsulation

An essential element of object-oriented programming is ***abstraction***. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

***Encapsulation*** is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world,



consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever.

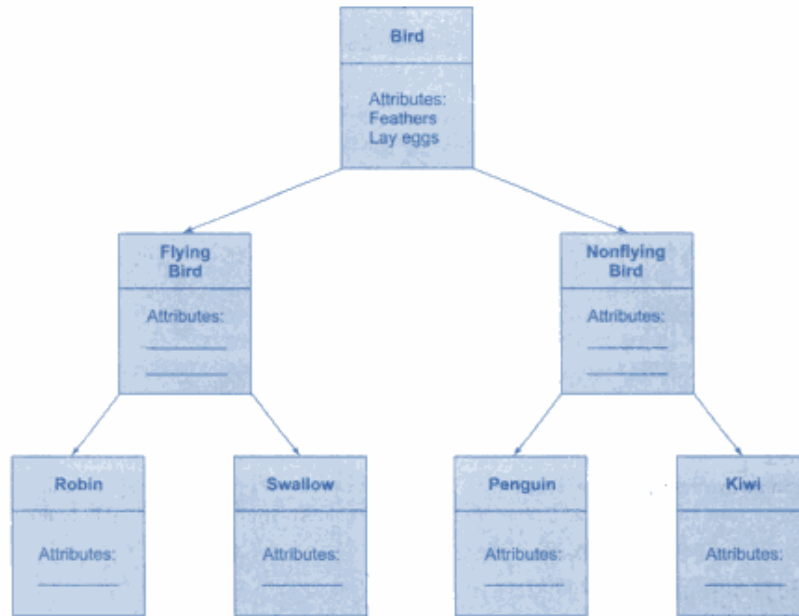


**Fig. 1.3. Encapsulation -Objects as “black boxes”**

### 1.3.3 Inheritance

Reusability is one of the important feature of object-oriented programming and it can be achieved through *inheritance*. Java supports the concepts of inheritance. With the use of inheritance, the information is made manageable in a hierarchical order. Inheritance can be defined as the process where one object acquires the properties of another. When we want to create a new class and there is already a class that includes some of the code that we want, we can derive the new class from the existing class. In doing this, we can reuse the fields and methods of the existing class without rewriting them again.

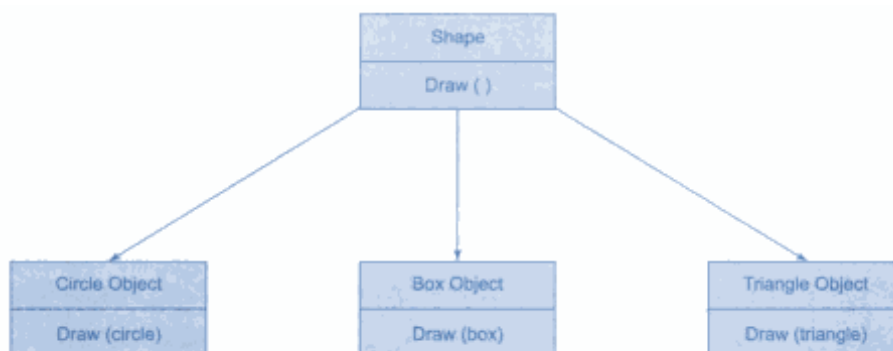
A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*). Constructors cannot be inherited by subclasses, but the constructor of the superclass can be invoked from the subclass. In Java, inheritance is implemented by the process of extension. To define a new class as an extension of an existing class, we simply use an *extend* clause in the header of the new classes definition. The concept of inheritance is used to make the things from general to more specific.



**Fig. 1.4. Property Inheritance**

### 1.3.4 Polymorphism

*Polymorphism* (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floatingpoint values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.



**Fig. 1.5. Polymorphism**

### 1.3.4 Dynamic Binding

In dynamic binding, the method call is bonded to the method body at runtime. This is also known as late binding. This is done using instance methods.

#### Example

```
class Super {
    public void sample() {
        System.out.println("This is the method of super class");
    }
}

Public class extends Super {
    Public static void sample() {
        System.out.println("This is the method of sub class");
    }

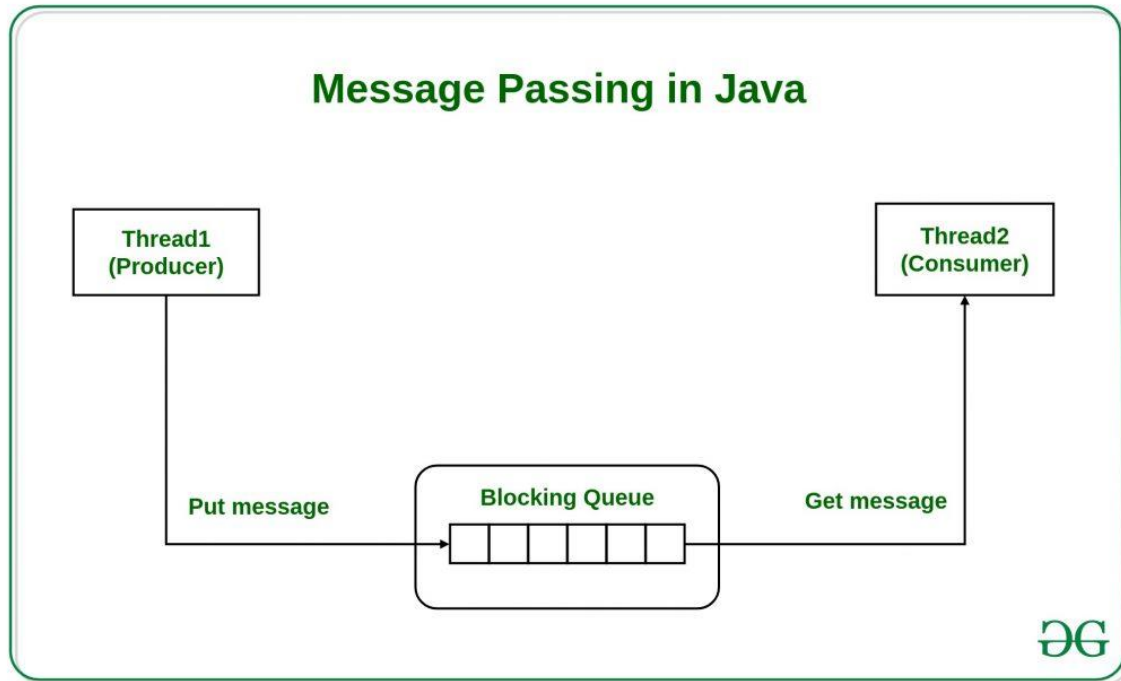
    Public static void main(String args[]) {
        new Sub().sample()
    }
}
```

#### Output

This is the method of sub class

### 1.3.5 Message Communication

Message Passing in terms of computers is communication between processes. It is a form of communication used in object-oriented programming as well as parallel programming. Message passing in Java is like sending an object i.e. message from one thread to another thread. It is used when threads do not have shared memory and are unable to share monitors or semaphores or any other shared variables to communicate. Suppose we consider an example of producer and consumer, likewise what producer will produce, the consumer will be able to consume that only. We mostly use **Queue** to implement communication between threads.



**Fig. 1.6 Message Passing**

A message for an object is a request for execution of a procedure, and therefore will invoke a method (procedure) in the receiving object that generates the desired result, as shown is Fig. 1.7.



**Fig. 1.7. Message triggers a method**

Message passing involves specifying the name of the object, the name of the method (message) and the information to be sent. For example, consider the statement

`Employee.salary (name);`

Here, Employee is the object, salary is the message and name is the parameter that contains information.



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

---

## 1.4 Benefits of Object-Oriented Programming

---

Following are the benefits of object-oriented programming:

1. **Simplicity:** software objects model real world objects, so the complexity is reduced and the program structure is very clear;
2. **Modularity:** each object forms a separate entity whose internal workings are decoupled from other parts of the system;
3. **Modifiability:** it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods;
4. **Extensibility:** adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones;
5. **Maintainability:** objects can be maintained separately, making locating and fixing problems easier;
6. **Re-usability:** objects can be reused in different programs

---

## 1.5 Applications of Object-Oriented Programming

---

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

---

## 1.6 Java Evolution

---

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend()**, **resume()**, and **stop()** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**,

chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language.

### **1.6.1 JAVA HISTORY**

James Gosling started working on the Java programming language in June 1991 for utilization in one of his numerous set-top box ventures. The programming language, at first, was called Oak. This name was kept after an oak tree that remained outside Gosling's office. This name was changed to the name Green and later renamed as Java, from a list of words, randomly picked from the dictionary. Sun discharged the first open usage as Java 1.0 in 1995. It guaranteed Write Once, Run Anywhere (WORA), giving no-expense run-times on prominent stages. On 13 November 2006, Sun discharged much of Java as free and open source under the terms of the GNU General Public License (GPL). On 8 May 2007, Sun completed the procedure, making the greater part of Java's center code free and open-source, beside a little parcel of code to which Sun did not hold the copyright.

**Table 1.1 Java Milestones**

<i>Year</i>	<i>Development</i>
1990	Sun Microsystems decided to develop special software that could be used to manipulate consume electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
1991	After exploring the possibility of using the most popular object-oriented language C++, the team announced a new language named "Oak".
1992	The team, known as Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
1993	The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
1994	The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.
1995	Oak was renamed "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
1996	Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
1997	Sun releases Java Development Kit 1.1 (JDK 1.1).
1998	Sun releases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2).
1999	Sun releases Java 2 Platform, Standard Edition (J2SE) and Enterprise Edition (J2EE).
2000	J2SE with SDK 1.3 was released.
2002	J2SE with SDK 1.4 was released.
2004	J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

## **1.7 Features of Java**

The Java team has summed up the basic features of Java with the following list of buzzwords:

### **1.7.1 Simple**

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.



### **1.7.2 Object-oriented**

Java is a true object-oriented language. Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions.

### **1.7.3 Robust**

Java is a robust language. The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

### **1.7.4 Secure**

Prior to Java, most users did not download executable programs frequently from Internet, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a "firewall" between a networked application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

### **1.7.5 Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

### **1.7.6 Architecture-neutral**

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

### **1.7.7 Portable**

In addition to being architecture-neutral, Java code is also portable. It was an important design goal of Java that it be portable so that as new architectures (due to hardware, operating system, or both) are Java and the runtime environment is written in POSIX-compliant C.

### **1.7.8 Distributed**

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-address space messaging. This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called *Remote Method Invocation (RMI)*. This feature brings an unparalleled level of abstraction to client/server programming.

### **1.7.9. Dynamic**

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

### **1.7.10 Interpreted and High Performance**

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

---

## **1.8 Java Running Environment**

---

Java is strongly associated with the internet because of the first application program written in Java was hot Java. Web browsers to run applets on internet. Internet users can use Java to create applet programs & run then locally using a Java-enabled browser such as hot Java. Java applets have made the internet a true extension of the storage system of the local computer.

### **1.8.1 Hardware and Software Requirements**

#### **Hardware Requirement for Java**

Minimum hardware requirement to download Java on your Windows operating system as follows:

- Minimum Windows 95 software
- IBM-compatible 486 system

- Hard Drive and Minimum of 8 MB memory
- A CD-ROM drive
- Mouse, keyboard and sound card, if required.

## Software requirement for Java

Nowadays, Java is supported by almost every operating systems. whether it is a Windows, Macintosh and Unix all supports the Java application development. So you can download any of the operating system on your personal computer. Here are the minimum requirement.

- Operating System
- Java SDK or JRE 1.6 or higher
- Java Servlet Container (Free Servlet Container available)
- Supported Database and library that supports the database connection with Java.

### 1.8.2 Java Support Systems

It is clear from the discussion we had up to now that the operation of Java and Java-enabled browsers on the Internet requires a variety of support systems. Table 1.1 lists the systems necessary to support Java for delivering information on the internet.

**Table 1.2 Java Support Systems**

<i>Support System</i>	<i>Description</i>
Internet Connection	Local computer should be connected to the Internet.
Web Server	A program that accepts requests for information and sends the required documents.
Web Browser	A program that provides access to WWW and runs Java applets.
HTML	A language for creating hypertext for the Web.
APPLET Tag	For placing Java applets in HTML document.
Java Code	Java code is used for defining Java applets.
Bytecode	Compiled Java code that is referred to in the APPLET tag and transferred to the user computer.

### 1.8.3 Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as Java development Kit (JDK) and the classes and methods are part of the Java Standard Library (JSL), also known as the Application Programming Interface (API).

### 1.8.3.1 Java development Kit

The Java development Kit comes with a collection of tools that are used for development and running Java programs. Some of them are:

**Java** The loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler.

**Javac** The compiler, which converts source code into Java bytecode

**Jar** The archiver, which packages related class libraries into a single JAR file.

**Javadoc** The documentation generator, which automatically generates documentation from source code comments

**Jdb** The Java debugger

**Jps** The process status tool, which displays process information for current Java processes

**Javap** The class file disassembler

**Appletviewer** This tool can be used to run and debug Java applets without a web browser.

**Javah** The C header and stub generator, used to write native methods

An application programming interface (API) is an interface implemented by a software program to enable interaction with other software, similar to the way a user interface facilitates interaction between humans and computers. Java APIs include hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

- Language support package
- Utility package
- Input/Output Package
- Networking Package
- AWT(Abstract Window Tool Kit) Package
- Applet Package

### 1.8.3.2 Application Programming Interface

An application programming interface (API) is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc. It can also provide extension mechanisms so that users can extend existing functionality in various ways and to varying degrees.[1] An API can be entirely

custom, specific to a component, or it can be designed based on an industry-standard to ensure interoperability. Through information hiding, APIs enable modular programming, which allows users to use the interface independently of the implementation.

## 1.9 Comparison of Java and C++

C was developed to meet general system programming needs. It was quickly adapted for general use and found widespread acceptance. C is a high-level procedural language that has many low-level features. These features help to make it versatile and efficient. However, many of these features give it a reputation for being cryptic and hard to maintain. C++ extends the C language to provide object-oriented features. The language is backward compatible with C, and code from the two languages can be used with each other with little difficulty. C++ has found quick acceptance and is supported by a number of pre-built specialized classes.

Java can be considered the third generation of the C/C++ family. It is not backward compatible with C/C++ but was designed to be very similar to these languages. The creators of Java intentionally left out some of the features of C/C++ that have been problematic for programmers. Java is strongly object-oriented. In fact, one cannot create Java code that is not object-oriented. Java's portability is a key advantage and is the reason why Java is often used for Web development.

**C++ vs Java :** A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
<b>Platform-independent</b>	C++ is platform-dependent.	Java is platform-independent.
<b>Mainly used for</b>	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
<b>Design Goal</b>	C++ was designed for systems and applications programming. It	Java was designed and created as an interpreter for printing systems but

	was an extension of C programming language.	later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
<b>Goto</b>	C++ supports the goto statement.	Java doesn't support the goto statement.
<b>Multiple inheritance</b>	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
<b>Operator Overloading</b>	C++ supports operator overloading.	Java doesn't support operator overloading.
<b>Pointers</b>	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
<b>Compiler and Interpreter</b>	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.
<b>Call by Value and Call by reference</b>	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
<b>Structure and Union</b>	C++ supports structures and unions.	Java doesn't support structures and unions.

<b>Thread Support</b>	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
<b>Documentation comment</b>	C++ doesn't support documentation comment.	Java supports documentation comment ( <code>/** ... */</code> ) to create documentation for java source code.
<b>Virtual Keyword</b>	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
<b>unsigned right shift &gt;&gt;&gt;</b>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
<b>Inheritance Tree</b>	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the inheritance tree in java.
<b>Hardware</b>	C++ is nearer to hardware.	Java is not so interactive with hardware.
<b>Object-oriented</b>	C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from <code>java.lang.Object</code> .



## 1.10 Basic Java Program

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
/*  
    This is a simple Java program.  
    Call this file "Example.java".  
*/  
  
class Example {  
    // Your program begins with a call to main().  
    public static void main(String args[]) {  
        System.out.println("This is a simple Java program.");  
    }  
}
```

### 1.10.1 Entering the Program

For most computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why.

In Java, a source file is officially called a *compilation unit*. It is a text file that contains (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the **.java** filename extension.

As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

### 1.10.2 Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, you must use the Java application launcher called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

This is a simple Java program.

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute **java** as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

### 1.10.3 A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features that are common to all Java programs. Let's closely examine each part of the program.

The program begins with the following lines:

```
/*
```

```
    This is a simple Java program.
```

```
    Call this file "Example.java".
```

```
*/
```

This is a *comment*. Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment*. This type of comment must begin with `/*` and end with `*/`. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace (`{`) and the closing curly brace (`}`). For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented. The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a `//` and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The next line of code is shown here:

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

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. The full meaning of each part of this line cannot be given now, since it involves a

detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access modifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But **java** has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, **java** would report an error because it would be unable to find the **main( )** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters*. If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the **{**. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

One other point: **main()** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main()** method to get things started. Furthermore, in some cases, you won't need **main()** at all. For example, when creating applets—Java programs that are embedded in web browsers—you won't use **main()** since the web browser uses a different means of starting the execution of applets. The next line of code is shown here. Notice that it occurs inside **main()**.

```
System.out.println("This is a simple Java program.");
```

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println()** method. In this case, **println()** displays the string which is passed to it. As you will see, **println()** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console. As you have probably guessed, console output (and input) is not used frequently in most real-world Java applications. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple utility programs, demonstration programs, and server-side code. Later in this book, you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods. Notice that the **println()** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

The first **}** in the program ends **main()**, and the last **}** ends the **Example** class definition.

---

## 1.11 Check Your Progress

---

1. Object is the combination of Data and \_\_\_\_\_
2. A class is a \_\_\_\_\_ that describes the behavior/state that the object of its type support.
3. \_\_\_\_\_ is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
4. \_\_\_\_\_ can be defined as the process where one object acquires the properties of another.

5. \_\_\_\_\_ is a feature that allows one interface to be used for a general class of actions.
6. In \_\_\_\_\_, the method call is bonded to the method body at runtime.
7. Objects can be reused in different programs; this feature is called \_\_\_\_\_
8. Java communicates with a Web page through a special tag called \_\_\_\_\_
9. Java Standard Library (JSL), also known as the \_\_\_\_\_
10. A Language used for creating hyperlinks is called \_\_\_\_\_

---

## 1.12 Summary

---

Java is a true object-oriented language. Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions.

The benefits of object-oriented programming: Simplicity, Modularity, Modifiability, Extensibility, Maintainability, Re-usability.

Features of Java are: Simple, Object-Oriented, Robust, Secure, Multithreaded, Architecture-Neutral, Portable, Distributed, Dynamic.

Minimum hardware requirement to download Java on your Windows operating system as follows:

- Minimum Windows 95 software
- IBM-compatible 486 system
- Hard Drive and Minimum of 8 MB memory
- A CD-ROM drive

- Mouse, keyboard and sound card, if required.

The minimum software requirement is:

- Operating System
- Java SDK or JRE 1.6 or higher
- Java Servlet Container (Free Servlet Container available)
- Supported Database and library that supports the database connection with Java.

## 1.13 Keywords

- 1 **Object**- An object is an instance of a class. Object have states and behaviors.
- 2 **Class**- A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.
- 3 **Encapsulation**- It is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- 4 **Inheritance**- Inheritance can be defined as the process where one object acquires the properties of another. When we want to create a new class and there is already a class that includes some of the code that we want, we can derive the new class from the existing class.
- 5 **Polymorphism**- Polymorphism is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
- 6 **Object-Oriented**- Java is a true object-oriented language. Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well.
- 7 **Robust**- Java is a robust language. The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.
- 8 **JDK**- JDK stands for Java Development Kit.

---

## 1.14 Self-Assessment Test

---

- Q.1 Describe any three basic features of Java programming language.
- Q.2. What are the benefits of OOP?
- Q.3. Describe the various applications of OOP.
- Q.4. Briefly describe History of Java.
- Q.5. Explain Data abstraction and Polymorphism concept.
- Q.6. Explain the similarities between C and C++.
- Q.7. Write down differences between C and C++

---

## 1.15 Answers to check your progress

---

- 1. Methods
- 2. Template/Blueprint
- 3. Encapsulation
- 4. Inheritance
- 5. Polymorphism
- 6. Dynamic Binding
- 7. Re-usability
- 8. <APPLET>
- 9. Application Programming Interface (API).
- 10. HTML (Hypertext Markup Language)

---

## 1.16 References / Suggested Readings

---

- 1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
- 2. Java- The Complete Reference by Herbert Schildt, ORACLE.
- 3. Java For Beginners by Scott Sanderson
- 4. Head First Java by Kathy Sierra & Bert Bates
- 5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
- 6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.



<b>SUBJECT: JAVA PROGRAMMING</b>	
<b>COURSE CODE: MCA-13</b>	<b>AUTHOR: AYUSH SHARMA</b>
<b>LESSON NO. 2</b>	
<b>Data Types and Operators</b>	

## **STRUCTURE**

- 2.0 Learning Objective
- 2.1 Introduction
- 2.2 Data Types
- 2.3 Java Tokens
  - 2.3.1 Keywords
  - 2.3.2 Identifiers
  - 2.3.3 Literals
    - 2.3.3.1 Integer Literals
    - 2.3.3.2 Floating-Point Literals
    - 2.3.3.3 Boolean Literals
    - 2.3.3.4 Character Literals
  - 2.3.4 Operators
  - 2.3.5 Separators
- 2.4 Operators
  - 2.4.1 Arithmetic Operators
  - 2.4.2 Relational Operators
  - 2.4.3 Logical Operators
  - 2.4.4 Assignment Operators
  - 2.4.5 Increment and Decrement Operators
  - 2.4.6 Bitwise Operators
  - 2.4.7 Special Ternary Operator
- 2.5 Precedence in Arithmetic Operators
- 2.6 Type Casting
  - 2.6.1 Java's Automatic Conversions

2.6.2	Casting Incompatible Types
2.7	Check Your Progress
2.8	Summary
2.9	Keywords
2.10	Self-Assessment Test
2.11	Answers to check your progress
2.12	References / Suggested Readings

---

## **2.0 LEARNING OBJECTIVE**

---

After going through this unit, you will be able to:

- Learn about Java tokens
- Learn about the variables, constants and data types in Java
- Declare and define variables in Java
- Learn about the various operators used in Java programming

---

## **2.1 INTRODUCTION**

---

The previous unit is an introductory unit where you have acquainted with the object-oriented features of the programming language Java. The installation procedure of Java SDK is also described in the unit. You have learnt how to write, save, compile and execute programs in Java from the previous unit.

In this unit we will discuss the basics of Java programming language which include tokens, variables, data types, constants etc. This might be a review for learners who have learnt the languages like C/C++ earlier. We extend this discussion by adding some new concepts associated with Java.

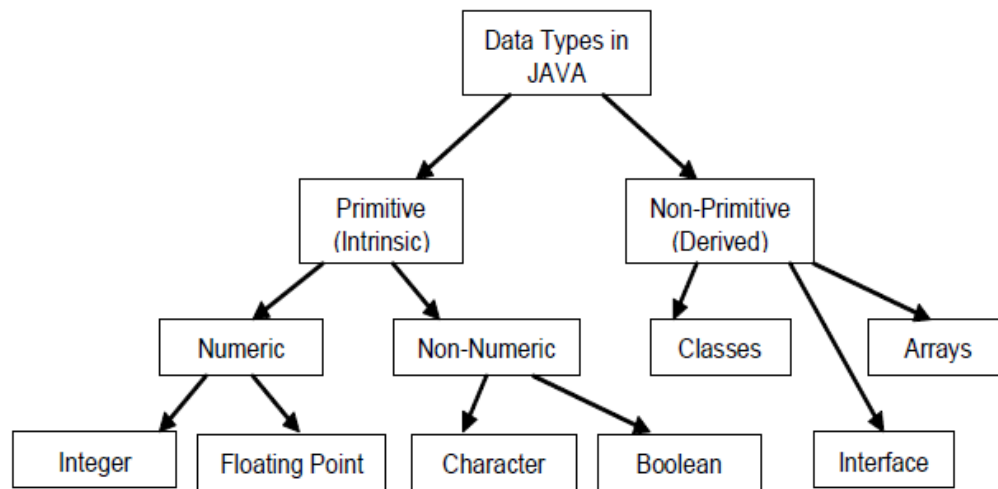
---

## **2.2 Data Types**

---

Every variable must have a data type. A data type determines the values that the variable can contain and the operations that can be performed on it. A variable's type also

determined how its value is stored in the computer's memory. The JAVA programming language has the following categories of data types (Fig 2.1):



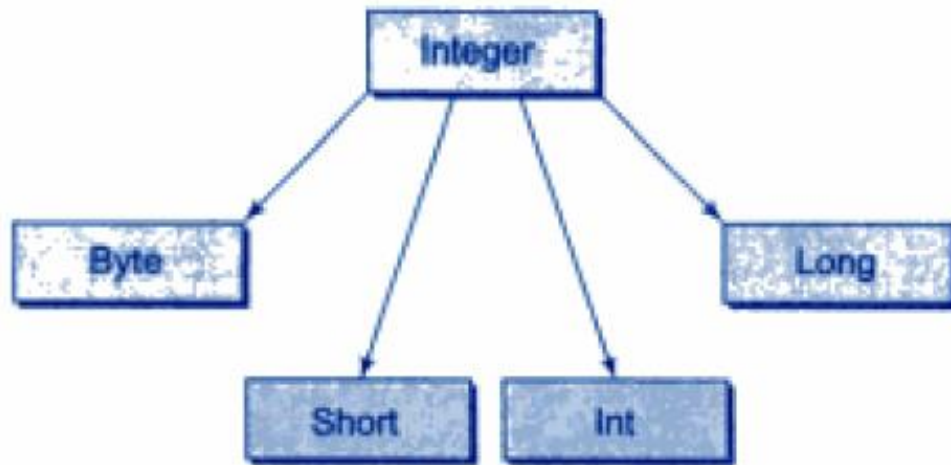
### Fig. 2.1 Data types in Java

A variable of ***primitive*** type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. Primitive types are also termed as *intrinsic* or *built-in* types. The primitive types are described below:

- **Integer type**

Integer type can hold whole numbers like 1,2, 3, ....-4, 1996 etc. Java supports four types of integer types: ***byte***, ***short***, ***int*** and ***long***. It does not support *unsigned* types and therefore all Java values are *signed* types. This means that they can be positive or negative.

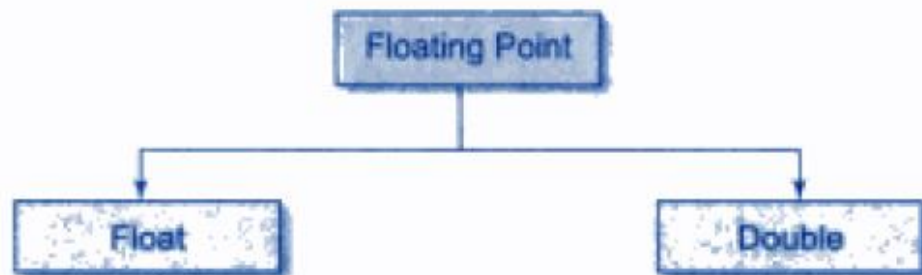
For example, the ***int*** value 1996 is actually stored as the bit pattern 0000000000000000000011111001100 as the binary equivalent of 1996 is 11111001100. Similarly, we must use a ***byte*** type variable for storing a number like 20 instead of an ***int*** type. It is because that smaller data types require less time for manipulation. We can specify a long integer by putting an ‘L’ or ‘l’ after the number. ‘L’ is preferred, as it cannot be confused with the digit ‘1’.



**Fig.2.2. Integer Data Types**

- **Floating Point type**

Floating point type can hold numbers containing factorial parts such as 2.5, 5.75, -2.358. i.e., a series of digits with a decimal point is of type floating point. There are two kinds of floating point storage in Java. They are: *float* (Single-precision floating point) and *double* (Doubleprecision floating point). In general, floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, we must append ‘f’ or ‘F’ to the numbers. For example, 5.23F, 2.25f



**Fig. 2.3 Floating point data types**

- **Character type**

Java provides a character data type to store character constants in memory. It is denoted by the keyword *char*. The size of char type is 2 bytes.

- **Boolean type**

Boolean type can take only two values: *true* or *false*. It is used when we want to test a particular condition during the execution of the program. It is denoted by the keyword *boolean* and it uses 1 byte of storage.

The memory size and range of all eight primitive data types are given in the following table 2.1:

Type	Size	Minimum Value	Maximum Value
byte	1 byte	-128	127
short	2 bytes	-32,768	32,767
int	4 bytes	-2,147,483, 648	2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	4 bytes	3.4e-038	3.4e+038
double	8 bytes	1.7e-308	1.7e+308
char	2 bytes	a single Unicode character	
boolean	1 byte	a boolean value (true or false)	

**Table 2.1 Size and Range of primitive type**

In addition to eight primitive types, there are also three kinds of *non-primitive* types in JAVA. They are also termed as *reference* or *derived* types. The non-primitive types are: *arrays*, *classes* and *interfaces*. The value of a non-primitive type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable. These are discussed later as and when they are encountered.

## 2.3 Java Tokens

The smallest individual units in a program are known as *tokens*. A Java program is basically a collection of classes. There are five types of tokens in Java language. They are: *Keywords*, *Identifiers*, *Literals*, *Operators* and *Separators*.

### 2.3.1 Keywords

Keywords are some reserved words which have some definite meaning. Java language has reserved 60 words as keywords. They cannot be used as variable name and they are written in lower-case letter. Since Java is case-sensitive, one can use these words as identifiers by

changing one or more letters to upper-case. But generally, it should be avoided. Java does not use many keywords of C/C++ language but it has some new keywords which are not present in C/C++. A list of Java keywords are given in the following table:

<b>abstract</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>byvalue</b>
<b>case</b>	<b>cast</b>	<b>catch</b>	<b>char</b>	<b>class</b>
<b>const</b>	<b>continue</b>	<b>default</b>	<b>do</b>	<b>double</b>
<b>else</b>	<b>extends</b>	<b>false</b>	<b>final</b>	<b>finally</b>
<b>float</b>	<b>for</b>	<b>future</b>	<b>generic</b>	<b>goto</b>
<b>if</b>	<b>implements</b>	<b>import</b>	<b>inner</b>	<b>instanceof</b>
<b>int</b>	<b>interface</b>	<b>long</b>	<b>native</b>	<b>new</b>
<b>null</b>	<b>operator</b>	<b>outer</b>	<b>package</b>	<b>private</b>
<b>protected</b>	<b>public</b>	<b>rest</b>	<b>return</b>	<b>short</b>
<b>static</b>	<b>super</b>	<b>switch</b>	<b>synchronized</b>	<b>this</b>
<b>threadsasafe</b>	<b>throw</b>	<b>throws</b>	<b>transient</b>	<b>true</b>
<b>try</b>	<b>var</b>	<b>void</b>	<b>volatile</b>	<b>while</b>

**Table 2.2. Java Keywords**

### 2.3.2 Identifiers

Java Identifiers are used for naming classes, methods, variables, objects, labels in a program. These are actually tokens designed by programmers. There are a few rules for naming the identifiers. These are: □ Identifier name may consist of alphabets, digits, dolar (\$) character, underscore (\_).

- Identifier name must not begin with a digit
- Upper case and lowercase letters are distinct.
- Blank space is not allowed in a identifier name.
- They can be of any length.

While writing Java programs, the following naming conventions should be followed by programmers:

- All local and private variables use only lower-case letters. Underscore is combined if required. For example, total\_marks, average
- When more than one word are used in a name, the second and subsequent words are marked with a leading upper-case letter. For example, dateOfBirth, totalMarks, studentName.

- Names of all public methods and interface variables start with a leading lower-case letter. For example, total, average.
- All classes and interfaces start with a leading upper-case letter. For example, HelloJava, Employee, ComplexNumber
- Variables that represent constant values use all upper-case letters and underscore between word if required. For example, PI, RATE, MAX\_VALUE.

### 2.3.3 Literals

Literals in Java are a sequence of characters such as digits, letters and other characters that represent constant values to be stored in a variable. A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	X	This is a test
-----	------	---	----------------

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

#### 2.3.3.1 Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f*) are substituted for 10 through 15.

### 2.3.3.2 Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E-05, and 2e+100.

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending a *D* or *d*. Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the smaller **float** type requires only 32 bits.

### 2.3.3.4 Boolean Literals

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as **boolean** or used in expressions with Boolean operators.

### 2.3.3.5 Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\ ' for the single-quote character itself and '\n' for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. For



hexadecimal, you enter a backslash-u ( \u), then exactly four hexadecimal digits. For example, '\u0061' is the ISO-Latin-1 'a' because the top byte is zero. '\u432 ' is a Japanese Katakana character.

### 2.3.4 Operators

An operator is a symbol that takes one or more arguments and operates on them to produce a result. Operators are special symbols that are commonly used in expressions. An operator performs a function on one, two, or three operands. An operator that requires one operand is called a unary operator. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a binary operator. For example, = is a binary operator that assigns the value from its righthand operand to its left-hand operand. And finally, a ternary operator is one that requires three operands. The Java programming language has one ternary operator (?:). Many Java operators are similar to those in other programming languages. Java supports most C++ operators. In addition, it supports a few that are unique to it. Operators in Java include arithmetic, assignment, increment and decrement, Relational and logical operations. We will read all these operators in detail later.

### 2.3.5 Separators

Separators are symbols used to indicate where groups of code are arranged and divided. Java separators are as follows:

{ } Braces

( ) Parentheses

[ ] Brackets

; Semicolon

, Comma

. Period

---

## 2.4 Operators

---

An operator is a symbol that takes one or more arguments and operates on them to produce a result. Operators are special symbols that are commonly used in expressions. An operator

performs a function on one, two, or three operands. An operator that requires one operand is called a unary operator. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a binary operator. For example, = is a binary operator that assigns the value from its righthand operand to its left-hand operand. And finally, a ternary operator is one that requires three operands. The Java programming language has one ternary operator (? :). Many Java operators are similar to those in other programming languages. Java supports most C++ operators. In addition, it supports a few that are unique to it. Operators in Java include arithmetic, assignment, increment and decrement, Relational and logical operations.

### 2.4.1 Arithmetic Operators

Java has five operators for basic arithmetic (Table 2.3)

Operator	Meaning	Example
+	Addition	2 + 3
-	Subtraction	5 - 2
*	Multiplication	2 * 3
/	Division	14 / 4
%	Modulus	14 % 4

**Table 2.3 Arithmetic Operators**

In the table, each operator takes two operands, one on either side of the operator. Integer division results in an integer. Because integers don not have decimal fractions, any remainder is ignored. The expression 14 / 4, for example, results in 3. The remainder 2 is ignored in this case. Modulus (%) gives the remainder once the operands have been evenly divided. For example, 14 % 4 results in 2 because 4 goes into 14 three times, with 2 left over. A sample program for arithmetic operation is given below:

*//Program 1: OperatorDemo.java*

```
public class OperatorDemo
{
public static void main (String [ ] args){
int a = 5, b =3;
```

```

double x = 25.50, y = 5.25;
System.out.println("Variable values are :\n");
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" x = " + x);
System.out.println(" y = " + y);
//Addition
System.out.println("\nAddition...");
System.out.println(" a + b = " + (a + b));
System.out.println(" x + y = " + (x + y));
//Subtraction
System.out.println("\nSubtraction...");
System.out.println(" a - b = " + (a - b));
System.out.println(" x - y = " + (x - y));
//Multiplication
System.out.println("\nMultiplication...");
System.out.println(" a * b = " + (a * b));
System.out.println(" x * y = " + (x * y));
//Division operation
System.out.println("\nDivision...");
System.out.println(" a / b = " + (a / b));
System.out.println(" x / y = " + (x / y));
//Modulus operation
System.out.println("\nModulus...");
System.out.println(" a % b = " + (a % b));
System.out.println(" x % y = " + (x % y));
}
}

```

The output of the above program will be:

```

C:\ Command Prompt
Variable values are :

a = 5
b = 3
x = 25.5
y = 5.25

Addition...
a + b = 8
x + y = 30.75

Subtraction...
a - b = 2
x - y = 20.25

Multiplication...
a * b = 15
x * y = 133.875

Division...
a / b = 1
x / y = 4.857142857142857

Modulus...
a % b = 2
x % y = 4.5

```

## 2.4.2 Relational Operators

Java has several expressions for testing equality and magnitude. All of these expressions return a boolean value (that is, true or false). Table 3.2 shows the relational operators:

Operator	Meaning	Example
==	Equal	x == 5
!=	Not equal	x != 10
<	Less than	x < 7
>	Greater than	x > 4
<=	Less than or equal to	y <= 8
>=	Greater than or equal to	z >= 15

Table 2.4 Relational Operators

### Example

```
public class Test {
```

```

public static void main(String args[]) {
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
}
}

```

### Output

```

a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false

```

### 2.4.3 Logical Operators

Expressions that result in boolean values (for example, the Relational operators) can be combined by using logical operators that represent the logical combinations **AND**, **OR**, **XOR**, and logical **NOT**. For **AND** operation, the && symbol is used. The expression will be true only if both operands tests are also true; if either expression is false, the entire expression is false. For **OR** expressions, the || symbol is used. OR expressions result in true if either or both of the operands is also true; if both operands are false, the expression is false. In addition, there is the **XOR** operator ^, which returns true only if its operands are different (one true and one false, or vice versa) and false otherwise (even if both are true). For **NOT**, the ! symbol with a single expression argument is used. The value of the NOT expression is the negation of the expression; if x is true, x is false.

#### Example

```

public class Test
{
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        System.out.println("a && b = " + (a&&b));
        System.out.println("a || b = " + (a||b) );
        System.out.println("!(a && b) = " + !(a && b));
    }
}

```

### Output

```

a && b = false
a || b = true
!(a && b) = true

```

## 2.4.4 Assignment Operators

Assignment operators are used to assign the value of an expression to a variable. The usual assignment operator is '='. The general syntax is:

**variableName = value;**

For example, `sum = 0;` // 0 is assigned to the variable sum `x = x + 1;`

Like C/C++, Java also supports the shorthand form of assignments.

For example, the statement `x = x + 1;` can be written as `x += 1;` in shorthand form.

### Example

```

public class Test
{
    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        int c = 0;
        c = a + b;
        System.out.println("c = a + b = " + c );
    }
}

```

```

c += a ;
System.out.println("c += a = " + c );
c -= a ;
System.out.println("c -= a = " + c );
c *= a ;
System.out.println("c *= a = " + c );
a = 10;
c = 15;
c /= a ;
System.out.println("c /= a = " + c );
a = 10;
c = 15;
c %= a ;
System.out.println("c %= a = " + c );
c <<= 2 ;
System.out.println("c <<= 2 = " + c );
c >>= 2 ;
System.out.println("c >>= 2 = " + c );
c >>= 2 ;
System.out.println("c >>= 2 = " + c );
c &= a ;
System.out.println("c &= a = " + c );
c ^= a ;
System.out.println("c ^= a = " + c );
c |= a ;
System.out.println("c |= a = " + c );
}
}

```

### Output

```

c = a + b = 30
c += a = 40
c -= a = 30
c *= a = 300

```

```

c /= a = 1
c %= a = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a = 0
c ^= a = 10
c |= a = 10

```

## 2.4.5 Increment and Decrement Operators

The unary **increment** and **decrement** operators ++ and -- comes in two forms, *prefix* and *postfix*. They perform two *operations*. They *increment* (or *decrement*) their operand, and *return* a value for use in some larger expression. In **prefix** form, they modify their operand and then produce the new value. In **postfix** form, they produce their operand's original value, but modify the operand in the background. For example, let us take the following two expressions:

```

y = x++;
y = ++x;

```

These two expressions give very different results because of the difference between prefix and postfix. When we use postfix operators (x++ or x--), y gets the value of x before x is incremented; using prefix, the value of x is assigned to y after the increment has occurred.

### Example

/ Increment and Decrement Operators in Java Example

```

package JavaOperators;
import java.util.Scanner;
public class IncrementandDecrement {
    private static Scanner sc;
    public static void main(String[] args) {
        int i, j;
        sc = new Scanner(System.in);
        System.out.println(" Please Enter two integer Value: ");

```



```

        i = sc.nextInt();
        j = sc.nextInt();
    System.out.println("----JAVA INCREMENT OPERATOR EXAMPLE---- \n");
        System.out.format(" Value of i : %d \n", i); //Original Value
        System.out.format(" Value of i : %d \n", i++); // Using increment Operator
        System.out.format(" Value of i : %d \n", i); //Incremented value

    System.out.println("----JAVA DECREMENT OPERATOR EXAMPLE---- \n");
        System.out.format(" Value of j : %d \n", j); //Original Value
        System.out.format(" Value of j : %d \n", j--); // Using Decrement Operator
        System.out.format(" Value of j : %d \n", j); //Decrement value
    }
}

```

## OUTPUT

```

Please Enter two integer Value:
15
40
----JAVA INCREMENT OPERATOR EXAMPLE----
Value of i : 15
Value of i : 15
Value of i : 16

----JAVA DECREMENT OPERATOR EXAMPLE----
Value of j : 40
Value of j : 40
Value of j : 39

```

### 2.4.6. Bitwise Operators

Bitwise operators are used to perform operations on individual bits in integers. Table 2.6 summarizes the bitwise operators available in the JAVA programming language. When both operands are boolean, the bitwise AND operator (&) performs the same operation as logical AND (&&). However, & always evaluates both of its operands and returns true if both are true. Likewise, when the operands are boolean, the bitwise OR (|) performs the same operation as is similar to logical OR (||). The | operator always evaluates both of its operands and returns true if at least one of its operands is true. When their operands are numbers, & and | perform bitwise manipulations.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment (x = x << y)
>>=	Right shift assignment (x = x >> y)
>>>=	Zero fill right shift assignment (x = x >>> y)
x&=y	AND assignment (x = x & y)
x =y	OR assignment (x = x   y)
x^=y	NOT assignment (x = x ^ y)

**Table 2.6 Bitwise operators in java**

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. **For example**, if op1 and op2 are two operands, then the statement

op1 << op2;

shift bits of op1 left by distance op2; fills with zero bits on the righthand side and op1 >> op2;

op1 >>> op2;

shift bits of op1 right by distance op2; fills with zero bits on the lefthand side. Each operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself.

**For example**, the statement 25 >> 1; shifts the bits of the integer 25 to the right by one position. The binary representation of the number 25 is 11001. The result of the shift operation of 11001 shifted to the right by one position is 1100, or 12 in decimal.

### Example

```
public class Test
{
    public static void main(String args[]) {
        int a = 60; /* 60 = 0011 1100 */
        int b = 13; /* 13 = 0000 1101 */
```

```

int c = 0;
c = a & b;    /* 12 = 0000 1100 */
System.out.println("a & b = " + c );
c = a | b;    /* 61 = 0011 1101 */
System.out.println("a | b = " + c );
c = a ^ b;    /* 49 = 0011 0001 */
System.out.println("a ^ b = " + c );
c = ~a;       /* -61 = 1100 0011 */
System.out.println("~a = " + c );
c = a << 2;    /* 240 = 1111 0000 */
System.out.println("a << 2 = " + c );
c = a >> 2;    /* 15 = 1111 */
System.out.println("a >> 2 = " + c );
c = a >>> 2;   /* 15 = 0000 1111 */
System.out.println("a >>> 2 = " + c );
}
}

```

### Output

```

a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2 = 15
a >>> 2 = 15

```

## 2.4.7 Special Ternary Operators

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-thenelse statements. This operator is the `?`. It can seem somewhat confusing at first, but the `?` can be used very effectively once mastered. The `?` has this general form:

*expression1 ? expression2 : expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same (or compatible) type, which can't be **void**.

Here is an example of the way that the **?** is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark. If **denom** equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire **?** expression. If **denom** does not equal zero, then the expression *after* the colon is evaluated and used for the value of the entire **?** expression. The result produced by the **?** operator is then assigned to **ratio**.

Here is a program that demonstrates the **?** operator. It uses it to obtain the absolute value of a variable.

**// Demonstrate ?.**

```
class Ternary {  
    public static void main(String args[]) {  
        int i, k;  
        i = 10;  
        k = i < 0 ? -i : i; // get absolute value of i  
        System.out.print("Absolute value of ");  
        System.out.println(i + " is " + k);  
        i = -10;  
        k = i < 0 ? -i : i; // get absolute value of i  
        System.out.print("Absolute value of ");  
        System.out.println(i + " is " + k);  
    }  
}
```

**The output** generated by the program is shown here:

Absolute value of 10 is 10

Absolute value of -10 is 10

## 2.6 Precedence in Arithmetic Operators

Table 2.7 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the `[ ]`, `( )`, and `.` can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator `->`. It was added by JDK 8 and is used in lambda expressions.

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

**Table 2.7 The Precedence of the Java Operators**

## 2.6 Type Casting

If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

<i>From</i>	<i>To</i>
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

**Table 2.8 Casts that result in No Loss of Information**

### 2.6.1 Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

### 2.6.2 Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For

example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

***(target-type) value***

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range. General Syntax for Type Casting is:

$Type\ variable1 = (type)\ variable2$
---------------------------------------

The process of converting one data type to another is called casting. Casting is often necessary when a method returns a type different than the one, we require.

---

## 2.7 Check Your Progress

---

1. Java has ..... operators for basic arithmetic.
2. In relational operators, all of expressions return a .....value.
3. IN Logical operators, for AND operation, the ..... symbol is used.

4. IN Logical operators, for ..... operation, the || symbol is used.
5. The unary increment and decrement operators ++ and -- comes in two forms,.....
6. ....operators are used to perform operations on individual bits in integers.
7. Java includes a special ternary (three-way) operator that can replace certain types of .....statements
8. The smallest individual units in a program are known as.....
9. ....in Java are a sequence of characters such as digits, letters and other characters that represent constant values to be stored in a variable.

---

## 2.8 Summary

---

An operator is a symbol that takes one or more arguments and operates on them to produce a result. Operators are special symbols that are commonly used in expressions. An operator performs a function on one, two, or three operands. An operator that requires one operand is called a unary operator. Operators in Java include arithmetic, assignment, increment and decrement, Relational and logical operations.

Java has five operators for basic arithmetic. In Relational Operators, Java has several expressions for testing equality and magnitude. All of these expressions return a boolean value (that is, true or false). In Logical operators, Expressions that result in boolean values (for example, the Relational operators) can be combined by using logical operators that represent the logical combinations AND, OR, XOR, and logical NOT. Assignment operators are used to assign the value of an expression to a variable. The usual assignment operator is '='. The unary increment and decrement operators ++ and -- comes in two forms, prefix and postfix. They perform two operations. They increment (or decrement) their operand, and return a value for use in some larger expression. In prefix form, they modify their operand and then produce the new value. In postfix form, they produce their operand's original value, but modify the operand in the background. Bitwise operators are used to perform operations on individual bits in integers.

The smallest individual units in a program are known as tokens. A Java program is basically a collection of classes. There are five types of tokens in Java language. They are: Keywords, Identifiers, Literals, Operators and Separators.



If the two types are compatible, then Java will perform the conversion automatically, it is called Type Casting. When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

## 2.9 Keywords

**AND-** The expression will be true only if both operands tests are also true; if either expression is false, the entire expression is false.

**OR-** OR expressions result in true if either or both of the operands is also true

**XOR-** which returns true only if its operands are different (one true and one false, or vice versa) and false otherwise (even if both are true)

**Special Ternary (?)**- It can replace certain types of if- thenelse statements.

**Dynamic Initialization-** Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

**Primitive Type-** A variable of primitive type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. Primitive types are also termed as intrinsic or built-in types.

**Tokens-** The smallest individual units in a program are known as tokens. A Java program is basically a collection of classes. There are five types of tokens in Java language. They are: Keywords, Identifiers, Literals, Operators and Separators.

---

## 2.10 Self-Assessment Test

---

- Q.1 Explain any four types of operators.
- Q.2 Briefly explain the concept of increment and decrement operator.
- Q.3 What do you understand by Dynamic Initialization.
- Q.4 Differentiate between Print() and Println() methods.
- Q.5 Explain various tokens in Java.
- Q.6 Explain concept of Type Casting in Java with example.

---

## 2.11 Answers to check your progress

---

- 1. five
- 2. boolean
- 3. &&
- 4. OR
- 5. prefix and postfix
- 6. Bitwise
- 7. if- thenelse
- 8. tokens
- 9. Literals

---

## 2.12 References / Suggested Readings

---

- 1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
- 2. Java- The Complete Reference by Herbert Schildt, ORACLE.
- 3. Java For Beginners by Scott Sanderson
- 4. Head First Java by Kathy Sierra & Bert Bates
- 5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
- 6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.

<b>SUBJECT: JAVA PROGRAMMING</b>	
<b>COURSE CODE: MCA-13</b>	<b>AUTHOR: AYUSH SHARMA</b>
<b>LESSON NO. 3</b>	
<b>Control Structures and Looping</b>	

## STRUCTURE

- 3.0 Learning Objective
- 3.1 Introduction
- 3.2 Control Structure with If Statement
  - 3.2.1 Simple If Statement
  - 3.2.2 If.....Else Statement
  - 3.2.3 Nesting of If.....Else Statement
  - 3.2.4 The Else If Ladder
- 3.3 The while Statement
  - 3.3.1 The Do Statement
- 3.4 The for Statement
  - 3.4.1 Additional Features of for Loop
  - 3.4.2 Nesting of for Loops
  - 3.4.3 The Enhanced for Loop
- 3.5 The switch Statement
- 3.6 Break & Continue Statement
  - 3.6.1 Break Statement
  - 3.6.2 Continue Statement
- 3.7 Return Statement
- 3.8 Check Your Progress
- 3.9 Summary
- 3.10 Keywords
- 3.11 Self-Assessment Test
- 3.12 Answers to check your progress
- 3.13 References / Suggested Readings

---

## **3.0 LEARNING OBJECTIVE**

---

After going through this unit, you will be able to:

- Learn about control and flow structure.
- Learn about If. Else statement, While statement, Switch statement and for loop.
- Learn about Break, Continue and return statement.

---

## **3.1 INTRODUCTION**

---

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This chapter describes all of Java's operators. This chapter will describe the available operators that you can add to your lines of code as you program more complex scenarios. Operators are mainly used to control, modify and compare data in a Java language environment. This chapter will emphasize on a non-sequential method of Java programming. You will get acquainted with the if-then-else and different loop statements. With our previous simple program examples, we were oriented that a Java class is executed in one direction – from the topmost line of code up to the bottom, or what we call sequential programming. However, there will be cases that you will be required to write codes in a non-sequential fashion, especially for those more complicated scenarios. This is accomplished using logical and looping statements, so you can control the flow of your program to perform more complex functions.

---

## **3.2 Control Structure with If Statement**

---

While programing, we have a number of situations where we may have to change the order of execution of statements based on certain conditions/decisions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly. The statements used to handle those situations are called decision making statement.

### 3.2.1 Simple if Statement

The *if* statement enables our program to selectively execute other statements, based on some criteria. The syntax of if statement is:

```
if (boolean_expression)
{
    statement-block ;
}
statement;
```

The *statement-block* may be a single statement or a group of statements. If the *boolean-expression* evaluates to true, then the block of code inside the *if* statement will be executed. If not the first set of code after the end of the *if* statement (after the closing curly brace) will be executed.

For example,

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

In this case, if **percentage** contains a value that is greater than or equal to 40, the expression is true, and **println( )** will execute. If **percentage** contains a value less than 40, then the **println( )** method is bypassed. What if we want to perform a different set of statements if the expression is false? We use the *else* statement for that.

**Example :**

```
public class Main {
    public static void main(String[] args) {
        if (20 > 18) {
            System.out.println("20 is greater than 18"); // obviously
        }
    }
}
```

**Output :** 20 is greater than 18

### 3.2.2 If.....Else Statement

The general syntax of *if-else* statement is:

```
if ( Boolean_expression )
    statement; //executes when the expression is true
else
    statement; //executes when the expression is false
```

Let us consider the same example but at this time the output should be *Pass* or *Fail* depending on percentage of marks. i.e., if percentage is equal to or more than 40 then the output should be *Pass*; otherwise *Fail*. This can be done by using an *if* statement along with an *else* statement.

Here is the segment of code:

```
if (percentage>=40)
System.out.println("Pass");
else
System.out.println("Fail");
```

When a series of decisions are involved, we may have to use more than one *if-else* statements in nested form.

#### Example :

```
public class Main {
    public static void main(String[] args) {
        int time = 20;
        if (time < 18) {
            System.out.println("Good day.");
        } else {
            System.out.println("Good evening.");
        }
    }
}
```

```
}  
}
```

**Output :** Good evening.

### 3.2.3. Nesting of If.....Else Statement

A *nested if* is an **if** statement that is the target of another **if** or **else**. Nested **ifs** are very common in programming. When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Here is an example:

```
if(i == 10)  
{  
    if(j < 20) a = b;  
    if(k > 100) c = d; // this if is  
    else a = c; // associated with this else  
}  
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

### 3.2.4 The Else If Ladder

A common programming construct that is based upon a sequence of nested **ifs** is the *if-elseif* ladder. It looks like this:

```
if(condition)  
    statement;  
else if(condition)  
    statement;  
else if(condition)
```

*statement;*

.  
. .  
.

else

*statement;*

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

**// Demonstrate if-else-if statements.**

```
class IfElse
{
public static void main(String args[])
{
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
```



```
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

**Here is the output produced by the program:**

April is in the Spring. You might want to experiment with this program before moving on. As you will find, no matter what value you give **month**, one and only one assignment statement within the ladder will be executed.

### 3.3 The while Statement

In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied. The process of repeatedly executing a block of statements is known as *looping*. At this point, we should remember that Java does not support *goto* statement. Like C/C++, Java also provides the three different statements for looping. These are:

- *while*
- *do-while*

**The *while* and *do-while* statements**

We use a *while* statement to continually execute a block while a condition remains true. The general syntax of the *while* statement is:

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

First, the *while* statement evaluates *expression* , which must return a boolean value. If the expression returns *true*, the *while* statement executes the statement(s) in the while block.

The *while* statement continues testing the expression and executing its block until the expression returns *false*.

The Java programming language provides another statement that is similar to the *while* statement: the ***do-while*** statement.

The general syntax of *do-while* is:

```
do
{
    statement(s);
}while (expression);
```

Statements within the block associated with a *do-while* are executed at least once. Instead of evaluating the expression at the top of the loop, *do-while* evaluates the expression at the bottom. Here is the previous program rewritten to use *do-while* loop.

**A program of *while* loop is shown below:**

```
class Fibo
{
public static void main(String args[])
{
System.out.println("0\n1");
int n0=0,n1=1,n2=1;
while(n2<50)
{
System.out.println(n2);
n0=n1;
n1=n2;
n2=n1+n0;
}
System.out.println(n2);
}
}
```

The output of the above program will be the Fibonacci series between 0 to 50 (i.e., 0 1 1 2 3 5 8 13 21 34).

**A program of *do-while* loop is shown below:**

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int i = 0;  
  
        do {  
  
            System.out.println(i);  
  
            i++;  
  
        }  
  
        while (i < 5);  
  
    }  
}
```

**Output :**

```
0  
1  
2  
3  
4
```

---

## 3.4 The for Statement

---

The *for* statement provides a compact way to iterate over a range of values. The general form of the *for* statement can be expressed like this:

```

for (initialization; termination_condition; increment)
{
    statement(s);
}

```

The *initialization* is an expression that initializes the loop. It is executed once at the beginning of the loop. The *termination\_condition* determines when to terminate the loop.

*This condition is evaluated at the top of each iteration of the loop. When the condition evaluates to false, the loop terminates. Finally, increment is an expression that gets invoked after each iteration through the loop. All these components are optional. In*

fact, to write an infinite loop, we can omit all three expressions:

```

for (; ; )
{
    // infinite loop
}

```

Often, *for* loops are used to iterate over the elements in an array or the characters in a string.

The following program segment uses a *for* loop to calculate the summation of 1 to 50:

```

for (i = 1; i <= 50; i++)
{
    sum = sum + i;
}

```

**A program of For loop is shown below:**

```

public class Main {

    public static void main(String[] args) {

        for (int i = 0; i < 5; i++) {

            System.out.println(i);

        }

    }

}

```

**Output :**

0  
1  
2  
3  
4

### 3.5 The switch Statement

We have seen that when one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, the program becomes difficult to read and follow when the number of alternatives increases. Like C/C++, JAVA has a built-in multiway decision statement known as a ***switch***. The *switch* statement provides variable entry points to a block. It tests the value of a given *variable* or *expression* against a list of *case* values and when a match is found, a block of statements associated with that *case* is executed.

The general form of *switch* statement is as follows :

```
switch( expression )
{
    case value : statements;
                                break;
    case value : statements
                                break;
    ...
    default : statements // optional default section
                break;
}
```

The *expression* is evaluated and compared in turn with each *value* prefaced by the *case* keyword. The values must be *constants* (i.e., determinable at compile-time) and may be of type byte, char, short, int, or long.

**Example :**

```
int day = 4;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
}
```

**Output:**

"Thursday" (day 4)

## 3.6 Break & Continue Statement

### 3.6.1 Break Statement

In JAVA, the *break* statements has two forms: *unlabelled* and *labelled*. We have seen the unlabelled form of the *break* statement used with *switch* earlier. As noted there, an unlabelled *break* terminates the enclosing *switch* statement, and the flow of control transfers to the statement immediately following the *switch*. It can be used to terminate a *for*, *while*, or *do-while* loop. A ***break*** (unlabelled form) statement, causes an immediate jump out of a loop to the first statement after its end. When the *break* statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loop is nested, the *break* would only exit from the loop containing it. This means, the *break* will exit only a single loop.

#### Example :

```
public class BreakExample1 {
    public static void main(String args[]){
        int num =0;
        while(num<=100)
        {
            System.out.println("Value of variable is: "+num);
            if (num==2)
            {
                break;
            }
            num++;
        }
        System.out.println("Out of while-loop");
    }
}
```

#### Output:

```
Value of variable is: 0
Value of variable is: 1
Value of variable is: 2
Out of while-loop
```

### 3.6.2 Continue Statement

The ***continue*** statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements. It is written as: **continue ;**

A continue does not cause an exit from the loop. Instead, it immediately initiates the next iteration. We can use the continue statement to skip the current iteration of a *for*, *while*, or *do-while* loop. In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, we have to place the label name before the loop with a colon at the end.

For example,

**loop1:** for (.....)

```
{
.....
.....
}
.....
```

We have seen that a simple break statement causes the control to jump outside the nearest loop and a simple continue statement returns the current loop. If we want to jump outside a nested loop or to continue a loop that is outside the current one, then we may have to use the *labelled break* and *labelled continue* statement.

#### Example

```
public class ContinueExample
{
    public static void main(String args[]){
        for (int j=0; j<=6; j++)
        {
            if (j==4)
            {
                continue;
            }
            System.out.print(j+" ");
        }
    }
}
```



## Output:

0 1 2 3 5 6

### 3.7 Return Statement

The last of the branching statements is the *return* statement. We can use *return* to exit from the current method. The flow of control returns to the statement that follows the original method call. The *return* statement has two forms: one that returns a value and one that does not. To return a value, simply put the value (or an expression that calculates the value) after the return keyword: **return sum;** The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of *return* that does not return a value: **return;**

#### Example :

```
class ReturnStatement
{
    public static void main(String arg[])
    {
        boolean t = true;

        System.out.println("Before the return"); // LINE A

        if(t) return; // return to caller

        System.out.println("This won't execute"); // LINE B
    }
}
```

#### Output :

Before the return

### 3.8 Check Your Progress

#### Write true or false (Q.1 to Q.4)

1. A program stops its execution when a break statement is encountered.
2. One if can have more than one else if clause.

3. A continue cause an exit from the loop.
4. A break statement causes an immediate jump out of a loop to the first statement after its end.
5. In JAVA, the break statements has two forms: .....and .....
6. The basic syntax for continue statement is: .....
7. We can use .....to exit from the current method.
8. Basic syntax for Break statement is .....
9. Like C/C++, JAVA has a built-in multiway decision statement known as.....

---

### 3.9 Summary

---

While programing, we have a number of situations where we may have to change the order of execution of statements based on certain conditions/decisions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly. The statements used to handle those situations are called decision making statement. In while loop, , a sequence of statements are executed until some conditions for the termination of the loop are satisfied. The process of repeatedly executing a block of statements is known as looping. Java also provides the three different statements for looping. These are: while, do-while.

In JAVA, the break statements have two forms: unlabelled and labelled. The continue statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements. It is written as: continue;

The last of the branching statements is the return statement. We can use return to exit from the current method. The flow of control returns to the statement that follows the original method call. The return statement has two forms: one that returns a value and one that does not.

---

### 3.10 Keywords

---

**print()**- To print a statement within same line.

**println()**- To print a statement in the next line.

**goto-** Java doesn't support goto keyword. Instead, it uses While loop.

**Break-** A break statement, causes an immediate jump out of a loop to the first statement after its end.

**Continue-** The continue statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements.

**Return-** We can use return to exit from the current method.

---

### 3.11 Self-Assessment Test

---

- Q.1 What do you mean by looping? What are the different types of loop in the Java programming language? Briefly explain with their syntax.
- Q.2. Write a Java program to display the multiplication table of a particular number using for loop.
- Q.3. What do you understand with control structure? Explain If-else with proper example.
- Q.4. What is the syntax for Do-while loop structure? Give an example.
- Q.5. What are the necessary conditions for a FOR Loop? Explain with example.
- Q.6. Differentiate between Break and Continue Statement with example.
- Q.7. Write down a program to illustrate the concept of Return Statement.

---

### 3.12 Answers to check your progress

---

- 1. False
- 2. True
- 3. False
- 4. True
- 5. unlabelled and labelled.
- 6. continue;
- 7. return
- 8. break;
- 9. Switch

---

### **3.13 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.

<b>SUBJECT: JAVA PROGRAMMING</b>	
<b>COURSE CODE: MCA-13</b>	<b>AUTHOR: AYUSH SHARMA</b>
<b>LESSON NO. 4</b>	
<b>Inheritance and Polymorphism</b>	

## STRUCTURE

- 4.0 Learning Objective
- 4.1 Introduction
- 4.2 Inheritance: Extending a Class
- 4.3 Single Inheritance
- 4.4 Multilevel Inheritance
- 4.5 Hierarchical Inheritance
- 4.6 Interfaces: Multiple Inheritance
  - 4.6.1 Defining Interfaces
  - 4.6.2 Extending Interfaces
  - 4.6.3 Implementing Interfaces
  - 4.6.4 Default Interface Methods
  - 4.6.5 Default Method Fundamentals
  - 4.6.6 Multiple Inheritance Issues
  - 4.6.7 Use static Methods in an Interface
- 4.7 Abstraction through Abstract Classes
  - 4.7.1 Using Final with Inheritance
- 4.8 Polymorphism
  - 4.8.1 Virtual Methods
- 4.9 Check Your Progress
- 4.10 Summary
- 4.11 Keywords
- 4.12 Self-Assessment Test
- 4.13 Answers to check your progress
- 4.14 References / Suggested Readings

---

## 4.0 LEARNING OBJECTIVE

---

After going through this chapter, you will be able to:

- Gain the concept of Inheritance.
- Differentiate between different form of Inheritance.
- Illustrate the basic concept of Interface
- Know how to use Interfaces.
- Understand the concept of Polymorphism.

---

## 4.1 INTRODUCTION

---

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

An **interface** is a reference type in **Java**. It is similar to class. It is a collection of abstract methods. A class implements an **interface**, thereby inheriting the abstract methods of the **interface**. Along with abstract methods, an **interface** may also contain constants, default methods, static methods, and nested types.

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

---

## 4.2 Inheritance: Extending a Class

---

Reusability is one of the important feature of object-oriented programming and it can be achieved through **inheritance**. Java supports the concepts of inheritance. With the use of inheritance, the information is made manageable in a hierarchical order. Inheritance can be defined as the process where one object acquires the properties of another. When we want to create a new class and there is already a class that includes some of the code that we want, we can derive the new class from the existing class. In doing this, we can reuse the fields and methods of the existing class without rewriting them again.

A class that is derived from another class is called a **subclass** (also a *derived class*, *extended class*, or *child class*). A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. The class from which the subclass is derived is called a **superclass** (also a *base class* or a *parent class*). Constructors cannot be inherited by subclasses, but the constructor of the superclass can be invoked from the subclass. In Java, inheritance is implemented by the process of extension. To define a new class as an extension of an existing class, we simply use an **extend** clause in the header of the new classes definition. The concept of inheritance is used to make the things from general to more specific.

For example, when we hear the word 'vehicle' then we get an image in our mind that it moves from one place to another and that is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four-wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the example that car is a specific word and vehicle is the general word. If we think technically about this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of its parent (in this case vehicle) class. At this point, we are going to describe the types of inheritance supported by Java.

```
// A simple example of inheritance.
```

```
// Create a superclass.
```

```
class A {  
    int i, j;  
    void showij() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

```

// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}

class SimpleInheritance {
public static void main(String args []) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}

```

The output from this program is shown here:



Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.

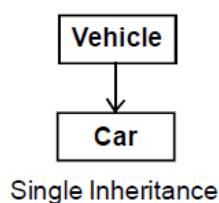
Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

---

## 4.3 Single Inheritance

---

When a subclass is derived from its parent class then this mechanism is known as single inheritance. In case of single inheritance there is only a sub class and its parent class. It is also called *one level* inheritance. The pictorial representation of single inheritance is as follows:



For example, let us consider a simple example for the demonstration of single inheritance:

//**Program 4.1:** B.java (Program showing Single Inheritance)

```
class A // super class A
{
int x;
int y;
int getValue(int p, int q) {
```

```

x = p;
y = q;
return(0);
}

void Show() {
System.out.println(x);
}
}

class B extends A // subclass B inheriting getValue A
{
public static void main(String args[ ]) {
A a = new A();
a.getValue(5,10);
a.Show();
}

void display() {
System.out.println("I am in B");
}
}

```

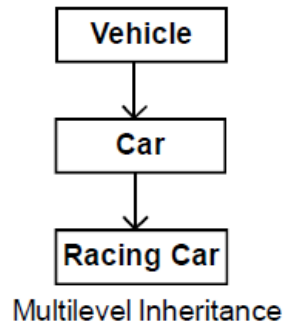
The output will display only 5. The *getValue()* and *show()* are members of superclass *A*. With the statement *a.getValue(5,10);* the *getValue()* is inherited from class *A*. As the *Show()* method of class *A* is displaying only the first parameter so it is displaying only one value in the subclass *B* although it is taking two values 5 and 10 when it invoked by the statement *a.Show()* in class subclass *B*.

---

## 4.4 Multilevel Inheritance

---

It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the *multilevel inheritance*. The derived class is called the subclass or child class for its parent class and this parent class works as the child class for its just above ( parent ) class. Multilevel inheritance can go up to any number of level. The pictorial representation of multilevel inheritance is as follows:



// **Program 4.2** : C.java (Program showing Multilevel Inheritance)

```
class A
{
    int x;
    int y;
    int get(int p, int q)
    {
        x = p;
        y = q;
        return(0);
    }
    void show()
    {
        System.out.println(x);
    }
}

class B extends A //subclass B inheriting from A
{
    void Showb()
    {
        System.out.println("I am in B ");
    }
}

class C extends B //subclass C inheriting from B
{
    void Display()
```

```

{
System.out.println("I am in C");
}
public static void main(String args[])
{
A a = new A();
a.get(5,10);
a.show();
}
}

```

The output of the above program will be 5. Here, *a* is an object of superclass *A* and it is inheriting *get( )* and *show( )* methods of *A*. The subclass *B* has one method *Showb()*. The class *C* is the subclass of *B* and it has one *Display()* method. We can also create objects of class *B* and *C* and use these two method *Showb()* and *Display()* for displaying the messages “*I am in B*” and “*I am in C*” respectively.

The mechanism of inheriting the features of more than one base class into a single class is known as *multiple inheritance*. **Java does not support multiple inheritance**. But the multiple inheritance can be achieved by using the *interface*. In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class. The concept of interface will be discussed in the next unit of this block.

## 4.5 Hierarchical Inheritance

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

### Example:

File: TestInheritance3.java

```

class Animal{
void eat(){System.out.println("eating...");}
}

```

```

class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}

```

**Output:**

meowing...

eating...

## 4.6 Interfaces: Multiple Inheritance

An interface is a collection of methods and variables like a class but it is not a class. An interface defines a set of methods but does not implement them. Writing an interface is similar to writing a class still there exists some differences. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements.

**Definition**

*An interface is a named collection of method definitions (without implementations). An interface can also declare constants.*

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Remember that - to implement an interface, a class must create the complete set of methods defined by the interface.

### 4.6.1 Defining Interfaces

The syntax for defining an interface is very similar to that for defining a class, which is shown below :

```
interface interfaceName
{
    return-type methodName1(parameter-list);
    return-type methodName2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    .....
    .....
    return-type methodNameN(parameter-list);
    type final-varnameN = value;
}
```

From the above syntax we have seen that an interface definition has two components: **the interface declaration** and the **interface body**. The interface declaration declares various attributes about the interface, such as its name and whether it extends other interfaces. The interface body contains the constant and the method declarations for that interface.

Variables declared inside of interface declarations are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a

constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Example of interface *Area* and *Shape* are shown below :

```
interface Area
{
    final static float pi =3.142F;
    float compute (float x, float y);
    void show();
}

and

interface Shape
{
    public double area();
    public double volume();
}
```

#### 4.6.2 Extending Interfaces

Like classes, interfaces can be extended i.e. an interface can be sub interfaced from other interfaces. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface. The syntax for using the extend keyword is shown below:

```
interface name2 extends name1
{
    body of name2
}
```

Here, *name2* is the child interface and *name1* is the parent interface.

In following the Sports interface is extended by the Hockey and the Football interfaces.

```
public interface Sports
{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}
```

```

public interface Football extends Sports
{
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

public interface Hockey extends Sports
{
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}

```

Here, in the Hockey interface has four methods, but it inherits two method from Sports. So, the class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

The ***public*** access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

### 4.6.3 Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract class.

To declare a class that implements an interface, you include an **implements** clause in the class declaration as shown in the following :



**class** *classname* **implements** *Interfacename*

```
{  
// body of classname  
}
```

For example, we have already define the interface Shape. Let us try to implement it using the Point class.

### **Program 4.3 : Implementation of Shape interface**

**// Point.java**

**interface** Shape

```
{  
public double area();  
public double volume();  
}
```

**public class** Point **implements** Shape

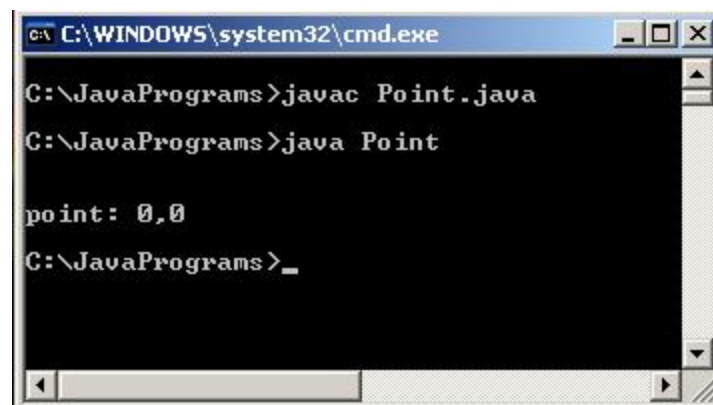
```
{  
static int x, y;  
public Point()  
{  
x = 0;  
y = 0;  
}  
public double area()  
{  
return 0;  
}  
public double volume()  
{  
return 0;  
}  
public static void print()  
{  
System.out.println("\n Point: " + x + "," + y);  
}
```

```

public static void main(String args[])
{ // object declaration
Point p = new Point();
p.print();
}
}

```

Output :



```

C:\WINDOWS\system32\cmd.exe
C:\JavaPrograms>javac Point.java
C:\JavaPrograms>java Point

point: 0,0
C:\JavaPrograms>_

```

**Fig 4.1 Output of Program 4.3**

One class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

#### **4.6.4 Default Interface Methods**

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were

added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove()**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove()** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove()**, even though it was not needed. Today, a default implementation for **remove()** can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**. Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created. One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

### 4.6.5 Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
}
```

**MyIF** declares two methods. The first, **getNumber( )**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString( )**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString( )** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString( )** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.  
class MyIFImp implements MyIF {  
    // Only getNumber() defined by MyIF needs to be implemented.  
    // getString() can be allowed to default.  
    public int getNumber() {  
        return 100;  
    }  
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {
public static void main(String args[]) {
    MyIFImp obj = new MyIFImp();
    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());
    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
}
}
```

The output is shown here:

100

Default String

As you can see, the default implementation of **getString( )** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString( )**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString( )** for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**:

```
class MyIFImp2 implements MyIF {
    // Here, implementations for both getNumber( ) and getString( ) are provided.
    public int getNumber() {
        return 100;
    }
    public String getString() {
        return "This is a different string.";
    }
}
```

Now, when **getString( )** is called, a different string is returned.

#### 4.6.6 Multiple Inheritance Issues

As explained earlier , Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behavior. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset( )** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the method? To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset( )** default method, **MyClass**' version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass**' implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset( )**, then an error will occur.

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset( )** will be used. It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**. Its general form is shown here:

*InterfaceName.super.methodName( )*

For example, if **Beta** wants to refer to **Alpha**'s default for **reset( )**, it can use this statement:

`Alpha.super.reset();`

#### 4.6.7 Use static Methods in an Interface

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

*InterfaceName.staticMethodName*

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
  
    // This is a static interface method.  
    static int getDefaultNumber() {  
        return 0;  
    }  
}
```

The **getDefaultNumber( )** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

Abstraction through Abstract Classes

## 4.7 Abstraction through Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area()** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area()** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*.

**// A Simple demonstration of abstract.**

```
abstract class A {  
    abstract void callme();  
    // concrete methods are still allowed in abstract classes  
    void callmetoo() {  
        System.out.println("This is a concrete method.");  
    }  
}
```



```

}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}

```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

**// Using abstract methods and classes.**

```

abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {

```

```

    dim1 = a;
    dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
    }
}

```

```

figref = t;

System.out.println("Area is " + figref.area());
}
}

```

As the comment inside **main()** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area()**. To prove this to yourself, try creating a subclass that does not override **area()**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

### 4.7.1 Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

#### 4.7.1.1 Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```

class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

```
}
```

Because **meth()** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

#### 4.7.1.2 Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {  
    //...  
}  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
    //...  
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

---

## 4.8 Polymorphism

---

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class

object. Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object. A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

### **Example**

```
public interface Vegetarian{ }  
public class Animal{ }  
public class Deer extends Animal implements Vegetarian{ }
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples –

A Deer IS-A Animal

A Deer IS-A Vegetarian

A Deer IS-A Deer

A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal –

### **Example**

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

### 4.8.1 Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the `super` keyword within the overriding method.

#### Example

```
/* File name : Employee.java */
public class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public void mailCheck() {
        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }
}
```

```

public String getAddress() {
    return address;
}

public void setAddress(String newAddress) {
    address = newAddress;
}

public int getNumber() {
    return number;
}
}

```

**Now suppose we extend Employee class as follows –**

```

/* File name : Salary.java */
public class Salary extends Employee {
    private double salary; // Annual salary

    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {

```

```

        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}

```

**Now, you study the following program carefully and try to determine its output –**

```

/* File name : VirtualDemo.java */
public class VirtualDemo {

    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

This will produce the following result –

### **Output**

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0



Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

Here, we instantiate two Salary objects. One using a Salary reference s, and the other using an Employee reference e.

While invoking s.mailCheck(), the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on e is quite different because e is an Employee reference. When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

## 4.9 Check your progress

1. A class that is derived from another class is called a.....
2. When a subclass is derived from its parent class then this mechanism is known as.....
3. When a subclass is derived from a derived class then this mechanism is known as the.....
4. When two or more classes inherits a single class, it is known as .....
5. An .....is a named collection of method definitions (without implementations) and it can also declare constants.
6. The..... access specifier indicates that the interface can be used by any class in any package.
7. Java does not support the .....inheritance of classes.
8. ....is the ability of an object to take on many forms.
9. Declaring a class as final implicitly declares all of its methods as.....
10. It is illegal to declare a class as both .....and final.

---

## 4.10 Summary

---

Inheritance can be defined as the process where one object acquires the properties of another. Reusability is one of the important feature of object-oriented programming and it can be achieved through inheritance. A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). A subclass inherits all the members (fields, methods, and nested classes) from its superclass. The class from which the subclass is derived is called a superclass (also a base class or a parent class).

When a subclass is derived from its parent class then this mechanism is known as single inheritance. In case of single inheritance there is only a sub class and its parent class. It is also called one level inheritance.

It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for its parent class and this parent class works as the child class for its just above (parent) class. Multilevel inheritance can go up to any number of level. When two or more classes inherits a single class, it is known as hierarchical inheritance.

An interface is a collection of methods and variables like a class but it is not a class. An interface defines a set of methods but does not implement them. Writing an interface is similar to writing a class still there exists some differences. An interface is a named collection of method definitions (without implementations). An interface can also declare constants.

An interface default method is defined similar to the way a method is defined by a class. The primary difference is that the declaration is preceded by the keyword default.

Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

There are situations in which you will want to define a superclass that declares the structure

of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

---

## 4.11 Keywords

---

**extend**- To define a new class as an extension of an existing class, we simply use an **extend**.

**one level inheritance**- In case of single inheritance there is only a sub class and its parent class. It is also called one level inheritance.

**Super**- superclass is used for parent class.

**Sub**- Subclass is used for Derived class.

**hierarchical**- When two or more classes inherits a single class, it is known as hierarchical inheritance.

**Multilevel**- When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance.

**Interface**- this keyword used for creating interfaces.

**Implement**- When a class implements an interface, we use this keyword.

**Final-** The keyword final has three uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.

---

## 4.12 Self-Assessment Test

---

- Q.1. What do you understand by Inheritance? What are different types of Inheritance in Java?
- Q.2. Explain and describe single inheritance with suitable example.
- Q.3. Differentiate between multilevel and hierarchical inheritance.
- Q.4. How is Interface differ from inheritance? Discuss.
- Q.5. Briefly discuss interface implementation.
- Q.6. Explain the use of static methods in interface.
- Q.7. Describe the concept of final keyword in inheritance.
- Q.8. What do you understand with polymorphism.? Give suitable example.

---

## 4.13 Answers to check your progress

---

1. subclass
2. single inheritance
3. multilevel inheritance
4. hierarchical inheritance.
5. interface
6. public
7. multiple
8. Polymorphism
9. final
10. abstract

---

## **4.14 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.

<b>SUBJECT: JAVA PROGRAMMING</b>	
<b>COURSE CODE: MCA-13</b>	<b>AUTHOR: AYUSH SHARMA</b>
<b>LESSON NO. 5</b>	
<b>Multithreaded Programming</b>	

## STRUCTURE

- 5.0 Learning Objective
- 5.1 Introduction
- 5.2 The Java Thread Model
  - 5.2.1 Thread Priorities
  - 5.2.2 Synchronization
  - 5.2.3 The Main Thread
- 5.3 Creating a Thread
  - 5.3.1 Implementing Runnable
  - 5.3.2 Extending Thread
- 5.4 Creating Multiple Threads
  - 5.4.1 Thread Priorities
  - 5.4.2 Synchronization
    - 5.4.2.1 Using Synchronized Methods
    - 5.4.2.2 The synchronized Statement
  - 5.4.3 Interthread Communication
  - 5.4.4 Deadlock
- 5.5 Stopping and Blocking a Thread
- 5.6 Life Cycle of a Thread
- 5.7 Check Your Progress
- 5.8 Summary
- 5.9 Keywords
- 5.10 Self-Assessment Test
- 5.11 Answers to check your progress
- 5.12 References / Suggested Readings

---

## 5.0 LEARNING OBJECTIVE

---

After going through this unit, you will be able to:

- learn about the concept of multithreaded programming.
- describe the life cycle of thread.
- learn about deadlock occurrence.
- Learn about Synchronization concept in java thread model and multiple threads case.
- Learn about Interthread communication.

---

## 5.1 INTRODUCTION

---

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details. Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks

that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

---

## 5.2 The Java Thread Model

---

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file



is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multi-core systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that Java's multithreading features work in both types of systems. In a singlecore system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Threads exist in several states. Here is a general description. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

### **5.2.1 Thread Priorities**

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead,

a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

- *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

### 5.2.2 Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

### 5.2.3 The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let’s begin by reviewing the following example:

// Controlling the main Thread.

```
class CurrentThreadDemo {
public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);
    try {
        for(int n = 5; n > 0; n--) {
            System.out.println(n);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted");
    }
}
```

```
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently.

**Here is the output generated by this program:**

```
Current thread: Thread[main,5,main]
```

```
After name change: Thread[My Thread,5,main]
```

```
5
```

```
4
```

```
3
```

```
2
```

```
1
```

Notice the output produced when **t** is used as an argument to **println( )**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**.

The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName( )**. You can obtain the name of a thread by calling **getName( )** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:

```
final void setName(String threadName)
final String getName( )
```

Here, *threadName* specifies the name of the thread.

---

## 5.3 Creating a Thread

---

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

### 5.3.1. Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

```
void start( )
```

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
```

```
class NewThread implements Runnable {
```

```
Thread t;
```

```
NewThread() {
```

```
// Create a new, second thread
```

```
t = new Thread(this, "Demo Thread");
```

```
System.out.println("Child thread: " + t);
```

```
t.start(); // Start the thread
```

```
}
```

```
// This is the entry point for the second thread.
```

```
public void run() {
```

```
try {
```

```
for(int i = 5; i > 0; i--) {
```

```
System.out.println("Child Thread: " + i);
```

```
Thread.sleep(500);
```

```
}
```

```
} catch (InterruptedException e) {
```

```
System.out.println("Child interrupted.");
```

```
}
```

```
System.out.println("Exiting child thread.");
```

```
}
```

```
}
```

```
class ThreadDemo {
```

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

```
new NewThread(); // create a new thread
```

```
try {
```

```

for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}

```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. After calling **start()**, **NewThread**'s constructor returns to **main()**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU in singlecore systems, until their loops finish. The output produced by this program is as follows.

(Your output may vary based upon the specific execution environment.)

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

### 5.3.2 Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```



```

}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}

```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

## 5.4 Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```

// Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
}
}

```

```

t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}

class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}

```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

One: 5  
Two: 5  
Three: 5  
One: 4  
Two: 4  
Three: 4  
One: 3  
Three: 3  
Two: 3  
One: 2  
Three: 2  
Two: 2  
One: 1  
Three: 1  
Two: 1  
One exiting.  
Two exiting.  
Three exiting.  
Main thread exiting.

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

### 5.4.1 Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)

A Higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to

be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run. To set a thread's priority, use the **setPriority()** method, which is a member of **Thread**. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN\_PRIORITY** and **MAX\_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM\_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**. You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

### 5.4.2 Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

### 5.4.2.1 Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

### 5.4.2.2 The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand

why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(objRef) {  
    // statements to be synchronized  
}
```

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run()** method:

```
// This program uses a synchronized block.  
class Callme {  
    void call(String msg) {  
        System.out.print "[" + msg;  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}  
  
class Caller implements Runnable {  
    String msg;  
    Callme target;  
    Thread t;  
    public Caller(Callme targ, String s) {  
        target = targ;
```

```

msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
}
}

```

Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

### 5.4.3 Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

Additional forms of **wait( )** exist that allow you to specify a period of time to wait. Before working through an example that illustrates interthread communication, an important point needs to be made. Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called,



there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify()** or **notifyAll()** having been called. (In essence, the thread resumes for no apparent reason.)

Because of this remote possibility, Oracle recommends that calls to **wait()** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses **wait()** and **notify()**. To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

// An incorrect implementation of a producer and consumer.

```
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

```

    }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

Although the **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```

Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2

```

Put: 3

Put: 4

Put: 5

Put: 6

Put: 7

Got: 7

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.

```
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
    }
```

```

this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();

```

```

new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}

```

Inside **get()**, **wait()** is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside **get()** resumes. After the data has been obtained, **get()** calls **notify()**. This tells **Producer** that it is okay to put more data in the queue. Inside **put()**, **wait()** suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify()** is called. This tells **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```

Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5

```

#### 5.4.4 Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo()** and **bar()**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition. The **foo()** and **bar()** methods use **sleep()** as a way to force the deadlock condition to occur.

#### // An example of deadlock.

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }
    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("B Interrupted");
        }
    }
}
```

```

    }
    System.out.println(name + " trying to call A.last()");
    a.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }
    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }
    public static void main(String args[]) {
        new Deadlock();
    }
}

```

When you run this program, you will see the output shown here:

MainThread entered A.foo

RacingThread entered B.bar

MainThread trying to call B.last()

RacingThread trying to call A.last()

Because the program has deadlocked, you need to press ctrl-c to end the program. You can see a full thread and monitor cache dump by pressing ctrl-break on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same

time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

## 5.5 Stopping and Blocking a Thread

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend()** method of the **Thread** class was deprecated by Java 2 several years ago. This was done because **suspend()** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The **resume()** method is also deprecated. It does not cause problems, but cannot be used without the **suspend()** method as its counterpart.

The **stop()** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can't now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method



periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to “running,” the **run()** method must continue to let the thread execute. If this variable is set to “suspend,” the thread must pause. If it is set to “stop,” the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait()** and **notify()** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread. The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread. Finally, the **main()** method has been modified to invoke the **mysuspend()** and **myresume()** methods.

#### **// Suspending and resuming a thread the modern way.**

```
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
```

```

Thread.sleep(200);
synchronized(this) {
while(suspendFlag) {
wait();
}
}
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
synchronized void mysuspend() {
suspendFlag = true;
}
synchronized void myresume() {
suspendFlag = false;
notify();
}
}

class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {
Thread.sleep(1000);
ob1.mysuspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");

```

```

Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}

```

When you run the program, you will see the threads suspend and resume. Although this mechanism isn't as "clean" as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

## 5.6 Life Cycle of a Thread

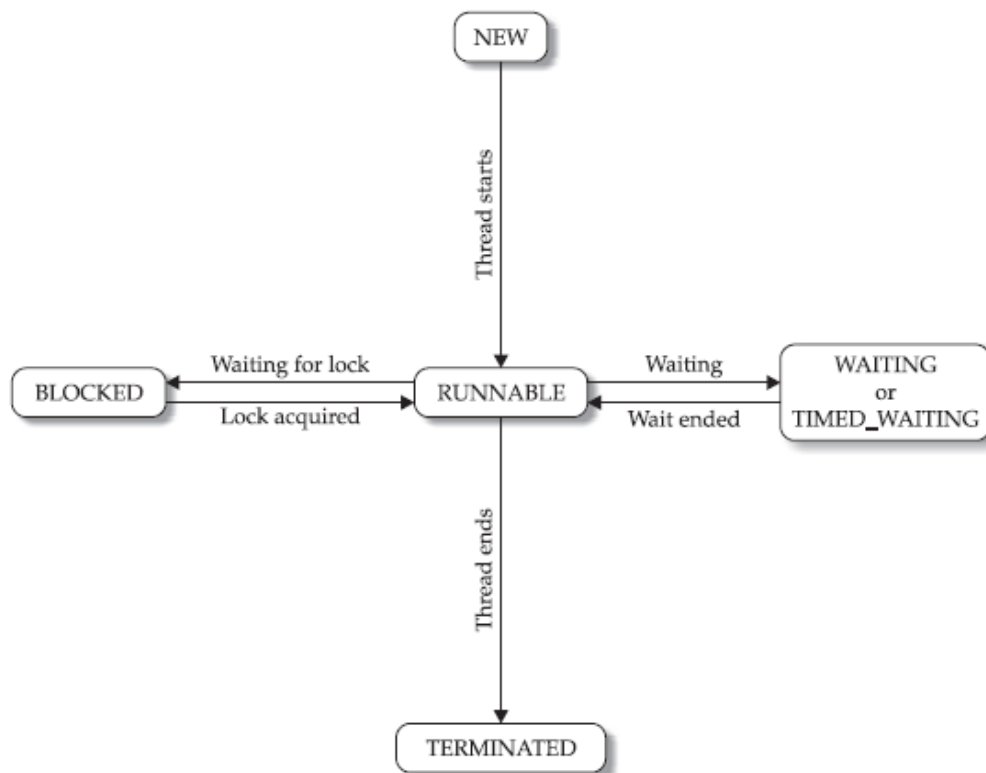
As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the **getState()** method defined by **Thread**. It is shown here:

**Thread.State** **getState()**

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. Here are the values that can be returned by **getState()**:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code> . This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code> .

Table 5.1. diagrams how the various thread states relate.



**Fig. 5.1** Thread states

Given a **Thread** instance, you can use `getState()` to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time `getState()` is called:

```

Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // ...

```

It is important to understand that a thread's state may change after the call to **getState()**. Thus, depending on the circumstances, the state obtained by calling **getState()** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState()** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

---

## 5.7 Check Your Progress

---

1. A .....program contains two or more parts that can run concurrently.
2. In a thread-based multitasking environment, the thread is the smallest unit of.....
3. Single-threaded systems use an approach called an.....
4. A thread can be .....by a higher-priority thread.
5. The way to create a thread is to create a new class that extends .....
6. When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time, this is called.....
7. ....method wakes up a thread that called wait() on the same object.
8. ....occurs when two threads have a circular dependency on a pair of synchronized objects.
9. The thread that has not begun execution, this state of thread is called.....
10. The thread that has completed execution, this state of thread is called .....

---

## 5.8 Summary

---

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

You create a thread by instantiating an object of type `Thread`. Java defines two ways in which this can be accomplished:

- You can implement the `Runnable` interface.
- You can extend the `Thread` class, itself.

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. The mechanisms to suspend, stop, and resume threads differ between a program used `suspend()`, `resume()`, and `stop()`, which are methods defined by `Thread` to pause, restart, and stop the execution of a thread.

The different states in a lifecycle of a thread are new, runnable, blocked and terminated and also a waiting state.

---

## 5.9 Keywords

---

**Multithreading-** It is a specialized form of multitasking.

**Interprocess-** This communication is expensive and limited.

**Blocked-** A thread can be blocked when waiting for a resource.

**Suspended-** A running thread can be suspended, which temporarily halts its activity.

**sleep()-** method causes the thread from which it is called to suspend execution for the specified period of milliseconds

**Runnable-** To create a thread is to create a class that implements the Runnable interface.

**Thread-** To create a thread is to create a new class that extends Thread.

**Deadlock-** which occurs when two threads have a circular dependency on a pair of synchronized objects.

---

## 5.10 Self-Assessment Test

---

- Q.1. How will you define thread priorities in case of Java thread model? Explain.
- Q.2. How synchronization is important in thread model? Describe.
- Q.3. Explain the concept of Main Thread with help of a program.
- Q.4. Write down a program to explain the concept of implementing a interface.
- Q.5. Write down a program for creating multiple threads.
- Q.6. Explain the concept of deadlock.
- Q.7. Explain the lifecycle of a thread.

---

## **5.11 Answers to check your progress**

---

1. multithreaded
2. dispatchable code
3. event loop with polling
4. preempted
5. Thread
6. synchronization
7. notify( )
8. Deadlock
9. New State
10. Terminated State

---

## **5.12 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.



SUBJECT: JAVA PROGRAMMING	
COURSE CODE: MCA-13	AUTHOR: AYUSH SHARMA
LESSON NO. 6	
Exception Handling	

## STRUCTURE

- 6.0 Learning Objective
- 6.1 Introduction
- 6.2 Types of Errors
  - 6.2.1 Compile-Time Errors
  - 6.2.2 Run-Time Errors
- 6.3 Exceptions
- 6.4 Exceptions Hierarchy
- 6.5 Syntax of Exception Handling Code
- 6.6 Multiple Catch Statements
- 6.7 Using finally Statement
- 6.8 Throwing Our Own Exceptions
- 6.9 Check Your Progress
- 6.10 Summary
- 6.11 Keywords
- 6.12 Self-Assessment Test
- 6.13 Answers to check your progress
- 6.14 References / Suggested Readings

### 6.0 LEARNING OBJECTIVE

After going through this unit, you will be able to:

- learn the concept of exceptions in Java

- learn to use the keywords *throw*, *try*, *catch*, and *finally* in exception handling
- learn about the *Throwable* class hierarchy
- learn about unchecked and checked exception in Java
- throw exceptions implicitly as well as explicitly
- learn about how to catch exceptions
- learn to handle user-defined exceptions and will be able to create your own exception

---

## 6.1 INTRODUCTION

---

This chapter examines Java's exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

By now, you must have been acquainted with so many concepts of Java programming language such as *data types and variables*, *operators*, *control flow statements* etc. We have also presented the concept of classes, objects, methods, constructors, inheritance, various types of modifiers, arrays, strings, vectors as well as the interfaces and packages. These concepts will help the learners to write and develop suitable programs. While writing program there may arise some errors for some mistakes. An error may produce an incorrect output or may terminate the execution of the program abruptly. It is therefore important to detect and manage those errors. Java facilitates the management of such situation by handling exceptions.

In this chapter, we will discuss how exceptions can be handled in Java programming language. The chapter describes when and how to use exceptions.

---

## 6.2 Types of Error

---

We can define an *exception* as an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. The Java programming language uses *exceptions* to handle errors and other exceptional events. Exceptions are used

in a program to signal that some error or exceptional situation has occurred, and that it does not make sense to continue the program flow until the exception has been handled.

There are two categories of errors: *Compile-time errors* and *Run-time errors*.

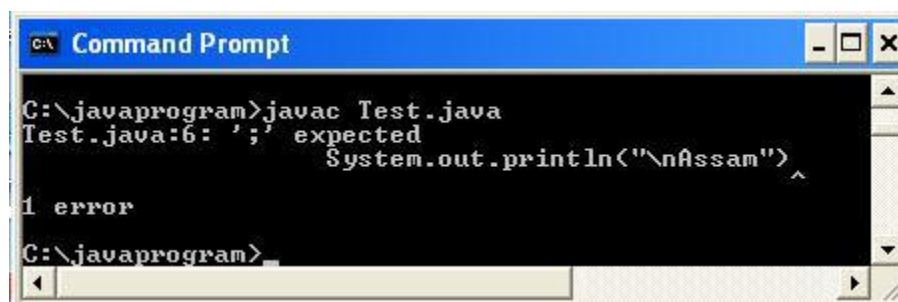
### 6.2.1 Compile-time errors

All syntax errors detected and displayed by the Java compiler are termed as *Compile-time errors*. If the compiler detects an error while compiling a program, then the *.class* file will not be created. For successful compilation we have to correct the syntax error first. Let us consider the following example for the demonstration of compile time error:

**Program 6.1:** Test.java (Program showing Compile-time error)

```
class Test
{
public static void main(String args[ ])
{
System.out.println("KKHSOU");
System.out.println("\nAssam")
}
}
```

While compiling the above program *Test.java*, the following message will be displayed on the screen:



We can see that the statement

**System.out.println("\nAssam")**

has no semicolon at the end. The Java compiler displays where the errors are in the program.

We can then correct the errors in the appropriate line and recompile the program. If there

is no other error in the program then it will create **.class** file (here, *Test.java*). We can then run the program to see the output. Some of the most common compile-time errors are:

- Use of variable without declaration
- Missing brackets in classes and methods
- Missing semicolons
- Incompatible assignment statements etc.

### 6.2.2. Run-time errors

Sometimes, a program may compile successfully creating **.class** file but may not execute properly i.e., they may produce wrong result or may terminate abruptly. These errors may occur due to wrong logic of the program and many more reasons like

- Dividing an integer by zero
- Converting invalid strings to number
- Trying to store a value into an array of incompatible class or type etc.

Such types of errors are termed as **Run-time errors**. Let us consider the following example for the demonstration of run-time error:

**//Program 6.2:** Error.java (Demonstration of Run-time error and

// Implicitly throwing an exception)

```
class Error
{
public static void main(String args[])
{
int p, q, r, result;
p = 20;
q = 6;
r = 6;
result = p / (q-r); //(q-r) is equal to zero
System.out.println("The result is : " + result);
}
}
```

When we compile the above program, then it will create the **Error.class** file as the compilation is successful. The compiler will not display any error message as there is no

syntax error in the code. When we try to run the program then it will display the following error message and terminate the execution of the program. In the statement **Result = p/(q-r);** we are dividing p by zero. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred)



```
C:\javaprogram>javac Error.java
C:\javaprogram>java Error
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Error.main(Error.java:11)
C:\javaprogram>
```

## 6.3 Exceptions

An **exception** in Java is an object that is created when an abnormal situation arises in a program. When an error occurs within a method, the method creates an object and hands it off to the runtime system. This exception object has data members that store information about the nature of the problem. Such an object can be instantiated by a running program in two ways:

- *explicitly* by a **throw** statement in the program
- or *implicitly* by the Java run-time system when it is unable to execute a statement in a program(as in *Program 6.2*).

One major benefit of having an error signaled by an exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly. Another positive aspect of exceptions is that they provide a way of enforcing a response to particular errors – with many kinds of exceptions, we must include code in our program to deal with them, otherwise our code will not compile. One important idea to grasp is that not all errors in our programs need to be signaled by exceptions.

At this point we will discuss the basics of Java's exception throwing and catching mechanism. When an error occurs in a Java program it usually results in an exception being thrown. A method may throw an exception for many reasons, for instance if the input

parameters are invalid (negative when expecting positive etc.). How we *throw*, *catch* and *handle* these exception matters. There are several different ways to do so.

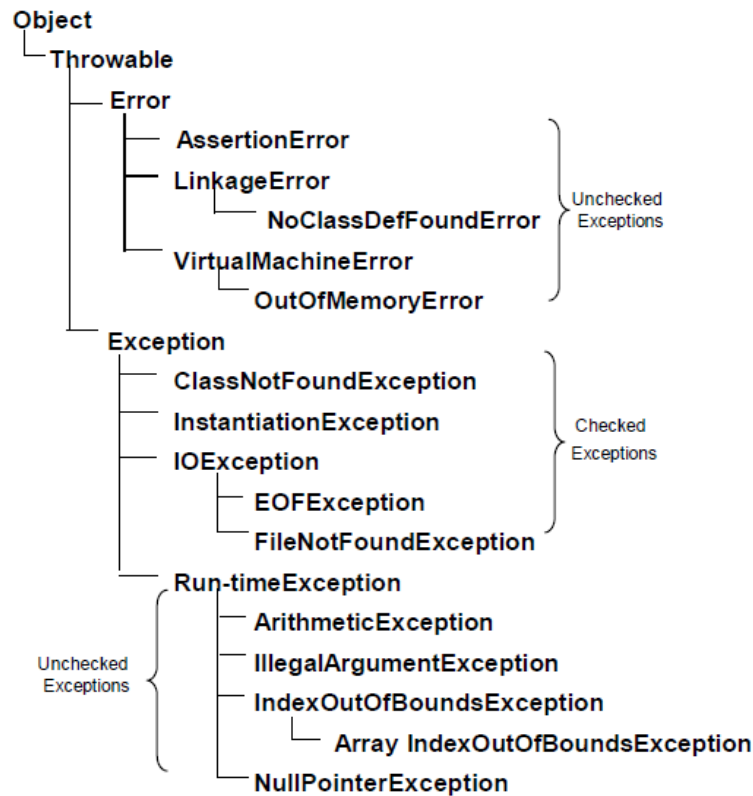
Creating an exception object and handing it to the runtime system is called *throwing an exception*. When an exception is thrown, it can be caught by a *catch* clause of *try* statement. If the exception object is not caught and handled properly, the interpreter will display an error message (e.g., like the output of *Program 6.2*) and terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and display an appropriate message. This process is known as *exception handling*.

---

## 6.4 Exceptions Hierarchy

---

The hierarchy of exception classes commence from *Throwable* class which is the base class for an entire family of exception classes, declared in **java.lang** package as **java.lang.Throwable**. An exception is always an object of some subclass of the standard class *Throwable*. Two direct subclasses of the *Throwable* class are the *Error* class and the *Exception* class which cover all the standard exceptions. Both these classes themselves have subclasses which identify specific exception conditions. The following figure 6.1 shows the exception hierarchy which gives some of the standard exception in Java.



**Fig. 6.1 Exception Hierarchy in Java**

### 6.4.1 Checked versus Unchecked Exceptions

In Java there are basically two types of built-in exceptions: *Unchecked exceptions* and *checked exceptions*. Exception in Java are classified on the basis of the exception handled by the java compiler.

#### • Unchecked Exceptions

The kinds of exception that can be prevented by writing better code are *unchecked exceptions*. They are instances of the **Error** class, the **RuntimeException** class, and their extensions. These exception arises during run-time ,that occur due to invalid argument passed to method.

#### • Checked Exceptions

The *checked exceptions* are checked by the compiler before the program is run. At compile time, the Java compiler checks that a program contains handlers for checked exceptions.

These exception are the object of the **Exception** class or any of its subclasses except *RuntimeException* class. These condition arises due to invalid input, problem with our network connectivity and problem in database. The statements that throw them either must be placed within **try** statement or they must be declared in their **method's header**.

The **java.lang** package defines several classes and exceptions. Some of these classes are not checked while some other classes are checked (Table 6.1).

EXCEPTIONS	DESCRIPTION	CHECKED	UNCHECKED
ArithmeticException	Arithmetic errors such as a divide by zero	-	YES
ArrayIndexOutOfBoundsException	Arrays index is not within array.length	-	YES
ClassNotFoundException	Related Class not found	YES	-
IOException	InputOutput field not found	YES	-
IllegalArgumentException	Illegal argument when calling a method	-	YES
InterruptedException	One thread has been interrupted by another thread	YES	-
NoSuchMethodException	Nonexistent method	YES	-
NullPointerException	Invalid use of null reference	-	YES
NumberFormatException	Invalid string for conversion to number	-	YES

Table 6.1

## 6.5 Syntax of Exception Handling Code

If we want to deal with the exceptions where they occur, there are three kinds of code block that we can include in a method to handle them. These are **try**, **catch**, and **finally**. At this point we will first discuss the detail of **try** and **catch** blocks and will come to the application of a **finally** block a little later.



## • The try Block

When we want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a **try** block. A **try** block is simply the keyword *try*, followed by braces enclosing the code that can throw the exception:

```
try
{
// Code that can throw one or more exceptions
}
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block. It should be remembered that every **try** statement should be followed by at least one **catch** statement if there is no *finally* block.

## • The catch Block

We enclose the code to handle an exception of a given type in a **catch** block. The catch block must immediately follow the try block that contains the code that may throw that particular exception. A **catch** block consists of the keyword *catch* followed by a parameter between parentheses that identifies the type of exception that the block is to deal with. This is followed by the code to handle the exception enclosed between braces:

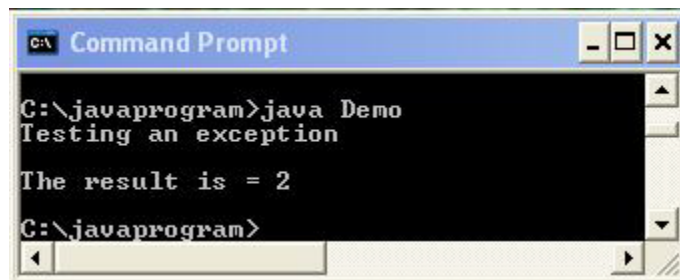
```
try
{
// Code that can throw one or more exceptions
}
catch( ArithmeticException e)
{
// Code to handle the exception
}
```

The above catch block only handles *ArithmeticException* exceptions. This implies that, this is the only kind of exception that can be thrown in the try block. If others can be thrown, this will not compile. Let us modify *Program 6.2* for the demonstration of try and catch blocks to handle an arithmetic expression.

**//Program 6.3:** Demo.java

```
class Demo
{
public static void main(String args[])
{
int p, q, r, x, y;
p = 20;
q = 5;
r = 5;
try
{
x = p / (q-r); //exception here
}
catch(ArithmeticException e)
{
System.out.println("Testing an exception");
}
y = p / (q+r);
System.out.println("\nThe result is = "+y);
}
}
```

If we run the above program, the following output will be displayed.



```
C:\ Command Prompt
C:\javaprogram>java Demo
Testing an exception

The result is = 2
C:\javaprogram>
```

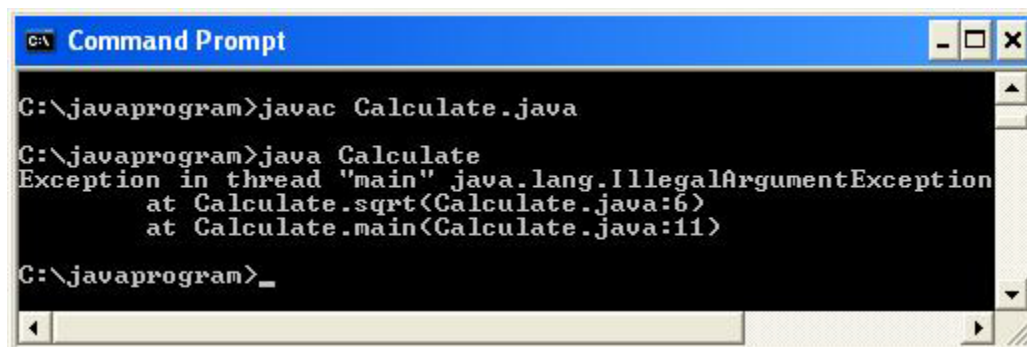
The execution of the program does not stop at the point of exceptional condition. It catches the error condition, displays the message “Testing an exception”. The execution continues and gives the result without terminating the program as if nothing has happened.

Program 6.3 is an example of implicitly throwing and catching an unchecked exception. Let us consider the following program for an illustration of unchecked exception that is thrown by an *explicit* throw statement.

**//Program 6.4:** Calculate.java (Explicit throw of unchecked exception)

```
class Calculate
{
    static double sqrt(double n)
    {
        if(n<0)
            throw new IllegalArgumentException();
        return Math.sqrt(n);
    }
    public static void main(String[ ] args)
    {
        System.out.println(sqrt(-16));
        System.out.println("\nEnd of Calculate Method");
    }
}
```

When we run the program after compiling, then it will display the following and terminate the program.



```
C:\ Command Prompt

C:\javaprogram>javac Calculate.java

C:\javaprogram>java Calculate
Exception in thread "main" java.lang.IllegalArgumentException
    at Calculate.sqrt(Calculate.java:6)
    at Calculate.main(Calculate.java:11)

C:\javaprogram>_
```

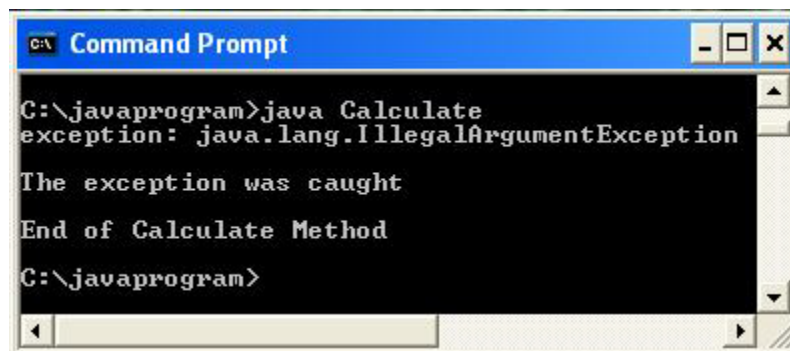
output is showing what happens when an exception is not caught. It can be prevented with the help of try and catch statement. The following program is a modification of Program 6.4 to handle such unchecked exception.

*//Program 6.5:* Calculate.java (Catching an unchecked exception

*//which is thrown explicitly)*

```
class Calculate
{
    static double sqrt(double n)
    {
        if(n<0)
            throw new IllegalArgumentException();
        return Math.sqrt(n);
    }
    public static void main(String[ ] args)
    {
        try
        {
            System.out.println(sqrt(-16));
        }
        catch(Exception exception)
        {
            System.out.println("exception: " + exception);
        }
        System.out.println("\nThe exception was caught");
        System.out.println("\nEnd of Calculate Method");
    }
}
```

The output will be like this:



```
C:\ Command Prompt
C:\javaprogram>java Calculate
exception: java.lang.IllegalArgumentException
The exception was caught
End of Calculate Method
C:\javaprogram>
```

The *IllegalArgumentException* object is thrown with the statement **throw new IllegalArgumentException();** and that exception is caught with the statement **catch(Exception exception)** as it is generated by the statement **System.out.println(sqrt(-16));** within that try block. The exception in the Program 6.5 can be handled by writing proper program code.

Normally, the try statement should be used only for checked exceptions. That is because the purpose of try statement is to handle unanticipated errors. Exceptions should be reserved for the unusual or catastrophic situations that can arise. The reason for this is that dealing with exceptions involves quite a lot of processing overhead, so if our program is handling exceptions a lot of the time it will be a lot slower than it needs to be.

Following are the list of various checked exception that defined in the **java.lang** package.

Exception	Reason for Exception
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program
Instantiation Exception	This Exception occurs when you create an object of an abstract class and interface
Illegal Access Exception	This Exception occurs when you create an object of an abstract class and interface.
Not Such Method Exception	This Exception occurs when the method you call does not exist in class

## 6.6 Multiple Catch Statements

If a *try* block can throw several different kinds of exception, we can put several *catch* blocks after the try block to handle them.

```
try
{
    // Code that may throw exceptions
}
catch(ArithmeticException e)
{
    // Code for handling ArithmeticException exceptions
}
catch(IndexOutOfBoundsException e)
{
    // Code for handling IndexOutOfBoundsException exceptions
}
// Execution continues here...
```

Exceptions of type *ArithmeticException* will be caught by the first catch block, and exceptions of type *IndexOutOfBoundsException* will be caught by the second. Of course, if an *ArithmeticException* exception is thrown, only the code in that catch block will be executed. When it is complete, execution continues with the statement following the last catch block. When we need to catch exceptions of several different types for a try block, the order of the catch blocks is important. When an exception is thrown, it will be caught by the first catch block that has a parameter type that is the same as that of the exception, or a type that is a superclass of the type of the exception.

An extreme case would be if we specify the catch block parameter as type *Exception*. This will catch any exception that is of type *Exception*, or of a class type that is derived from *Exception*. This includes virtually all the exceptions we are likely to meet in the normal course of events. This has implications for multiple catch blocks relating to exception class types in a hierarchy. The catch blocks must be in sequence with the most *derived type first*, and the *most basic type last*. Otherwise our code will not compile. The simple reason for this is that if a catch block for a given class type precedes a catch block for a type that is

derived from the first, the second catch block can never be executed and the compiler will detect that this is the case

Suppose we have a catch block for exceptions of type *ArithmeticException*, and another for exceptions of type *Exception* . If we write them in the following sequence, exceptions of type *ArithmeticException* could never reach the second catch block as they will always be caught by the first.

```
// Invalid catch block sequence – will not compile!
```

```
try
{
// try block code
}
catch(Exception e)
{
// Generic handling of exceptions
}
catch(ArithmeticException e)
{
// Specialized handling for these exceptions
}
```

The above code will not compile. Thus if we have catch blocks for several exception types in the same class hierarchy, we must put the catch blocks in order, starting with the lowest subclass first, and then progressing to the highest superclass. In principle, if you are only interested in generic exceptions, all the error handling code can be localized in one catch block for exceptions of the superclass type. However, in general it is more useful, and better practice, to have a catch block for each of the specific types of exceptions that a try block can throw.

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block. The following example traps two different exception types:

**// Program 6.7 Demonstrate multiple catch statements.**

```
class MultipleCatches {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[] = { 1 };  
            c[42] = 99;  
        } catch(ArithmeticException e) {  
            System.out.println("Divide by 0: " + e);  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index oob: " + e);  
        }  
        System.out.println("After try/catch blocks.");  
    }  
}
```

This program will cause a division-by-zero exception if it is started with no commandline arguments, since **a** will equal zero. It will survive the division if you provide a commandline argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

**Here is the output generated by running it both ways:**

```
C:\>java MultipleCatches
```

```
a = 0
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
```

```
C:\>java MultipleCatches TestArg
```

```
a = 1
```

```
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
```

```
After try/catch blocks.
```



When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.
```

```
    A subclass must come before its superclass in a series of catch statements. If not,  
    unreachable code will be created and a compile-time error will result.
```

```
*/
```

```
class SuperSubCatch {  
    public static void main(String args[]) {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        } catch(Exception e) {  
            System.out.println("Generic Exception catch.");  
        }  
        /* This catch is never reached because  
        ArithmeticException is a subclass of Exception. */  
        catch(ArithmeticException e) { // ERROR – unreachable  
            System.out.println("This is never reached.");  
        }  
    }  
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

## 6.7 Using finally Statement

The immediate nature of an exception being thrown means that execution of the try block code breaks off, regardless of the importance of the code that follows the point at which the exception was thrown. This introduces the possibility that the exception leaves things in an unsatisfactory state. We might have opened a file, for instance, and because an exception was thrown, the code to close the file is not executed.

The *finally* block provides the means to clean up at the end of executing a try block. We use a finally block when we need to be sure that some particular code is run before a method returns, no matter what exceptions are thrown within the previous try block. A finally block is always executed, regardless of what happens during the execution of the method. If a file needs to be closed, or a critical resource released, we can guarantee that it will be done if the code to do it is put in a *finally* block. The finally block has a very simple structure:

**finally**

```
{  
// Clean-up code to be executed last  
}
```

Just like a catch block, a finally block is associated with a particular try block, and it must be located immediately following any catch blocks for the try block. If there are no catch blocks then we position the finally block immediately after the try block. If we do not do this, our program will not compile. Java provides the *finally* statement that can be used handle an exception that is not caught by any of the previous catch statement. i.e., a try statement does not have to have a catch block if it has a *finally* block. If the code in the try statement has multiple exit points and no associated catch clauses, the code in the finally block is executed no matter how the try block is exited. Thus, it makes sense to provide a finally block whenever there is code that must always be executed.

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes

it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

**finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

// **Program 6.8** Demonstrate finally.

```
class FinallyDemo {  
    // Throw an exception out of the method.  
    static void procA() {  
        try {  
            System.out.println("inside procA");  
            throw new RuntimeException("demo");  
        } finally {  
            System.out.println("procA's finally");  
        }  
    }  
    // Return from within a try block.  
    static void procB() {  
        try {  
            System.out.println("inside procB");  
            return;  
        } finally {  
            System.out.println("procB's finally");  
        }  
    }  
}
```

```

}
}
// Execute a try block normally.
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}

public static void main(String args[]) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Exception caught");
    }
    procB();
    procC();
}
}

```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

**Here is the output generated by the preceding program:**

```

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

```

## 6.8 Throwing Our Own Exceptions

As we come across *Built-in* exception, we create our own customized exception as per requirements of the application. On each application there is a specific constraint. Error-handling becomes necessary while developing a constraint application. For example, suppose in the case of a banking application, a customer whose age is less than 18 needs to open a Joint Account. The Exception class and its subclass in Java are not able to meet up the required constraint in application. For this, we create our own customized exception to address these constraints and ensure the integrity in the application. Let us see how to handle and create *user-defined* exception. The keywords `try`, `catch` and `finally` are used in implementing user-defined exceptions. This ***Exception*** class inherits all the methods from the ***Throwable*** class.

In the following program, a class ***MyException*** is created which is a subclass of the ***Exception*** class. The ***MyException*** class has one constructor, i.e. ***MyException()***.

*//Program 6.9: UserDefinedException.java*

```
import java.lang.Exception;

class MyException extends Exception
{
    MyException(String m)
    {
        super(m);
    }
}

class UserDefinedException
{
    public static void main(String args[ ])
    {
        int a=5, b=5000;
        try
        {
            float c =(float)a/(float)b;
            if(c<0.01)
```

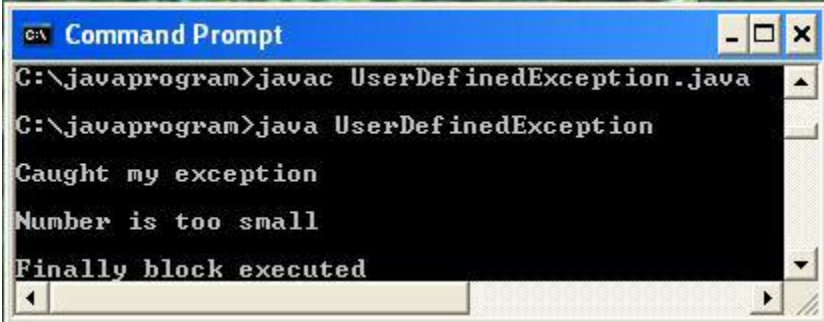
```

{
throw new MyException("\nNumber is too small");
}
}
catch(MyException e)
{
System.out.println("\nCaught my exception");
System.out.println(e.getMessage());
}
finally
{
System.out.println("\nFinally block executed");
}
}
}

```

The object **e** which contain the error message “*Number is too small*” is caught by the catch block which then displays the message using the *getMessage()* method. The output will be like this:

We can also learn how to use the statement *finally* with the above program. The last line of the output is produced by the *finally* block.



```

C:\ Command Prompt
C:\javaprogram>javac UserDefinedException.java
C:\javaprogram>java UserDefinedException
Caught my exception
Number is too small
Finally block executed

```

---

## 6.9 Check Your Progress

---

1. The two categories of errors: Compile-time errors and .....
2. All syntax errors detected and displayed by the Java compiler are termed as .....
3. Dividing an integer by zero is a type of.....
4. Two direct subclasses of the Throwable class are the Error class and the.....
5. The basically two types of built-in exceptions: Unchecked exceptions and.....
6. The kinds of exception that can be prevented by writing better code are .....
7. ....are checked by the compiler before the program is run.
8. When we want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a .....block.
9. We enclose the code to handle an exception of a given type in a ..... block.
10. The ..... block provides the means to clean up at the end of executing a try block.

---

## 6.10 Summary

---

We can define an exception as an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. The Java programming language uses exceptions to handle errors and other exceptional events. Exceptions are used in a program to signal that some error or exceptional situation has occurred, and that it does not make sense to continue the program flow until the exception has been handled.

There are two categories of errors: Compile-time errors and Run-time errors.

All syntax errors detected and displayed by the Java compiler are termed as Compile-time errors. If the compiler detects an error while compiling a program, then the .class file will not be created.

Sometimes, a program may compile successfully creating .class file but may not execute properly i.e., they may produce wrong result or may terminate abruptly. These errors may occur due to wrong logic of the program and many more reasons like

- Dividing an integer by zero
- Converting invalid strings to number
- Trying to store a value into an array of incompatible class or type etc.

An exception in Java is an object that is created when an abnormal situation arises in a program. When an error occurs within a method, the method creates an object and hands it off to the runtime system. This exception object has data members that store information about the nature of the problem. Such an object can be instantiated by a running program in two ways:

- explicitly by a throw statement in the program
- or implicitly by the Java run-time system when it is unable

The hierarchy of exception classes commence from Throwable class which is the base class for an entire family of exception classes, declared in java.lang package as java.lang.Throwable. An exception is always an object of some subclass of the standard class Throwable.

In Java there are basically two types of built-in exceptions: Unchecked exceptions and checked exceptions. Exception in Java are classified on the basis of the exception handled by the java compiler.

When we want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a try block. The catch block must immediately follow the try block that contains the code that may throw that particular exception.

The finally block provides the means to clean up at the end of executing a try block. We use a finally block when we need to be sure that some particular code is run before a method returns, no matter what exceptions are thrown within the previous try block.

---

## 6.11 Keywords

---

**Compile-Time Errors-** All syntax errors detected and displayed by the Java compiler are termed as Compile-time errors.

**Run-Time Errors-** These errors may occur due to wrong logic of the program and many more reasons.



**Exception-** An exception in Java is an object that is created when an abnormal situation arises in a program.

**Explicitly-** Exception by a throw statement in the program.

**Implicitly-** Exception by the Java run-time system when it is unable.

**Unchecked Exceptions-** The kinds of exception that can be prevented by writing better code are unchecked exceptions.

**Checked Exceptions-** The checked exceptions are checked by the compiler before the program is run.

**try-** When we want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a try block

**catch-** We enclose the code to handle an exception of a given type in a catch block.

**finally-** The finally block provides the means to clean up at the end of executing a try block.

---

## 6.12 Self-Assessment Test

---

- Q.1 What do you understand by error in java? Explain the types of error.
- Q.2 Explain run-time error with the help of a program.
- Q.3 Differentiate between error and exception in java.
- Q.4 Explain the hierarchy of exception in java.
- Q.5 Differentiate between checked and unchecked exceptions with example.
- Q.6 What do you understand by try and catch block? Explain.
- Q.7 Explain the concept of multiple catch statements.
- Q.8 Explain the concept of finally keyword.

---

## 6.13 Answers to check your progress

---

1. Run-time errors

2. Compile-time errors
3. run-time error
4. Exception class
5. checked exceptions
6. unchecked exceptions
7. Checked exceptions
8. try
9. catch
10. finally

---

## **6.14 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.

SUBJECT: JAVA PROGRAMMING	
COURSE CODE: MCA-13	AUTHOR: AYUSH SHARMA
LESSON NO. 7	
File Handling	

## STRUCTURE

- 7.0 Learning Objective
- 7.1 Introduction
- 7.2 I/O Basics: Streams
- 7.3 The Stream Classes
- 7.4 The Predefined Streams
- 7.5 Reading Console Input
- 7.6 Writing Console Output
- 7.7 Reading and Writing Files
- 7.8 Check Your Progress
- 7.9 Summary
- 7.10 Keywords
- 7.11 Self-Assessment Test
- 7.12 Answers to check your progress
- 7.13 References / Suggested Readings

## 7.0 LEARNING OBJECTIVE

After going through this unit, you will be able to:

- learn about the most important io package in Java
- learn about the predefined streams
- learn about console input and output
- describe how to read data from file and how to write data to file

## 7.1 INTRODUCTION

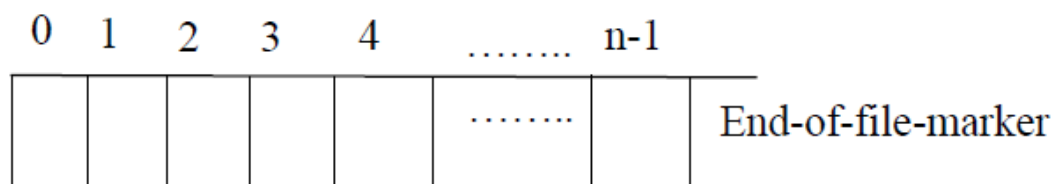
In this unit we will learn one of the Java's most important package **io**. We will learn about the streams, different stream classes. Besides this how to read data from input and how to write data into output. We also give a brief introduction of file handling in Java.

The **io** package supports Java's basic I/O (Input/Output) system including file I/O. Java program perform I/O through streams. A stream is linked to a physical device by the Java I/O system. Java define 2 types of stream, *byte* and *character*. In Java 1.0, the only way to perform console input was to use a byte stream. The preferred method of reading console input for Java 2 is use a character-oriented stream, which makes the program easier to internationalize and maintain.

Java provides a number of classes and methods that allow to read and write files. In Java all files are byte oriented, and Java provides method to read and write bytes from and to a file.

## 7.2 I/O BASICS: STREAMS

Java views each file as a sequential stream of bytes. Each operating system provides a mechanism to determine the end of a file, such as an end-of-file marker or count of the total bytes in the file that is recorded in a system maintained administrative data structure.

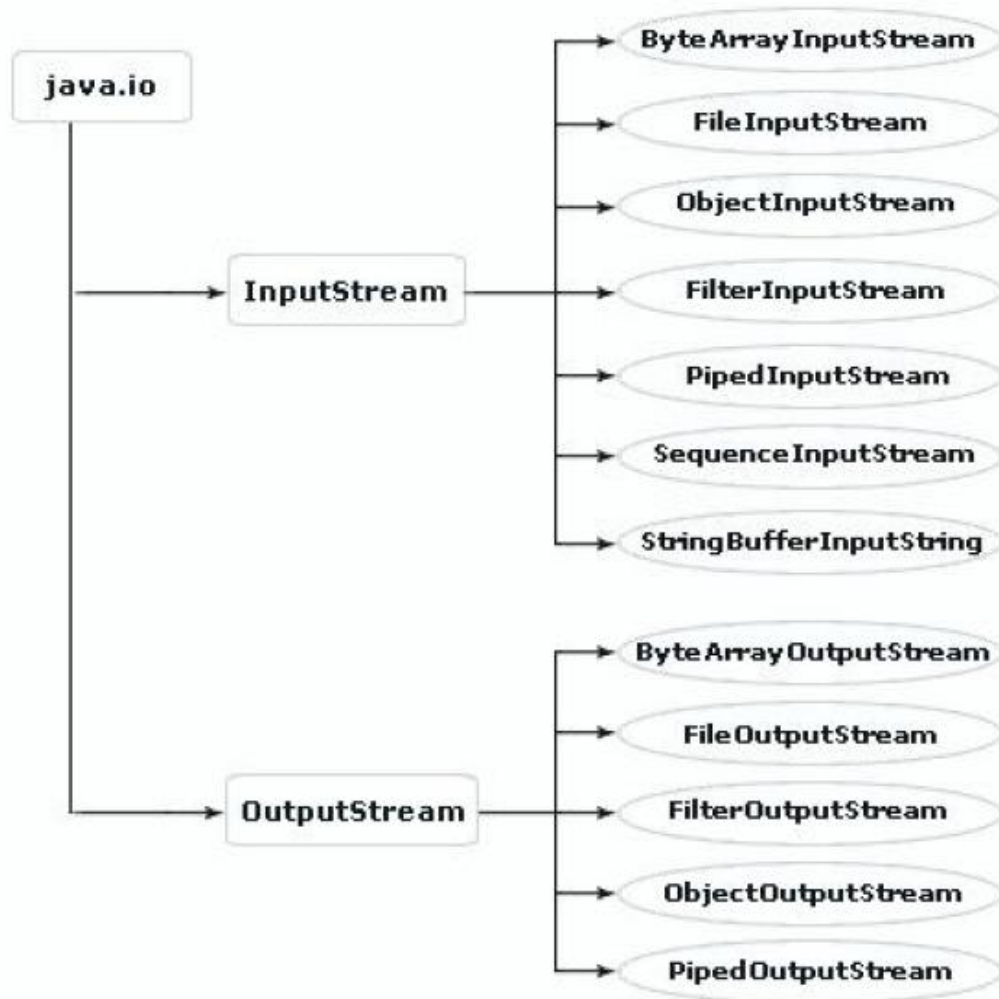


A Java program processing a stream of bytes simply receives an indication from the operating system when the program reaches the end of the stream the program does not need to know how the underlying platform represents files or streams.

The Java **Input/Output (I/O)** is a part of **java.io** package. The **java.io** package contains a relatively large number of classes that support input and output operations. The classes in

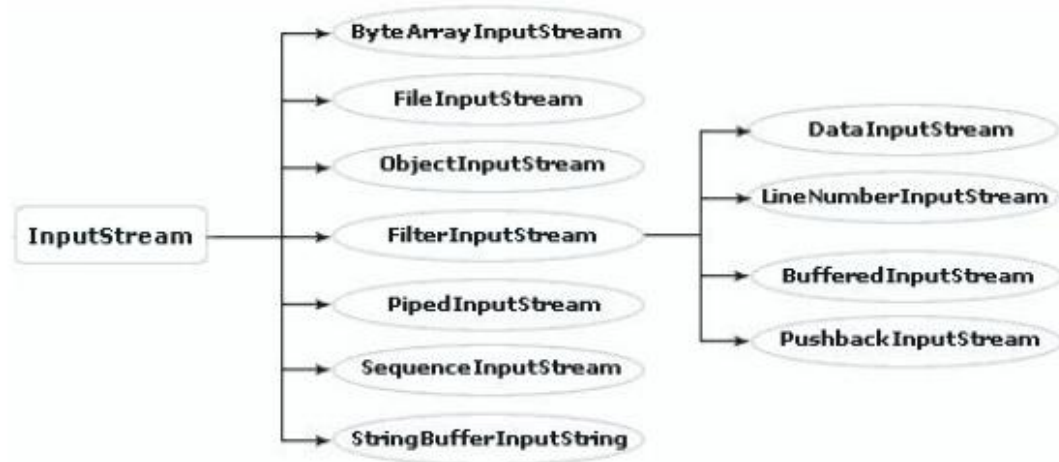
the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The `InputStream` and `OutputStream` are central classes in the package which are used for reading from and writing to byte streams, respectively.

The ***java.io*** package can be categorised along with its stream classes in a hierarchy structure as shown below:



The ***InputStream*** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a file, a string, or memory that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when we create it. We can explicitly close a stream with the `close( )` method, or let it be closed implicitly when the object is found as a garbage.

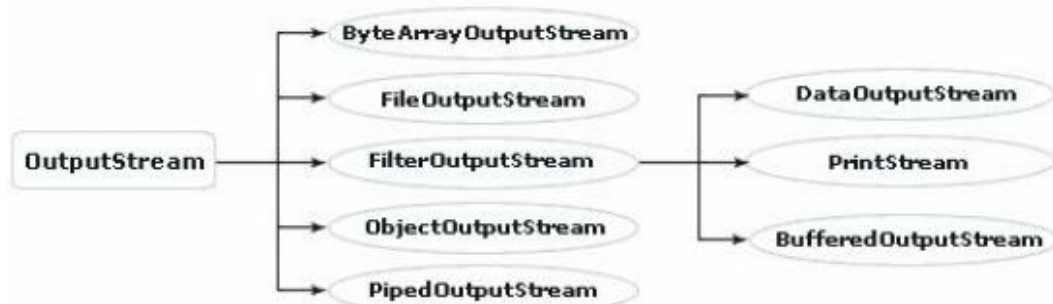
The subclasses inherited from the *InputStream* class can be seen in a hierarchy manner as shown below:



*InputStream* is inherited from the *Object* class. Each class of the *InputStreams* provided by the *java.io* package is intended for a different purpose.

The *OutputStream* class is a sibling to *InputStream* that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the *close()* method, or let it be closed implicitly when the object is garbage collected.

The classes inherited from the *OutputStream* class can be seen in a hierarchy structure shown below:

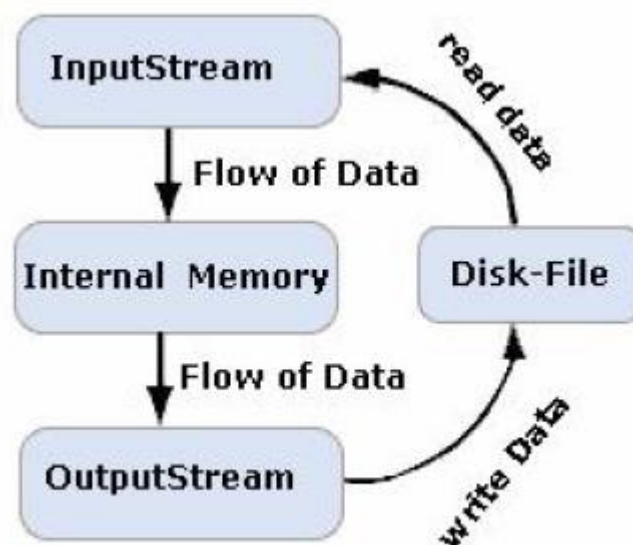


OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.

### How Files and Streams Work:

Java uses streams to handle I/O operations through which the data is flowed from one location to another. For example, an InputStream can flow the data from a disk file to the internal memory and an OutputStream can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file. When we work with a text file, we use a character stream where one character is treated as per byte on disk. When we work with a binary file, we use a binary stream.

The working process of the I/O streams can be shown in the given diagram.



## 7.3 THE STREAM CLASSES

There are two types of streams

1. Byte –for Binary I/O
2. Character – for Character I/O

Programs use byte streams to perform input and output of 8-bit bytes.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way, they differ mainly in the way they are constructed.

We'll explore `FileInputStream` and `FileOutputStream` by examining an example program named `CopyBytes.java`, which uses byte streams to copy `kkhsou.txt`, one byte at a time and write the content of the file on `outagain.txt`.

### **Kkhsou.java**

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("kkhsou.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```



```
}
```

We need to create one file `kkhsou.txt` which contains the text “DISHPUR GUWAHATI”. The program will copy this text into a new file called `outagain.txt`.

`CopyBytes` spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.

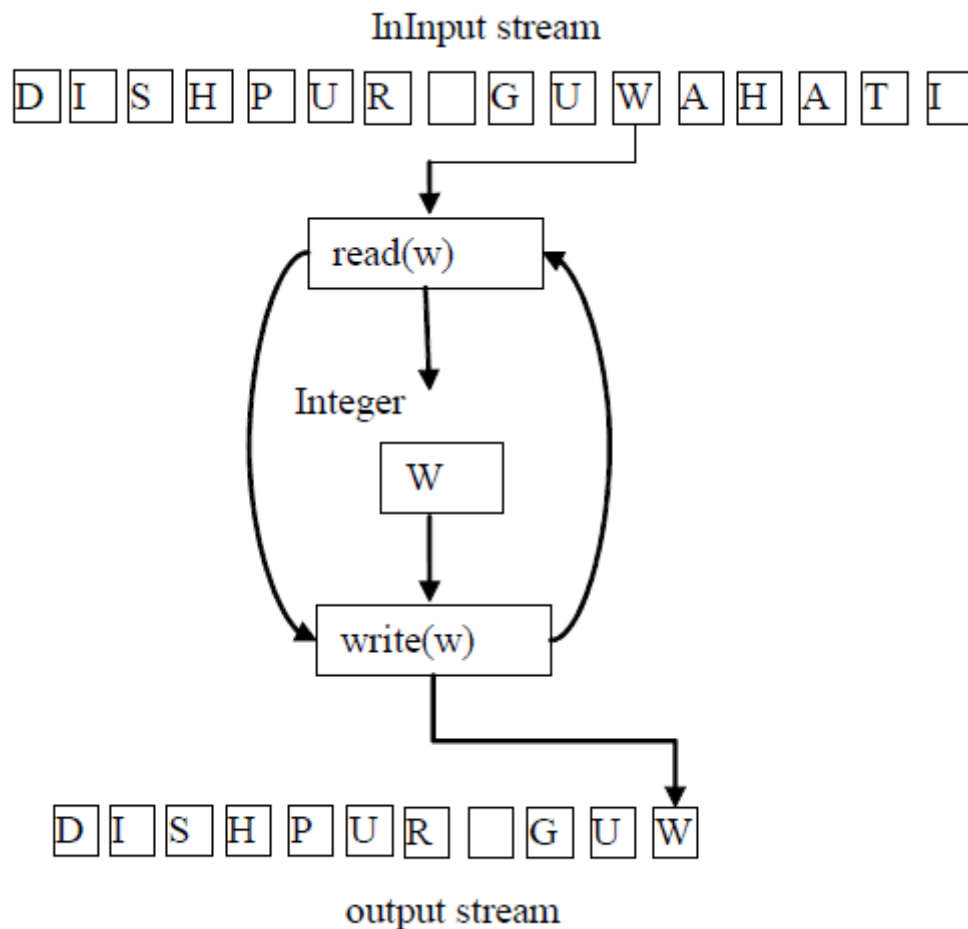


Fig 7.1: Simple byte stream input and output.

We can notice that `read()` returns an `int` value. Using a `int` as a return type allows `read()` to use `-1` to indicate that it has reached the end of the stream.

The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.

Java represents strings in *Unicode*, an International standard character encoding that is capable of representing most of the world's written languages. Typical human-readable text files, however, use encodings that are not necessarily related to Unicode, or even to ASCII, and there are many such encodings. Character streams hide the complexity of dealing with these encodings by providing two classes that serve as bridges between byte streams and character streams. The `InputStreamReader` class implements a character-input stream that reads bytes from a byte-input stream and converts them to characters according to a specified encoding.

Similarly, the `OutputStreamWriter` class implements a character-output stream that converts characters into bytes according a specified encoding and writes them to a byte-output stream.

A second advantage of character streams is that they are potentially much more efficient than byte streams. The implementations of many of Java's original byte streams are oriented around byte-at-a-time read and write operations. The character-stream classes, in contrast, are oriented around buffer-at-a-time read and write operations. This difference, in combination with a more efficient locking scheme, allows the character stream classes to more than make up for the added overhead of encoding conversion in many cases.

---

## 7.4 THE PREDEFINED STREAMS

---

All Java programs automatically import **java.lang** package. This package defines a class called `System`, which encapsulates several aspects of the run-time environment. For example, using some of its method we can obtain the current time and the settings of various properties associated within the system. `System` also contains three predefined system variables, *in*, *out* and *err*.

`System.out` refers to the standard output stream, which is console by default. `System.in` refers to standard input, which is the keyboard by default. `System.err` refers to the standard error stream, which is the console.

*System.in* is an object of type *InputStream*, *System.out* and *System.err* are object of type *PrintStream*. These are byte stream.

## 7.5 READING CONSOLE INPUT

Java also supports three Standard Streams:

- **Standard Input:** Accessed through *System.in* which is used to read input from the keyboard.
- **Standard Output:** Accessed through *System.out* which is used to write output to be display.
- **Standard Error:** Accessed through *System.err* which is used to write error output to be display.

### Working with Reader classes:

Java provides the standard I/O facilities for reading text from either the file or the keyboard on the command line. The *Reader* class is used for this purpose that is available in the *java.io* package. It acts as an abstract class for reading character streams. The only methods that a subclass must implement are *read(char[ ], int, int)* and *close()*. the *Reader* class is further categorized into the subclasses.

The following diagram shows a class-hierarchy of the *java.io.Reader*



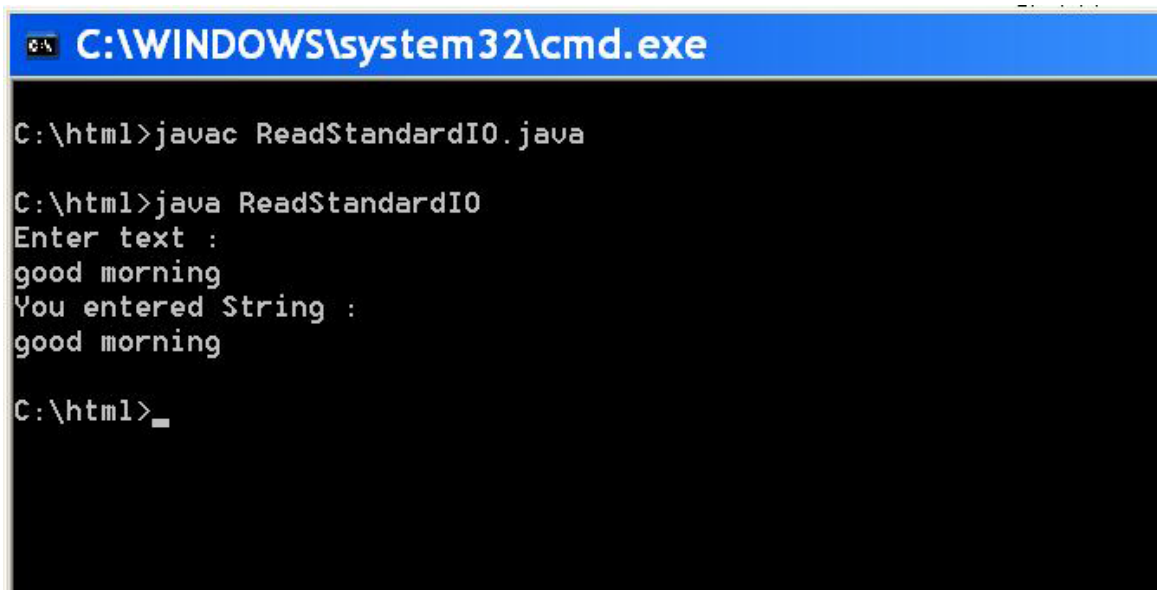
This program illustrates you how to use standard input stream to read the user input.

```
import java.io.*;

public class ReadStandardIO{

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

InputStreamReader inp=new
InputStreamReader(System.in);
BufferedReader br = new BufferedReader(inp);
System.out.println("Enter text : ");
String str = br.readLine();
System.out.println("You entered String : ");
System.out.println(str);
}
}
```

A screenshot of a Windows command prompt window. The title bar is blue and reads "C:\WINDOWS\system32\cmd.exe". The command prompt shows the following sequence of commands and output:  
C:\html>javac ReadStandardIO.java  
C:\html>java ReadStandardIO  
Enter text :  
good morning  
You entered String :  
good morning  
C:\html>\_

## 7.6 WRITING CONSOLE OUTPUT

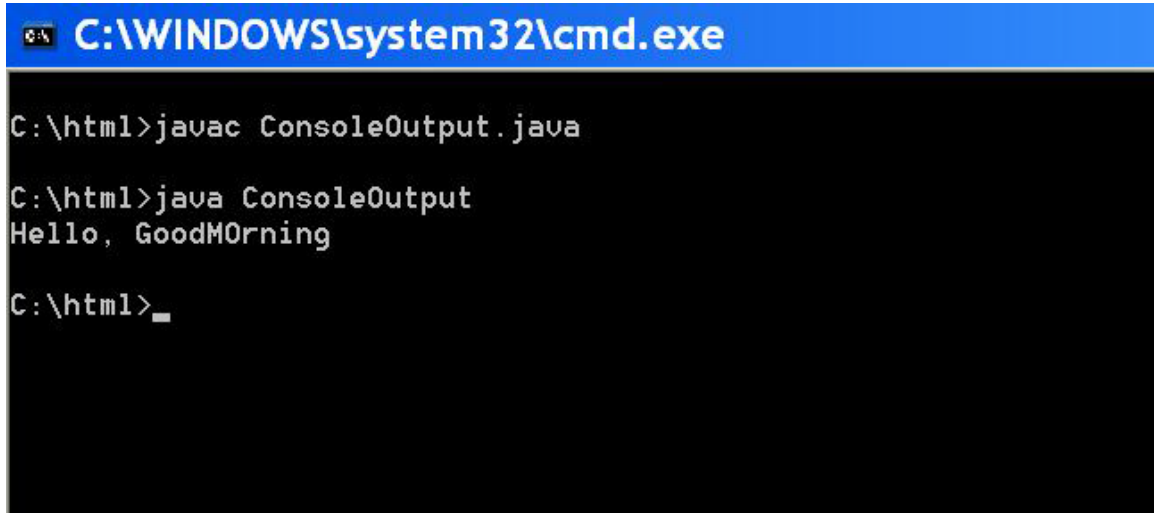
We can write programs that write text lines to the “console”, which is typically a DOS command window.

**ConsoleOutput.java:**

```
public class ConsoleOutput {

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

```
System.out.println("Hello, GoodMOrning");  
}  
}
```



```
C:\WINDOWS\system32\cmd.exe  
  
C:\html>javac ConsoleOutput.java  
  
C:\html>java ConsoleOutput  
Hello, GoodMOrning  
  
C:\html>_
```

No imports are required, The System class is automatically imported (as are all java.lang classes).

You can write one complete output line to the console by calling the System.out.println() method. The argument to this method will be printed. println comes from Pascal and is short for “print line”. There is also a similar print method which writes output to the console, but doesn’t start a new line after the output.

## 7.7 READING AND WRITING FILES

Java provides a number of classes and methods that allow us to read and write files. In Java all files are byte oriented, and Java provides methods to read and write bytes from and to a file.

The **File** class deals with the machine dependent files in a machine independent manner i.e., it is easier to write platform-independent code that examines and manipulates files using the File class. This class is available in the java.lang package.

The **java.io.File** is the central class that works with files and directories. The instance of this class represents the name of a file or directory on the host file system.

When a File object is created, the system does not check to the existence of a corresponding file/directory. If the file exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, reading from or writing to it.

Lets understand some I/O streams that are used to perform reading and writing operation in a file.

Java supports the following I/O file streams.

- FileInputStream
- FileOutputStream
- FileInputstream

This class is a subclass of Inputstream class that reads bytes from a specified file name . The read() method of this class reads a byte or array of bytes from the file. It returns -1 when the end-of-file has been reached. We typically use this class in conjunction with a BufferedInputStream and DataInputstream class to read binary data. To read text data, this class is used with an InputStreamReader and BufferedReader class. This class throws FileNotFoundException, if the specified file is not exist. We can use the constructor of this stream as:

```
FileInputstream(File filename);
```

### **FileOutputStream:**

This class is a subclass of OutputStream that writes data to a specified file name. The write() method of this class writes a byte or array of bytes to the file. We typically use this class in conjunction with a BufferedOutputStream and a DataOutputStream class to write binary data. To write text, we typically use it with a PrintWriter, BufferedWriter and an OutputStreamWriter class. You can use the constructor of this stream as:

```
FileOutputstream(File filename);
```

**DataInputStream:**

This class is a type of `FilterInputStream` that allows you to read binary data of Java primitive data types in a portable way. In other words, the `DataInputStream` class is used to read binary Java primitive data types in a machine-independent way. An application uses a `DataOutputStream` to write data that can later be read by a `DataInputStream`. You can use the constructor of this stream as:

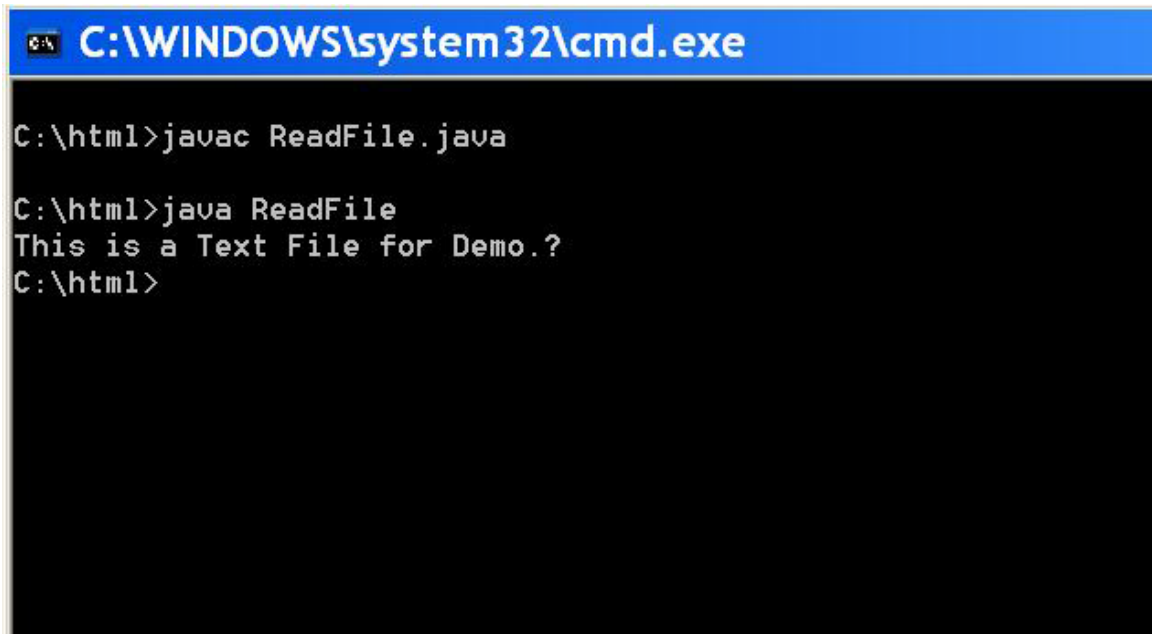
```
DataInputStream(FileOutputstream finp);
```

The following program demonstrate, how the contains are read from a file.

```
import java.io.*;

public class ReadFile{

    public static void main(String[] args) throws IOException{
        File f;
        f=new File("demo.txt");
        if(!f.exists() && f.length()<0)
            System.out.println("The specified file is not exist");
        else{
            FileInputStream finp=new FileInputStream(f);
            byte b;
            do{
                b=(byte)finp.read();
                System.out.print((char)b);
            }
            while(b!=-1);
            finp.close();
        }
    }
}
```



```
C:\WINDOWS\system32\cmd.exe

C:\html>javac ReadFile.java

C:\html>java ReadFile
This is a Text File for Demo.?
C:\html>
```

In the section, we will learn how to write data to a file. As we have discussed, the `FileOutputStream` class is used to write data to a file.

Let us consider an example that writes the data to a file converting into the bytes. This program first check the existence of the specified file. If the file exist, the data is written to the file through the object of the `FileOutputStream` class.

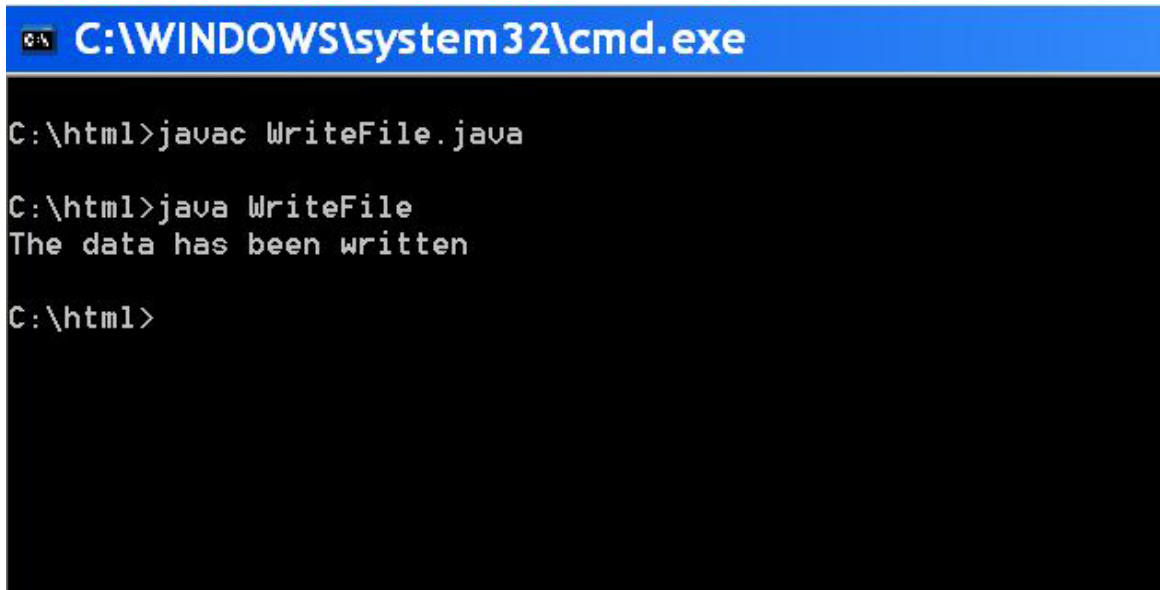
```
import java.io.*;

public class WriteFile{

    public static void main(String[] args) throws IOException{
        File f=new File("textfile1.txt");
        FileOutputStream fop=new FileOutputStream(f);
        if(f.exists()){
            String str="This data is written through the program";
            fop.write(str.getBytes());
            fop.flush();
            fop.close();
            System.out.println("The data has been written");
        }
        else
            System.out.println("This file is not exist");
    }
}
```



```
}  
}
```



```
C:\WINDOWS\system32\cmd.exe  
  
C:\html>javac WriteFile.java  
  
C:\html>java WriteFile  
The data has been written  
  
C:\html>
```

---

## 7.8 Check Your Progress

---

1. The Java Input/Output (I/O) is a part of ..... package.
2. ....is inherited from the Object class.
3. ....an International standard character encoding that is capable of representing most of the world's written languages.
4. System also contains three predefined system variables.....
5. The .....is the central class that works with files and directories.
6. ....Accessed through System.in which is used to read input from the keyboard
7. ....Accessed through System.out which is used to write output to be display.
8. ....Accessed through System.err which is used to write error output to be display.
9. ....is an object of type InputStream, System.out and System.err are object of type PrintStream.
10. Programs use .....streams to perform input and output of 8-bit bytes.

---

## 7.9 Summary

---

The Java Input/Output (I/O) is a part of java.io package. The java.io package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources.

The InputStream and OutputStream are central classes in the package which are used for reading from and writing to byte streams, respectively.

InputStream is inherited from the Object class. Each class of the InputStreams provided by the java.io package is intended for a different purpose. The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way, they differ mainly in the way they are constructed.

All Java programs automatically import java.lang package. This package defines a class called System, which encapsulates several aspects of the run-time environment.

Java supports the following I/O file streams.

- FileInputStream
- FileOutputStream
- FileInputstream

The File class deals with the machine dependent files in a machine independent manner i.e., it is easier to write platform-independent code that examines and manipulates files using the File class. This class is available in the java.lang package.

The `java.io.File` is the central class that works with files and directories. The instance of this class represents the name of a file or directory on the host file system.

---

## 7.10 Keywords

---

**InputStream**- The `InputStream` class is used for reading the data such as a byte and array of bytes from an input source.

**OutputStream**- The `OutputStream` class is a sibling to `InputStream` that is used for writing byte and array of bytes to an output source.

**Byte Stream** – for Binary I/O

**Character Stream** – for Character I/O

**Unicode**- Java represents strings in Unicode, an International standard character encoding that is capable of representing most of the world's written languages.

**InputStreamReader**- The `InputStreamReader` class implements a character-input stream that reads bytes from a byte-input stream and converts them to characters according to a specified encoding.

**OutputStreamWriter**- `OutputStreamWriter` class implements a character-output stream that converts characters into bytes according a specified encoding and writes them to a byte-output stream.

**System.out** - refers to the standard output stream.

**System.err** - refers to the standard error stream, which is the console.

**System.in** - refers to standard input.

**Console-** for writing programs that write text lines to the “console”, which is typically a DOS command window.

**File-** The File class deals with the machine dependent files in a machine independent manner.

---

## 7.11 Self-Assessment Test

---

- Q.1 What is stream in java? How Java represents a file?
- Q.2 How file and stream work in Java?
- Q.3 What are the two different types of streams? What are the advantages of character streams?
- Q.4 What is predefined stream?
- Q.5 Explain Console Output with proper program.
- Q.6 Discuss the hierarchy of reading console input.
- Q.7 How will you read and write file in Java? Explain.

---

## 7.12 Answers to check your progress

---

- 1. java.io
- 2. InputStream
- 3. Unicode
- 4. in, out and err
- 5. java.io.File
- 6. Standard Input
- 7. Standard Output
- 8. Standard Error
- 9. System.in
- 10. Byte

---

## **7.13 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.

SUBJECT: JAVA PROGRAMMING	
COURSE CODE: MCA-13	AUTHOR: AYUSH SHARMA
LESSON NO. 8	
GUI Programming	

## STRUCTURE

- 8.0 Learning Objective
- 8.1 Introduction
- 8.2 AWT Basics
- 8.3 AWT Components
- 8.4 Event Handling
- 8.5 Introduction to Swing
- 8.6 Swing Components
- 8.7 Event Handling
- 8.8 Display Text and Image in a Window
- 8.9 Layout Manager
- 8.10 Check Your Progress
- 8.11 Summary
- 8.12 Keywords
- 8.13 Self-Assessment Test
- 8.14 Answers to check your progress
- 8.15 References / Suggested Readings

## 8.0 LEARNING OBJECTIVE

After going through this unit, you will be able to:

- learn about the concept of AWT and its components
- describe event handling

- learn about Swing and its components
- display text and images in a window
- learn about layout manager

---

## 8.1 INTRODUCTION

---

In this unit we will learn about the **AWT** package of the Java and a brief description of **Swing**. The AWT is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the **Java Foundation Classes (JFC)** — the standard API for providing a graphical user interface (GUI) for a Java program. Here we will provide a brief description about different AWT components, event handling in AWT and Swing. At the end we will learn about Layout Manager.

---

## 8.2 AWT Basics

---

**AWT** stands for *Abstract Windowing Toolkit*. It contains all classes to write the program that interface between the user and different windowing toolkits. We can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.

Now a days developer are using Swing components instead of AWT to develop good GUI for windows applications.

---

## 8.3 AWT Components

---

In this section we will learn about the different components available in the Java AWT package for developing user interface for our program. Following are some of the components of Java AWT:

**Labels:** This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in our application and label never perform any type of action. Syntax for defining the label is:

```
Label label_name = new Label ("This is the label text");
```

Above code simply represents the text for the label.

```
Label label_name = new Label ("This is the label text.", Label.CENTER);
```

label can be left, right or centered. Above declaration used the center justification of the label using the *Label.CENTER*

**Buttons:** This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for our application. The syntax of defining the button is as follows:

```
Button button_name = new Button ("This is the label of the button.");
```

We can change the Button's label or get the label's text by using the `Button.setLabel(String)` and `Button.getLabel()` method. Buttons are added to its container using the `add (button_name)` method.

**Check Boxes:** This component of Java AWT allows us to create check boxes in our applications. The syntax of the definition of Checkbox is as follows:

```
Checkbox checkbox_name = new Checkbox ("Optional check box 1", false);
```

Above code constructs the unchecked Checkbox by passing the Boolean valued argument `false` with the Checkbox label through the `Checkbox()` constructor. Defined Checkbox is added to its container using `add (checkbox_name)` method. We can change and get the checkbox's label using the `setLabel (String)` and `getLabel()` method. We can also set and get the state of the checkbox using the `setState(boolean)` and `getState()` method provided by the `Checkbox` class.

**Radio Button:** This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows:

```
CheckboxGroup chkgp = new CheckboxGroup();
```

```
add (new Checkbox ("One", chkgp, false);
```

```
add (new Checkbox ("Two", chkgp, false);
```



**add (new Checkbox (“Three”,chkgp, false);**

In the above code we are making three check boxes with the label “One”, “Two” and “Three”. If we mention more than one true valued for checkboxes then our program takes the last true and show the last check box as checked.

**Text Area:** This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

**TextArea txtArea\_name = new TextArea();**

We can make the Text Area editable or not using the setEditable (Boolean) method. If we pass the Boolean valued argument false then the text area will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in the text area using the setText(string) method of the TextArea class.

**Text Field:** This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows:

**TextField txtfield = new TextField(20);**

We can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

As shown in the example below, a button is represented by a single label. That is the label shown in the example can be pushed with a click of a mouse.

#### **MyButton.java**

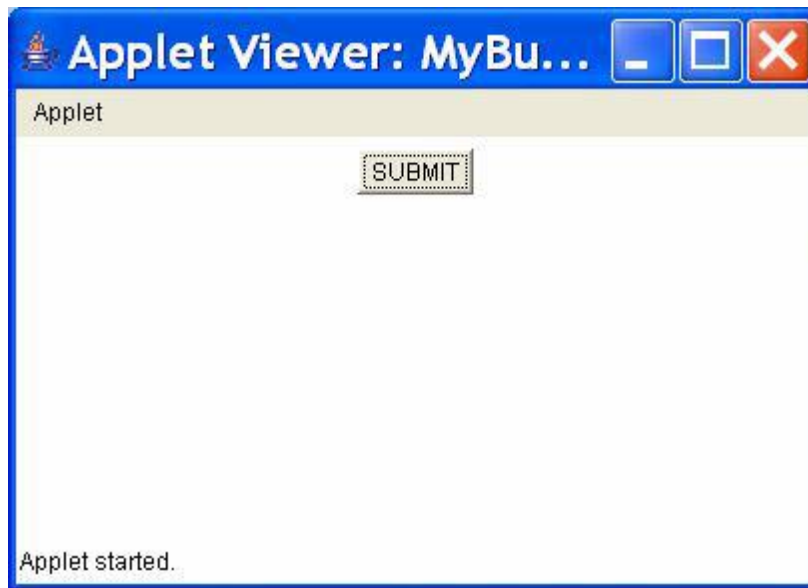
```
import java.awt.*;
import java.applet.Applet;
public class MyButton extends Applet
{
public void init()
{
Button button = new Button(“SUBMIT”);
add(button);
}
}
```

Here is the HTML code:

```
<HTML>  
<HEAD>  
</HEAD>  
<BODY>  
<APPLET ALIGN="CENTER" CODE="MyButton" WIDTH="400"  
HEIGHT="200"></APPLET>  
</BODY>  
</HTML>
```



```
C:\WINDOWS\system32\cmd.exe - appletviewer  
C:\html>javac MyButton.java  
C:\html>appletviewer mybutton.html
```



## 8.4 Event Handling

There are many types of events that are generated by our AWT Application. These events are used to make the application more effective and efficient. Generally, there are twelve types of event are used in Java AWT. These are as follows:

1. **ActionEvent** : It indicates the component-defined events occurred i.e. the event generated by the component like Button, Checkboxes etc.
2. **AdjustmentEvent** : This is the AdjustmentEvent class extends from the AWTEvent class. When the Adjustable Value is changed then the event is generated.
3. **ComponentEvent** : This is the low-level event which indicates, if the object moved, changed and its states (visibility of the object). This class only performs the notification about the state of the object.
4. **ContainerEvent** : This is a low-level event which is generated when container's contents changes because of addition or removal of a components.
5. **FocusEvent** : This indicates about the focus where the focus has gained or lost by the object

6. **InputEvent** : This event class handles all the component-level input events. This class acts as a root class for all component-level input events.

7. **ItemEvent** : The ItemEvent class handles all the indication about the selection of the object i.e., whether selected or not.

8. **KeyEvent** : It handles all the indication related to the key operation in the application if we press any key for any purposes of the object then the generated event gives the information about the pressed key. These types of events check whether the pressed key left key or right key, 'A' or 'a' etc.

9. **MouseEvent** : It handle all events generated during the mouse operation for the object. That contains the information whether mouse is clicked or not if clicked then checks the pressed key is left or right.

10. **PaintEvent** : The PaintEvent class only ensures that the paint() or update() are serialized along with the other events delivered from the event queue.

11. **TextEvent** : TextEvent is generated when the text of the object is changed.

12. **WindowEvent** : If the window or the frame of our application is changed (Opened, closed, activated, deactivated or any other events are generated), WindowEvent is generated.

---

## 8.5 Introduction to Swing

---

The Java *Swing* provides the multiple platform independent APIs interfaces for interacting between the users and GUIs components. Java provides an interactive feature for design the GUIs toolkit or components like: *labels, buttons, text boxes, checkboxes, combo boxes, panels* and *sliders* etc. All AWT flexible components can be handled by the Java Swing. The Java Swing supports the plugging between the look and feel features. The look and feel that means the dramatically changing in the component like JFrame, JWindow, JDialog etc. for viewing it into the several types of window.

---

## 8.6 Swing Components

---

There are many components which are used for the building of GUI in Swing. The Swing Toolkit consists of many components for the building of GUI. These components are also helpful in providing interactivity to Java applications. Following are the some of the components which are included in Swing toolkit:

1. list controls
2. buttons
3. labels
4. tree controls
5. table controls

All AWT flexible components can be handled by the Java Swing. Swing toolkit contains far more components than the simple component toolkit. In the next section we are going to show some examples.

**Text Field:** The following example shows how to create a text field. The swing text field is encapsulated by the JTextComponent class which extends JComponent. One of its subclass JTextField allows to edit one line of text box.

### JTextFields.java:

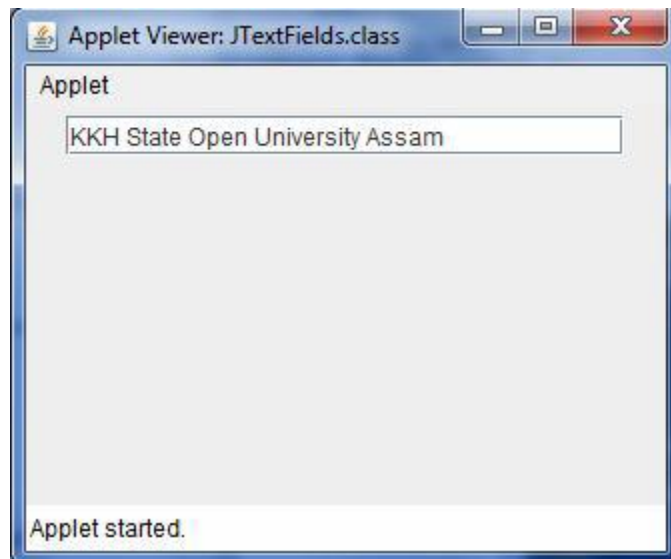
```
import java.awt.*;
import javax.swing.*;
public class JTextFields extends JApplet
{
    JTextField jtf;
    public void init()
    {
        //Get content pane
        Container contentPane=getContentPane();
        contentPane.setLayout(new FlowLayout());
        //Add Text field
        jtf= new JTextField(25);
```

```
//add Textbox to the content pane
contentPane.add(jtf);
}
}
```

### **JTextField.html**

```
<html>
<applet code="JTextFields.class" height=200 width=320>
</applet>
</html>
```

### **Output:**



### **Buttons:**

Swing buttons are subclasses of the AbstractButton class, which extends JComponent. The JButton class provides the functionality of a push button. The following example shows a push button.

### **JButtonDemo.java:**

```
import java.awt.*;
import javax.swing.*;
public class JButtonDemo extends JApplet{
```

```

public void init(){
//Get content pane
Container contentPane=getContentPane();
contentPane.setLayout(new FlowLayout());
//Add button to the content pane
JButton jb=new JButton("kkhsou");
contentPane.add(jb);
}
}

```

### **JButton.html:**

```

<html>
<applet code="JButtonDemo.class" height=200 width=320>
</applet>
</html>

```

### **Output:**



**Check Boxes:** The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. The following example shows how to create an applet that displays three check boxes.

### **JCheckBoxDemo.java**

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

public class JCheckBoxDemo extends JApplet
{
    public void init()
    {

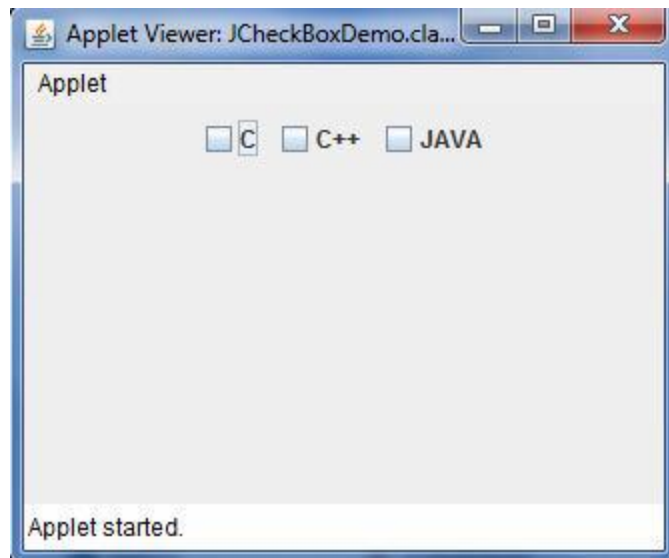
        //Get content pane
        Container contentPane=getContentPane();
        contentPane.setLayout(new FlowLayout());
        //Add checkbox to the content pane
        JCheckBox cb=new JCheckBox("C");
        contentPane.add(cb);
        cb=new JCheckBox("C++");
        contentPane.add(cb);
        cb=new JCheckBox("JAVA");
        contentPane.add(cb);
    }
}
```

### **JCheckBox.html:**

```
<html>
<applet code="JCheckBoxDemo.class" height=200 width=320>
</applet>
</html>
```



## Output:



## 8.7 Event Handling

Events are an important part in any GUI program. All GUI applications are event-driven. An application reacts to different event types which are generated during its life. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g., internet connection, window manager, timer. In the event model, there are three participants:

- event source
- event object
- event listener

The ***Event source*** is the object whose state changes. It generates Events. The ***Event object*** (Event) encapsulates the state changes in the event source. The ***Event listener*** is the object that wants to be notified. Event source object delegates the task of handling an event to the event listener. Event handling in Java Swing toolkit is very powerful and flexible. Java uses **Event Delegation Model**. We can specify the objects that are to be notified when a specific event occurs.

**Event object:** When something happens in the application, an event object is created. For example, when we click on the button or select an item from list. There are several types of events. An `ActionEvent`, `TextEvent`, `FocusEvent`, `ComponentEvent` etc. Each of them is created under specific conditions. Event object has information about an event, that has happened.

## 8.8 Display Text and Image in a Window

The following example shows how to create and display a label consisting of both text and image. The applet started by getting its content pane. Then an `ImageIcon` object created for the file `kkhsou_logo.jpg`. On the `JLabel` constructor first argument is text, second argument is `ImageIcon` object and the third argument is alignment. The align argument is `LEFT`, `RIGHT`, `CENTER`, `LEADING` or `TRAILING`. Finally the label is added to the content pane.

### **JLabelDemo.java**

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

public class JLabelDemo extends JApplet{
    public void init(){
        //Get content pane
        Container contentPane=getContentPane();
        //create icon
        ImageIcon ii=new ImageIcon("kkhsou_logo.jpg");
        //create label
        JLabel jl=new JLabel("KKH Open University", ii, JLabel.CENTER);
        //add label to the content pane
        contentPane.add(jl);
    }
}
```

## JLabelDemo.html

```
<html>  
<applet code="JLabelDemo.class" height=200 width=320>  
</applet>  
</html>
```

### Output



## **8.9 Layout Manager**

To create layouts, we use layout managers. Layout managers are one of the most difficult parts of modern GUI programming. We can use no layout manager, if we want. There might be situations, where we might not need a layout manager. But to create truly portable, complex applications, we need layout managers. Without layout manager, we position components using absolute values.

There are some of the common tasks associated to use layout managers:

- Setting Layout Manager
- Adding Components to a Container
- Providing Size and Alignment Hints
- Putting Space Between Components
- Setting the Container's Orientation

- Tips on Choosing a Layout Manager
- Third-Party Layout Managers

In Java a layout manager class implements the Layout Manager interface. It is used to determine the position and size of the components within a container. Components can provide size and alignment hints, still the container's layout manager has the final authority on the size and position of the components within the container.

**FlowLayout manager :** This is the simplest layout manager in the Java Swing toolkit. It is mainly used in combination with other layout managers. When calculating its children size, a flow layout lets each component assume its natural (preferred) size.

The manager puts components into a row. In the order, they were added. If they do not fit into one row, they go into the next one. The components can be added from the right to the left or vice versa. The manager allows aligning the components.

**GridLayout:** The GridLayout layout manager lays out components in a rectangular grid. The container is divided into equally sized rectangles. One component is placed in each rectangle.

**BorderLayout:** A BorderLayout manager is a very handy layout manager. It divides the space into five regions. *North, West, South, East* and *Centre*. Each region can have only one component. If we need to put more components into a region, we can simply put a panel there with a manager of our choice. The components in N, W, S, E regions get their preferred size. The component in the centre takes up the whole space left.

**BoxLayout:** BoxLayout is a powerful manager that can be used to create sophisticated layouts. This layout manager puts components into a row or into a column. It enables nesting, a powerful feature, which makes this manager very flexible. It means that we can put a box layout into another box layout.

---

## 8.10 Check Your Progress

---

1. ....is generally used to show the text or string in your application.

2. .... never perform any type of action.
3. .... component of Java AWT allows you to create check boxes in your applications.
4. .... is the special case of the Checkbox component of Java AWT package.
5. .... and .... is the text container component.
6. .... the event generated by the component like Button, Checkboxes etc.
7. .... is the low-level event which indicates, if the object moved, changed and it's states.
8. .... handle all events generated during the mouse operation for the object.
9. Events are an important part in any ..... program.
10. The ..... layout manager lays out components in a rectangular grid.
11. .... can be used to create sophisticated layouts.
12. A layout manager class implements the .....interface.

---

## 8.11 Summary

---

The AWT is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program.

AWT stands for Abstract Windowing Toolkit. It contains all classes to write the program that interface between the user and different windowing toolkits.

Some of the components of Java AWT: Labels, Buttons, Check Boxes, Radio Button, Text Area, Text Field.

There are twelve types of event are used in Java AWT. These are as follows: ActionEvent, AdjustmentEvent, ComponentEvent, ContainerEvent, FocusEvent, InputEvent, ItemEvent, KeyEvent, MouseEvent, PaintEvent, TextEvent, WindowEvent.

The Java Swing provides the multiple platform independent APIs interfaces for interacting between the users and GUIs components. Java provides an interactive feature for design

the GUIs toolkit or components like: labels, buttons, text boxes, checkboxes, combo boxes, panels and sliders etc.

Following are the some of the components which are included in Swing toolkit: list controls, buttons, labels, tree controls, table controls.

In the event model, there are three participants: event source, event object, event listener.

There are some of the common tasks associated to use layout managers:

- Setting Layout Manager
- Adding Components to a Container
- Providing Size and Alignment Hints
- Putting Space Between Components
- Setting the Container's Orientation
- Tips on Choosing a Layout Manager
- Third-Party Layout Managers.

## 8.12 Keywords

**AWT** - stands for Abstract Windowing Toolkit.

**Text Area** - This is the text container component of Java AWT package.

**ActionEvent** - It indicates the component-defined events occurred.

**AdjustmentEvent** - This is the AdjustmentEvent class extends from the AWTEvent class.

**ComponentEvent** - This is the low-level event which indicates, if the object moved.

**ContainerEvent** - This is a low-level event which is generated when container's contents changes because of addition or removal of a components.

**FocusEvent** - This indicates about the focus where the focus has gained or lost by the object.

**Swing** - The Java Swing provides the multiple platform independent APIs interfaces for interacting between the users and GUIs components.

**JTextComponent** - The swing text field is encapsulated by the JTextComponent class which extends JComponent.

**FlowLayout manager** - This is the simplest layout manager in the Java Swing toolkit.

**GridLayout** - The GridLayout layout manager lays out components in a rectangular grid.

---

### 8.13 Self-Assessment Test

---

- Q.1 What is AWT? List 5 AWT components.
- Q.2 Briefly discuss about three Layout manager.
- Q.3 What are the two different types of event used in Java AWT?
- Q.4 Describe some components of Java AWT.
- Q.5 What do you mean by event handling? What are the types of events used in Java AWT?

---

### 8.14 Answers to check your progress

---

- 1. Label
- 2. label
- 3. check boxes
- 4. radio button
- 5. text area, text field
- 6. ActionEvent
- 7. Component Event
- 8. MouseEvent
- 9. GUI
- 10. GridLayout
- 11. BoxLayout
- 12. LayoutManager

---

## **8.15 References / Suggested Readings**

---

1. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill.
2. Java- The Complete Reference by Herbert Schildt, ORACLE.
3. Java For Beginners by Scott Sanderson
4. Head First Java by Kathy Sierra & Bert Bates
5. Java: A Beginner's Guide, Eighth Edition by Herbert Schildt.
6. A Programmer's Guide to Java Certification, Mughal K. A., Rasmussen R. W., Addison – Wesley.