Yashwantrao
Chavan
Maharashtra
Open University

CMP512

JAVA

JAVA

Yashwantrao Chavan Maharashtra Open University
Dnyangangotri, Near Gangapur Dam
Nashik-422222

Yashwantrao Chavan Maharashtra Open University

Vice-Chancellor: Prof. E. Vayunandan

SCHOOL OF COMPUTER SCIENCE

Dr. Pramod Khandare Director School of Computer Science Y.C.M.Open University Nashik	**Shri. Madhav Palshikar** Associate Professor School of Computer Science Y.C.M.Open University Nashik	**Dr. P.V. Suresh** Director School of Computer and Information Sciences I.G.N.O.U. New Delhi
Dr. Pundlik Ghodke General Manager R&D, Force Motors Ltd. Pune.	**Dr. Sahebrao Bagal** Principal, Sapkal Engineering College Nashik	**Dr. Madhavi Dharankar** Associate Professor Department of Educational Technology S.N.D.T. Women's University, Mumbai
Dr. Urmila Shrawankar Associate Professor, Department of Computer Science and Engineering G.H. Raisoni College of Engineering Hingana Road, Nagpur	**Dr. Hemant Rajguru** Associate Professor, Academic Service Division Y.C.M.Open University Nashik	**Shri. Ram Thakar** Assistant Professor School Of Continuing Education Y.C.M.Open University Nashik
Mrs. Chetna Kamalskar Assistant Professor School of Science and Technology Y.C.M.Open University, Nashik	**Smt. Shubhangi Desle** Assistant Professor Student Service Division Y.C.M.Open University Nashik	

Writer/s	Editor	Co-ordinator	Director
Prof. Mr. Vipin Wani Assistant Professor, Department of Computer Science & Engineering, Sandip University, Nashik	Prof. Milind Bhandare Assistant Professor, Department of Computer Science, L.G.N. Sapkal COE Nashik	Ms. Monali R. Borade Academic Co-ordinator School of Computer Science, Y.C.M. Open University, Nashik	**Dr. Pramod Khandare** **Director** School of Computer Science, Y.C.M. Open University, Nashik

Production

INDEX

Chapter Number	Chapter Name		Page Number
1	**EVOLUTION OF JAVA, VARIABLES AND NAMING RULES**		1
	1.1	Basic Introduction	
	1.2	Brief History	
	1.3	Features of Java	
	1.4	Difference between C++ and Java	
	1.5	What are JDK, JVM and JRE?	
	1.6	Evolutions of Java	
	1.7	Introduction to Java Class and Objects	
	1.8	Instantiation in Java	
	1.9	Variables in Java	
	1.10	Java Data Types	
	1.11	Java Operators	
	1.12	Java Garbage Collection	
	1.13	What are java source file declaration rules?	
2	**DICISION MAKING AND LOOPS**		
	2.1	Java If-else Statement	
	2.2	Loops in Java	
	2.3	Java for Loop vs. While Loop vs. Do While Loop	
	2.4	Java Switch Statement	
	2.5	Java Break Statement	
	2.6	Java Continue Statement	
	2.7	Java Comments	
3	**IMPLEMENTATION OF METHODS**		
	3.1	**Introduction**	
	3.2	Methods in Java	
	3.3	**What is Constructor?**	
	3.4	Method Overloading	
	3.5	Constructor overloading	
	3.6	Method Overriding	
	3.7	Final Keyword in Java	
	3.8	Static keyword in java	
	3.9	This keyword in java	
	3.10	Inheritance in Java	
	3.11	Super Keyword in Java	

4	**WRAPPER CLASSES, ARRAY AND STRING**		
	4.1	Wrapper Classes in Java	
	4.2	Java.lang.Number Class in Java	
	4.3	Java Integer Class	
	4.4	Java FloatClass	
	4.5	Java.Lang.Byte class in Java	
	4.6	Java.Lang.Short class in Java	
	4.7	Java.Lang.Long class in Java	
	4.9	Java.lang.Character Class in Java	
	4.10	**Type conversion / Type Casting**	
	4.11	**Narrowing or Explicit Conversion**	
	4.12	**Autoboxing & Unboxing**	
	4.13	Arrays in Java	
5	**STRING AND EXCEPTIONS HANDLING**		
	5.1	Java String	
	5.2	StringBuffer and StringBuilder class	
	5.3	Immutable String in Java	
	5.4	Exception Handling	
	5.5	Terminology Related to Exceptions	
	5.6	Important Points to remember	
6	**PACKAGE AND DEFERRED IMPLIMENTATIONS**		
	6.1	What is Package in Java?	
	6.2	How packages work?	
	6.3	Types of packages	
	6.4	Import a Package	
	6.5	**Access Protection in Java Packages**	
	6.6	Abstract Classes and Interfaces	
	6.7	Interfaces in Java	
	6.8	Generalization, Specialization, and Inheritance	
7	**JAVA INPUT-OUTPUT**		
	7.1	Introduction	
	7.2	File Class in Java	
	7.3	Reading and Writing Files	
	7.4	Stream	
	7.5	Java BufferedWriter Class	
	7. 6	Java BufferedReader Class	

	7.7	Reading data from console using InputStreamReaderand BufferedReader	
	7.8	Serialization and Deserialization in Java	
	7.9	ObjectOutputStream class	
	7.10	ObjectInputStream class	
	7.11	Example of Java Serialization	
	7.12	Example of Java Deserialization	
	7.13	Java Scanner	
8	**THREAD, GENERICS AND COLLECTIONS**		
	8.1	Introduction to Threading	
	8.2	Multithreading in Java	
	8.3	Lifecycle and States of a Thread in Java	
	8.4	Constructors in Thread Class	
	8.5	Methods in Thread Class	
	8.6	Java Thread Priority in Multithreading	
	8.7	Synchronization in Java	
	8.8	Java Generics (Generic Methods and Classes)	
	8.9	Generic Methods	
	8.10	Generic Classes	
	8.11	Collections in Java	

CHAPTER 1

EVOLUTION OF JAVA, VARIABLES AND NAMING RULES

Objectives :

- To study java introduction, features of java , learn JRE JDK JVM.
- To study history of java, difference between c++ and java, evolution of java.
- To study java variables, java identifiers, data types, java operators.
- To learn the working of garbage collection in java.

1.1 BASIC INTRODUCTION

Java is a High level object oriented programming language which is designed at Sun Microsystems (Sun) in 1991by James Gosling. The intention behind java language was to develop languages which allow writing a program once and executing multiple times. Java allows to write a program once and then run this program on multiple operating systems.

1.2 BRIEF HISTORY

Java Programming Language was written by James Gosling along with two other people 'Mike Sheridan' and 'Patrick Naughton', while they were working at Sun Microsystems (which has since been acquired by Oracle Corporation). Initially it was named oak Programming Language. Oak was a tree that stood outside Gosling's office that was the inspiration Gosling for writing java language so they released first version of java in 1991. later on they came to know that there is another language is already registered with name Oak so James Gosling called a meeting to discuss a new name for their language, coincidentally while discussion they were having a coffee and the brand name of that coffee was java so here they get the idea why not to rename it to java and they rename it to java in 1995, and that's why java having symbol of coffee. The term Java does not have any full form, It is a programming language originally developed by James Gosling at Sun Microsystems in 1995. Java derives much of its syntax from the: C and C++ which are known as the most popular programming languages of all time. Today Java of Sun Microsystems is a subsidiary of Oracle Corporation.

1.3 FEATURES OF JAVA

Java Comes with various distinct features as compared to other object oriented programming languages.

1.3.1.Object oriented:

Java is an Object Oriented Programming Language, which supports the construction of programs that consist of collections of collaborating objects. These objects have a unique identity and have oop features such as encapsulation, abstraction, Class, inheritance and polymorphism.

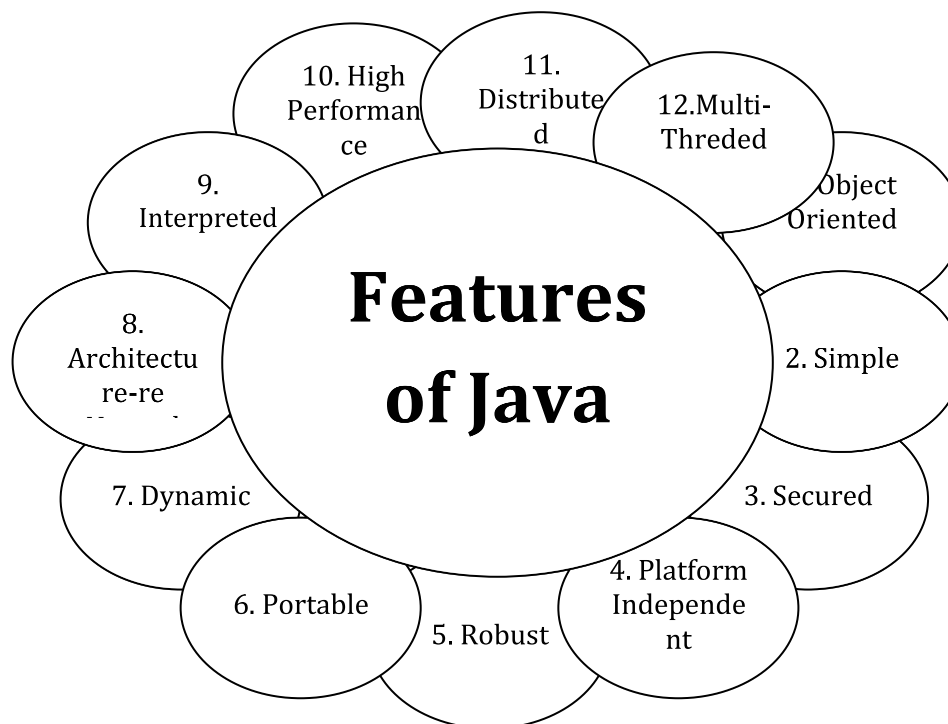


Fig. 1.1 : Features of Java

1.3.2. Simple:

Java was designed with a small number of language constructs so that programmers could learn it quickly which make it simple. It eliminates several language features that were in C/C++ that was associated with poor programming practices or rarely used such as multiple inheritance, goto statements, header files, structures, operator overloading, and pointers, security reason was also there for removing or not adding pointer in java.

1.3.3. Secure:

Java is designed to be more secure in a networked environment. A byte code verification processis used by the Java run-time environment to ensure that code loaded over the network does not violate Java security constraints. Absence of pointer also adds on values for java security features.

1.3.4. Platform Independent:

As we have seen on Introduction page Byte code and JVM which plays the major role for making the java platform independent and which make it a distinct.

1.3.5. Robust:

Java is designed to eliminate certain types of programming errors. Java is strongly typed, which allows extensive compile-time error checking. Its automatic memory management (garbage collection) eliminates memory leaks and other problems associated with dynamic memory allocation and deallocation. Java does not support memory pointers, which eliminates the possibility of overwriting memory and corrupting data.

1.3.6. Portable:

Has usually meant some work when moving an application program to another machine. Recently, the Java programming language and runtime environment has made it possible to have programs that run on any operating system and machine that supports the **Java** standard (from Sun Microsystems) without any porting work.

1.3.7. Architecture Neutral:

Any system that implements the Java Virtual Machine interprete the Java applications that are compiled to bytecodes. The Java Virtual Machine is supported by most of operating systems, this means that Java applications are able to run on most platforms.

1.3.8. Dynamic:

Java supports dynamic loading of classes class loader is responsible for loading the class dynamically in to JVM.

1.3.9. Interpreted:

Java source code is compiled to bytecodes, which are interpreted by a Java run-time environment.

1.3.10. High Performance:

Java is an interpreted language, it was designed to support "just-in-time" compilers, which dynamically compile bytecodes to machine code.

1.3.11. Multithreaded:

Java supports multiple threads of execution including a set of synchronization primitives. This makes programming with threads much easier.

1.3.12. Distributed:

Java supports various levels of network connectivity. Java applications are network conscious: TCP/IP support is built into Java class libraries. They have ability to open and access remote objects on the Internet.

1.4 Difference between Working of C++ and Java

1.4.1 C++ Vs. Java Program Execution Steps

C++ language is a platform dependent so The C++ compiler is designed to produce platform-specific, optimized code which means for different platform you will required a different compiler. Whereas java is a platform independent and Java supports portability. That is the main feature of java. Following figure shows that the compiler of C++ is strictly dependent on java while java compiler produces a byte code that can be run on any platform.

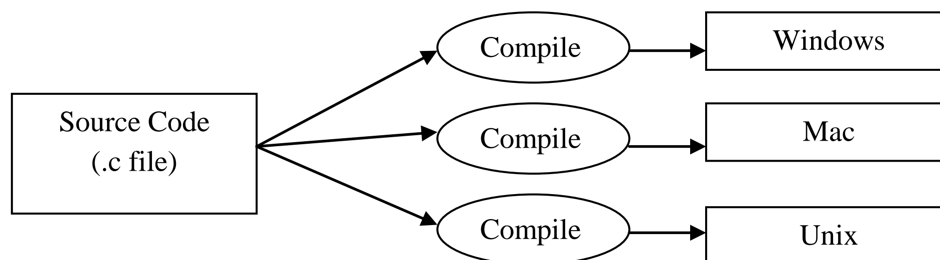


Fig.1.2 :C++ Program Execution steps

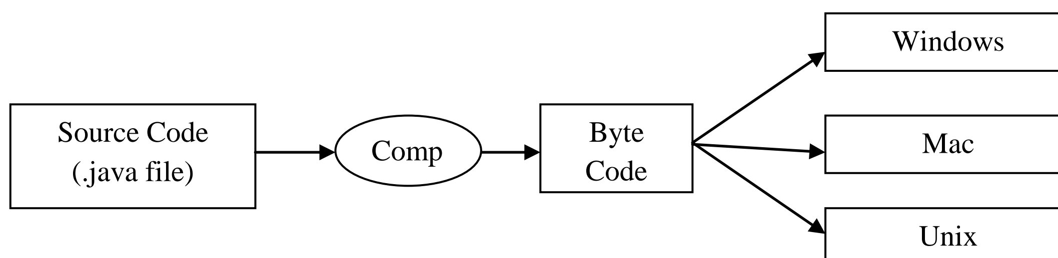


Fig. 1.3 : Java Program Execution

1.4.2 Java Environment Architecture

Java adopt both the approaches of compilation and interpretation. After compiling source file that is .java file compiler produces a .Class file which is nothing but a collection of byte code, at the run time, Java Virtual Machine (JVM) interprets this bytecode to generate the machine code which will be directly executed by the machine in which java program runs. So java is both compiled and interpreted language.

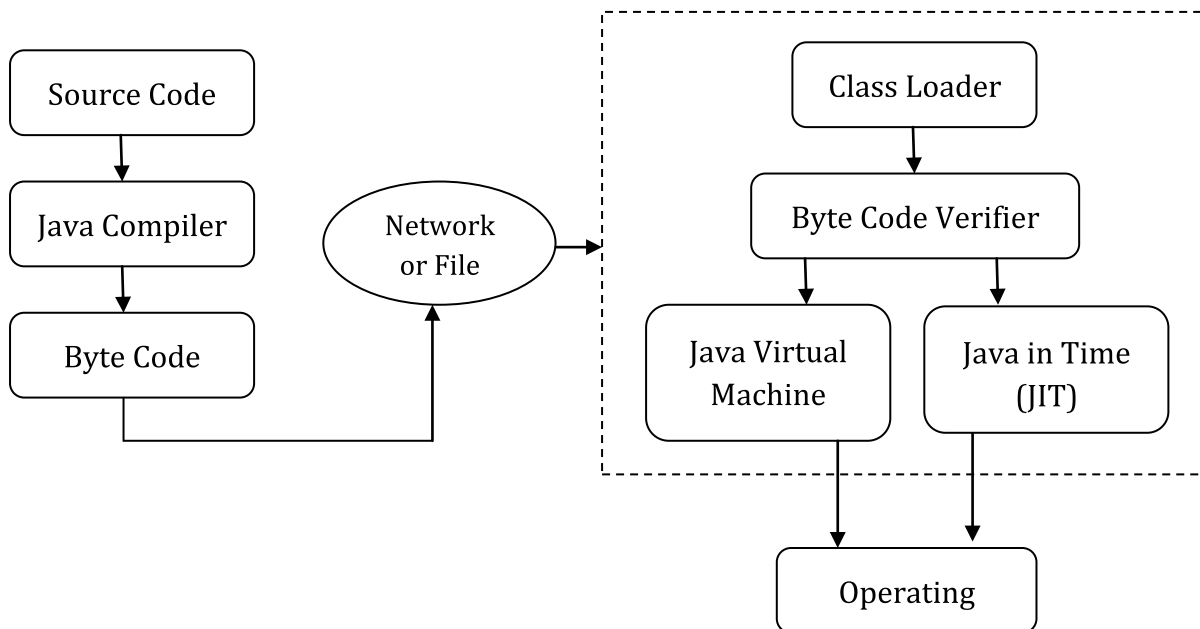


Fig. 1.4 : Java EnvironmentArchitecture

1.4.2.1. Bytecode: As discussed above, javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. The compiler saves the the bytecode in a .class file.

1.4.2.2. Class Loader: The Java Classloader is a part of the Java Run-time Environment that is responsible for dynamically loading of Java classes into the Java Virtual Machine. Just because of the class loaders the Java run time system does not required to know about files and file systems.

1.4.2.3. Byte Code Verifier: Is again a part of Java Run-time Environment, after loading the bytecode in JVM bytecode are first inspected by a verifier. The byte code verifier also checks that theobviously damaging instructions cannot perform actions.

1.4.2.4. Java Virtual Machine (JVM): This is generally referred as JVM. Let us discuss phases of program execution first, then will discuss about JVM. The various Phases of program

executions are as follows: we write the program using any suitable editor, then we compile the program and at last we run the program.

1) Writing of the program is done by java programmer like you and me using suitable editors.
2) Then Compilation of program is done with the help of javac compiler, javac is the primary java compiler included in java development kit (JDK). Java Compiler takes the java program (.java file) as input and generates java bytecode also called as class file (.class) as output.
3) At last, JVM executes the bytecode (.class file) generated by compiler. This is called program run phase.

1.4.2.5. The Just-In-Time (JIT) compiler: It is the one of the most important component of the java runtime environment which improves the performance of Java applications by compiling bytecodes at run time to native machine code. Java programs consist of classes, which contain platform-independent bytecodes that can be interpreted by a JVM. The JVM loads the class files at run time, and determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation indicates that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

1.5 What are JDK, JRE and JVM?

1.5.1 Java Development Kit (JDK):

JDK is complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc. To create, compile and run Java program you requires JDK installed on your computer. The Java Development Kit (JDK) is a software development environment. It is used for developing Java applications and applets. JDK includes various other important componants such as Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

1.5.2 Java Runtime Environment (JRE):

The Java Runtime Environment (JRE) provides the various componants to run applets and applications. Componants such as libraries, the Java Virtual Machine and many more other components for applications written in the Java programming language. In addition to componants it also suports, two key deployment technologies are part of the JRE: Java Plug-in, which enables applets to run in popular browsers; and Java Web Start, which deploys standalone

applications over a network. It is also the foundation for the technologies in the Java 2 Platform, Enterprise Edition (J2EE) for enterprise software development and deployment. IT does not contain any tools and utilities such as compilers or debuggers required for developing applets and applications. JRE includes JVM, browser plugins and applets support. You required only JRE in case you only need to run a java program on your computer.

1.5.2.1 What does JRE consists of?

JRE consists of the following components:

- **Deployment technologies**, including services and tools like deployment, Java Web Start and Java Plug-in.
- **User interface toolkits**, It including services like Print Service, Sound,Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, drag and drop (DnD) and input methods.
- **Integration libraries**, includingJava Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Interface Definition Language (IDL), Java Database Connectivity (JDBC), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
- **Other base libraries**, which includingJava Native Interface (JNI), Math, Networking, international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
- **Lang and util base libraries**, including services like Java Archive (JAR), Logging, Preferences API,lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects and Regular Expressions.
- **Java Virtual Machine (JVM)**, including services like Java Hotspot Client and Server Virtual Machines.

1.5.3 Java Virtual Machine (JVM):

The **Java Virtual Machine** (JVM) is the virtual machine that executes the Java bytecodes. The JVM doesn't understand Java source code, hence you need to compile *.java files first to obtain *.class files that contain the byte codes understood by the JVM. It's also the reason that makes Java to be a "portable language" (*write once, run anywhere*). Indeed, there are specific implementations of the JVM for different systems. The aim is that with the same byte codes they all give the same results.is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

1.6 Evolutions of Java

The development of each programming language is based on a fact: there is a need to solve a problem that was not resolved by previous programming languages. Early programmers had to choose different programming languages, usually for various tasks, such as a specific language for a type of field. A certain language was sufficiently abel to solve the problems of its field but was not capable to solve the problems of other fields. Let us consider the example of Fortran language, Itcould have been used to write efficient programs for scientific problems, but it was not good for system code. Similarly, *Basic* problems and algorithms was easy to understand but was not robust to write big programs; While the assembly language was powerful for writing efficient programs, but it was not easy to remember and execution.

Many Programming languages such as Cobol, Fortran any many more do not have structural principles. Such languages use the Goto statement to control the flow of the program. Hence, writing the programs using such type of code and statements are made up of many jumps and conditional statements that make it challenging to understand.

Hence, *C* was invented in 1970, to replace the assembly language and to create a structured, effective and high-level language. The Scientist Dennis Ritchie started the development of C which was the result of the development process with *BCPL*, which is an old language developed by Martin Richard. A language called *B*, which was influenced by BCPL was developed by Ken Thompson.

C is a middle level and processor-oriented programming language; which is easy to execute and understand. C got quite famous at that time because of its features like reliable, simple and easy to use.

Though C was a efficient and successful programming language, but the complexity of the program was seeking more efficient language to solve problems. When we write a program in C, it has a limit, such as a maximum of 25000 lines of code, beyond which it cannot handle the complexity. But writing and managing large programs was a need at that time. So a new concept came.

C++ came with object-oriented programming features. The language C++ is the extension of C language which has been used widely. It is a powerful and modern language which has the power and simplicity of C and the characteristics of OOP. C++ provides more functional software benefits than C.

C ++ with OOP became quite famous but then a new problem arised,which is to control the software on different machines, a separate compiler is required for that CPU. But building a C++ compiler was quite expensive. Therefore, an efficient and easy and fixed solution was needed, and

this requirement became the motivation for the creation of Java, which is a portable and platform-independent language.

1.6.1 History of various Java versions:

Table 1.1 : History of Java Versions

VERSION	RELEASE DATE	MAJOR CHANGES
JDK Beta	1995	
JDK 1.0	January 1996	The Very first version was released on January 23, 1996. The principal stable variant, JDK 1.0.2, is called Java 1.
JDK 1.1	February 1997	Was released on February 19, 1997. There were many additions in JDK 1.1 as compared to version 1.0 such as • A broad retooling of the AWT occasion show • Inner classes added to the language • JavaBeans • JDBC • RMI
J2SE 1.2	December 1998	"Play area" It was the codename which was given to this form and was released on 8th December 1998. Its real expansion included: strictfp keyword • the Swing which is a graphical API was coordinated into the centre classes • Sun's JVM was outfitted with a JIT compiler out of the blue • Java module • Java IDL, an IDL usage for CORBA interoperability • Collections system
J2SE 1.3	May 2000	Codename-"KESTREL "Release Date- 8th May 2000 Additions: • HotSpot JVM included • Java Naming and Directory Interface • JPDA • JavaSound • Synthetic proxy classes
J2SE 1.4	February 2002	Codename-"Merlin "Release Date- 6th February 2002 Additions: Library improvements • Regular expressions modelled after Perl regular expressions • The image I/O API which is used for reading and writing images in various formats like JPEG and PNG • Integrated XML parser and XSLT processor (JAXP) (specified in JSR 5 and JSR 63) • Preferences API (java.util.prefs) Public Support and security updates for this version ended in October 2008.

J2SE 5.0	September 2004	Codename-“Tiger “Release Date- “30th September 2004” Originally numbered as 1.5 which is still used as its internal version. Added several new language features such as: • for-each loop • Generics • Autoboxing • Var-args
JAVA SE 6	December 2006	Codename-“Mustang “Released Date- 11th December 2006 Packaged with a database supervisor and encourages the utilization of scripting languages with the JVM. Replaced the name J2SE with ava SE and dropped the .0 from the version number. Additions: • Upgrade of JAXB to version 2.0: Including integration of a StAX parser. • Support for pluggable annotations (JSR 269). • JDBC 4.0 support (JSR 221)
JAVA SE 7	July 2011	Codename-“Dolphin “Release Date- 7th July 2011 Added small language changes which include strings in the switch. The JVM was extended with support for dynamic languages. Additions: • Compressed 64-bit pointers. • Binary Integer Literals. • Upstream updates to XML and Unicode.
JAVA SE 8	March 2014	Released Date- 18th March 2014 Language level support for lambda expressions and default methods and a new date and time API inspired by Joda Time.
JAVA SE 9	September 2017	Release Date: 21st September 2017 Project Jigsaw: designing and implementing a standard, a module system for the Java SE platform, and to apply that system to the platform itself and the JDK.
JAVA SE 10	March 2018	ReleasedDate-20thMarch Addition: • Additional Unicode language-tag extensions • Root certificates • Thread-local handshakes • Heap allocation on alternative memory devices • Remove the native-header generation tool – javah. • Consolidate the JDK forest into a single repository.

JAVA SE 11	September 2018	ReleaseDate-25thSeptember,2018 Additions- • Dynamic class-file constants • Epsilon: a no-op garbage collector • The local-variable syntax for lambda parameters • Low-overhead heap profiling • HTTP client (standard) • Transport Layer Security (TLS) 1.3 • Flight recorder
JAVA SE 12	March 2019	ReleasedDate-19thMarch2019 Additions- • Shenandoah: A Low-Pause-Time Garbage Collector (Experimental) • Microbenchmark Suite • Switch Expressions (Preview) • JVM Constants API • One AArch64 Port, Not Two • Default CDS Archives

1.7 Introduction to Java Class and Objects

1.7.1 Java Class:

Class is the Entity which binds the all data members and data functions together.It can also be defined as class is the basic building block of an object-oriented language which is a template that describes the data and behaviour associated with instances of that class. When you instantiate a class you create an object,This Object looks and feels like other instances of the same class.

Syntax to Write a Class:

```
Keyword class ClassName
{
\\ Variable Declarations
main function()
{
\\Your Logic comes here
}
}
```

1.7.1.1 Saving Java File: Save a java File as ClassName.java

Is it Compulsory to save Java File as same name with Class Name? Answer is NO, When we save file and when we compile it then the compiler generates .class file with the same name as class name, and remember always for compilation we use .java file and for execution we use .class file. So if you want to save your program with different name than your class name, then you just have to run your .class file which is generated after compilation.

Example 1:

```
ClassHelloWorld
{
int i;   // Variable Declaration
public static void main(String [] args)   //Main Function Declaration
{
int j;
System.out.println(" Hello World Welcome to Java");
}
}
Output: Hello World Welcome to Java
```

1.7.1.2 What is Static in Public static void main?

In Example 1: HelloWorld is a class name then we have declared a global variable i and local variable j, public static void main is a function to which we have pass array of argument of type string. In main function public is an access specifier void is a return type and main is a function, while static member is member of a class that isn't associated with an instance of a class. Instead, the member belongs to the class itself. A method with static keyword is known as static method. A static method belongs to the class rather than object of a class. A static methodinvoked without the need for creating an instance of a class.static methods are allowed to assess thestatic data member and can also change the value of it.

1.7.1.3. What is System.out.println?

In this Sentence System is a class out is a field and println is inbuilt function about which we will discuss in upcoming sections.

Example 2:

```
class Demo
{
public static void main(String [] args)   //Main Function Declaration
{
int=10;
System.out.println(”J==>”+j);
}
}
Output: J==>10
```

1.7.2. Java Object:

It is a basic unit of **Object** Oriented Programming and Object represent any real life entity that can be considered for implementation. **Objects** have states and behaviours. Example: A cat has states - colour, name, breed as well as behaviours – wagging the tail, meowing, eating. An **object** is an instance of a class. Class − A class is a template/blueprint that describes the behaviour/state that the **object** of its type support.

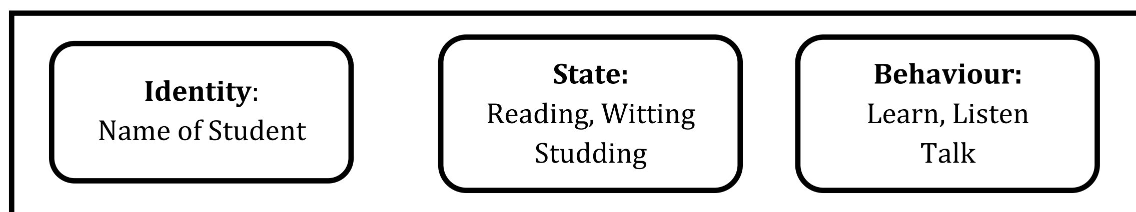
1.7.2.1 An object consists of:

1. **State:** It is denoted by attributes of an object. It also reflects the properties of an object.

2. **Behaviour:** It is described by methods of an object. It also imitates the response of an object with other objects.

3. **Identity:** It describe a unique name to an object and permits one object to interact with other objects.

Example of an object: Student



The class is said to be **instantiated,** when an object of a class is created. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have one or any number of instances.

```
Student S1 =new Student()

Student S2 =new Student()

Student S3 =new Student()

Student S3 =new Student()
```

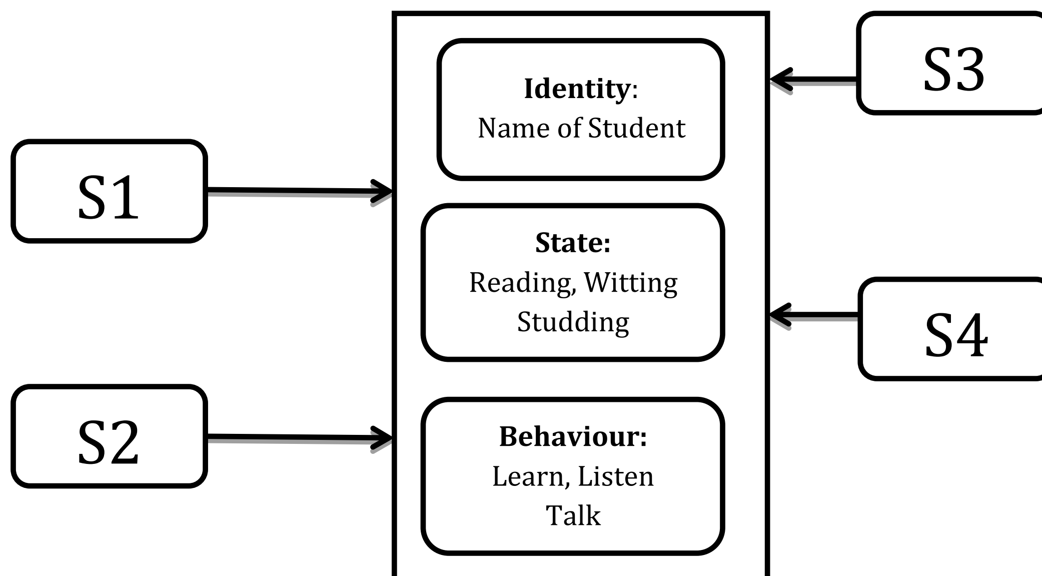


Fig. 1.5 : Example of object student

1.8 Instantiation in Java

To implement and to run the program in java one need to write a Source Code using java Syntaxes, then need to save the program with ".Java" extension. This java file then will be compiled to produce ".class" file which will be executed to get the required output. To Compile the program; we need run the java compiler javac, with the name of the source file on the command line as shown below – "javac myJavaProgram.java" If your program Is error free then javac compiler creates a file called "myJavaProgram.class" containing the bytecode of the program, which then can be executed to get the required output.

Instantiation

In Java, **instantiation** mean to call the constructor of a class that creates an **instance** or **object** of the type of that class. In other words, creating an object of the

class is called **instantiation**. It occupies the initial memory for the object and returns a reference. An object instantiation in Java provides the blueprint for the class.
What is an object?

- It is a runtime entity.
- It contains the blueprint of the class.
- We can create any number of objects of a class.
- It may represent user-defined data like Vector, Lists, etc.

Syntax for Instantiation

```
1. ClassName objName = new ClassName();
```

Or

```
1. ClassName cn;
2. cn= new ClassName;
```

Let's understand the above statements through an example.

Creating Instances

There are two ways to create instances:

- Using the **new** Keyword
- Using **Static Factory Method**

Using the new Keyword

Java provides the **new** keyword to instantiate a class.

Defining a Reference

```
1. //defines a reference (variable) that can hold an object of the DemoClass
2. DemoClass dc;
```

Instantiation

```
1. DemoClass dc = new DemoClass();  //instantiation
```

We can also instantiate the above class as follows if we defining a reference variable.

```
1. //creates a DemoClass object (instantiate)
2. //new keyword allocates memory space for the newly created object
3. dc = new DemoClass();
```

We observe that when we use the **new** keyword followed by the class name, it creates an instance or object of that class. **Creating a constructor** of the class is also known as **instantiation**.

1.9 Variables in Java

1.9.1 Variable

Variable is name allotted to reserved area in memory. In simple words, it is a name given to memory location. It is a combination of "vary + able" that means its value can be changed.

1.9.1.1 Types of Variables: There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable: A variable declared in some local scope such as inside the body of the method is called local variable. Such variablesare assesible only within that method and the other methods in the class aren't even aware that the variable exists.
A local variable cannot be defined with "static" keyword.

2) Instance Variable: These variables are not static and are declared inside the class but outside the body of the method, are called instance variable.
Such variables are called instance variable because their valuesare instance specific and is not shared among other instances.

3) Static variable: A variable which is declared with keyword static is called as static variable. It cannot be local. Single copy of static variable can be created to share among all the instances of the class. Memory allocation for static variable happens when the class is loaded in the memory and it happens only once. Variables are containers for storing data values.

```
Syntax:
type variable = value;
```

Where *type* is one of Java's types (such as `int` or `String`), and *variable* is the name of the variable (such as **x** or **name**). The **equal symbol** is used to assign some values written in right side to the variable.

To create a variable to store some text, let us consider the following example:

```
String name ="Rama";

System.out.println(name);
```

1.9.2. Java Identifiers

All Java variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be any small names (like a and b) or more expressive names (age, sum, totalVolume).

The general rules for writting names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names cannot contain whitespace and itshould start with a lowercase letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are strongly case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names

1.9.3 Scope in java

1.9.3.1 Scope of Variables in Java: Scope of a variable is the area or region in the program where the variable is accessible. Like C/C++, in Java, Scope of a variable can determined at compile time and independent of function call stack.
Java programs are structured in the form of classes. Every class is part of some package. Java scope rules can be learned under following categories.
1.9.3.2 Class Scope: Each variable declared inside of a class's brackets ({*}*) with *private* access modifier but outside of any method, has class scope. As a result, **these variables can be used everywhere in the class, but not outside of it**:

```
PublicclassClassScopeDemo

{

   PrivateInteger balance = 0;

   public voiddemoMethod()
```

```
            {
             balance++;
            }
             PublicvoidanotherDemoMethod()
            {
             Integer anotherBalance = balance+ 4;
            }
}
```

We can see that *ClassScopeDemo* has a class variable balancethat can be accessed within the class's methods.

1.9.3.3 Method Scope: When a variable is declared inside a method, it has method scope and **it will only be valid inside the same method:**

```
PublicclassMethodScopeDemo
{
   PublicvoidmethodAbc() {
   Integer age = 2;
 }
   PublicvoidmethodXyz()
{
      // It will give error: compiler error, age cannot be resolved to a variable
      age= age+ 2;
   }
}
```

In methodAbc, we created a method variable called *age*. Hence we can use *age* inside methodAbc, but we can't use it anywhere else.

1.9.3.4. Loop Scope: If we declare a variable inside a loop, it will have a loop scope and **will only be available inside the loop:**

```
PublicclassLoopScopeDemo
{
```

```
   List<String>listOfNames = Arrays.asList("Ram", "Sham", "Lakhan");

   PublicvoiditerationOfNames()

{

      String allNames = "";

      for(String name : listOfNames)

          {

         allNames = allNames + " "+ name;

      }

      // It will give error: compiler error, name cannot be resolved to a variable


      String lastNameUsed = name;

   }

}
```

In above example method *iterationOfNames* has a local variable called *name*. This variable can be used only inside the method *iterationOfNames* and is not valid outside of it.

1.9.3.5. Bracket Scope: We can define additional scopes anywhere using brackets (*{}*):

```
PublicclassBracketScopeDemo

{

   PublicvoidmathOperationExample() {

      Integer sum = 0;

      {

         Integer number = 2;

         sum = sum + number;

      }

// It will give error: compiler error, number cannot be resolved to a variable

      number++;

   }

}
```

The variable *number* is only accessible within the brackets.

1.10. Java Data Types

As explained in the previous chapter, a variable in Java must be a specified data type:

1.10.1 Primitive Data Types

A primitive data type describe the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Table 1.2 : Primitive Data Types

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

1.10.2 Non-Primitive Data Types

Non-primitive data types are also called as **reference types**as they refer to objects.

The important difference between **primitive** and **non-primitive** data types are:

- Primitive types are inbuilt or predefined (already defined) in Java. Where as Non-primitive data types are created by the programmer as per programming requirement and are not defined by Java (except for String).

1.11. Java Operators

Operators are used to perform operations on variables and values.

In the example below, we use the +**operator** to add together two values:

Example int **a =10+ 20;**

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

1.11.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations.

Table 1.3 : Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value from another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

1.11.2 Java Assignment Operators

Assignment operators are used to assign values to variables.

Consider the example given below, we have used the **assignment** operator (=) to assign the value **10** to a variable called **a**:

Example **inta=10;**

The addition assignment operator (+=) adds a value to a variable:

A list of all assignment operators:

Table 1.4 : Assignment Operators

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

1.11.3 Java Comparison Operators

Comparison operators are used to compare two values:

Table 1.5 : Comparision Operators

Operator	**Name**	**Example**
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

1.11.4 Java Logical Operators

Logical operators are used to determine the logic between variables or values:

Table 1.6 : Java Logical Operators

Operator	**Name**	**Description**	**Example**
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the	x < 5 || x < 4

		statements is true	
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

1.12. Java Garbage Collection

Java garbage collection is the process to perform automatic memory management which is done by java programs. Java programs get compile to generate the bytecode that can be run on a Java Virtual Machine. While executiong Java programs on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector identifies such unused objects so that memory can be freed by deleteing them.

Garbage Collection is process of deallocating the runtime unused memory automatically. In other words, it is a way to free the memory occupied by unused objects.

1.12.1 Advantage of Garbage Collection

- Garbage collector removes the unreferenced objects from heap which makes java **memory efficient**.
- No extra efforts are required as it is **automatically done** by the garbage collector(a part of JVM).

1.12.2 How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another

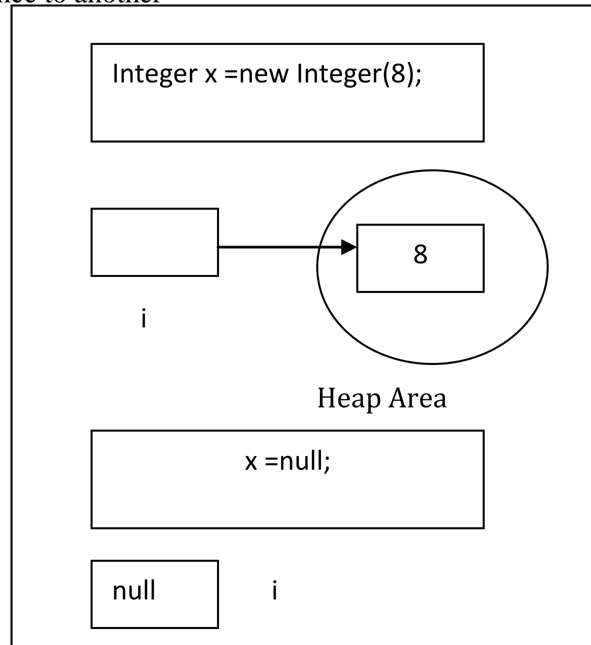


Fig. 1.6 : Object reference

1.12.3 finalize() method

The finalize() method is get invoked automatically each time before the object is garbage collected.

1.12.4 gc() method

The gc() method is used to call the garbage collector to perform cleanup processing. The gc() method is found in System and Runtime classes.

1.13. What are Java source file declaration rules?

A Java source file is a file containing human readable text called as plain text, this file containing Java source code and having .java extension. The .java extension indicates that the file is the Java source file. Java source code file contains source code for various object such asinterface, enumeration, class, or annotation type. There are some rules

associated to Java source file. We should followthe following rules while writing Java source code.

- We can write only one public class per source code file.
- Comments dscribe the information about the code and can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here. Java comment can be written anywhere in a program code where programmer need to describe some information.
- The name of the file must match the name of the class. Let us consider for example, a class declared as public class Dog { } must be in a source code file named Dog.java.
- If the class is member or part of some package, then the package statement must be the first line in the source code file, before any import statements that may be present.
- If there are import statements, they must go between the package statement (if there is one) and the class declaration. If there isn't a package statement, then the import statement(s) must be the first line(s) in the source code file. But Remember the class declaration must be the first line in the source code file, If there are no package or import statements. Import and package statements are applicable to all classes written in a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.
- A file can have more than one non~public classes written in it.
- Files with non~public classes can have a name that does not match any of the classes in the file

1.13.1 Java Naming Conventions

Java naming conventions are guidelines, which application programmers are expected to follow to generate a consistent and readable code throughout the application. If teams do not follow these conventions, they may collectively write an application code which is hard to read and difficult to understand.

Java frequently uses **Camel Case** notations for naming the variables,methods, constants etc. and **Title Case** notations for classes and interfaces.

Let's see these naming conventions in detail with examples.

1.13.2 Packages naming conventions

Package names conventions describe rule to write package name which must be a group of words starting with all lowercase field names (e.g. com, org, net etc.). Subsequent parts

of the package name may be different according to an organization's own internal naming conventions.

```
packagecom.howtodoinjava.webapp.controller;

packagecom.company.myapplication.web.controller;

packagecom.google.search.common;
```

1.13.3. Classes naming conventions

In Java, class names are written in title case with the first letter of each separate word capitalized and generally should be **nouns**. e.g.

```
PublicclassArrayList {}

PublicclassEmployee {}

PublicclassRecord {}

PublicclassIdentity {}
```

1.13.4. Interfaces naming conventions

In Java, interfaces names, generally, should be **adjectives**. Interfaces name should be written in title case with the first letter of each separate word capitalized. Some time, interfaces can be **nouns** as well when they present a family of classes e.g. `List` and `Map`.

```
publicinterfaceSerializable {}

publicinterfaceClonable {}

publicinterfaceIterable {}

publicinterfaceList {}
```

1.13.5. Methods naming conventions

Methods always should be **verbs**. They represent an action and the method name should clearly state the action they perform. The method name can have single or more than one words as needed to clearly represent the action. Words should be in camel case notation.

```
publicLong getId() {}

publicvoidremove(Object o) {}
```

```
publicObject update(Object o) {}

publicReport getReportById(Long id) {}

publicReport getReportByName(String name) {}
```

1.13.6. Variables naming conventions

All variables, instances, static and method parameter variable names should be in camel case notation. They should be short and enough to describe their purpose. Temporary variables can be a single character e.g. the counter in the loops.

```
publicLong id;

publicEmployeeDaoemployeeDao;

privateProperties properties;

for(inti = 0; i <list.size(); i++) {

}
```

1.13.7. Constants naming conventions

Java constants should be all **UPPERCASE** where words are separated by **underscore** character (“_”). Make sure to use final modifier with constant variables.

```
publicfinalString SECURITY_TOKEN = "...";

publicfinalintINITIAL_SIZE = 16;

publicfinalInteger MAX_SIZE = Integer.MAX;
```

1.13.8. Enumeration naming conventions

Similar to class constants, enumeration names should be all uppercase letters.

```
enumDirection {NORTH, EAST, SOUTH, WEST}
```

Outcomes:

- Students will able to study java introduction, features of java , learn JRE JDK JVM.
- Students will able to study history of java, difference between c++ and java, evolution of java.
- Students will able to study java variables, java identifiers, data types, java operators.
- Students will able to determine garbage collection in java.

Questions:

1. Write a Brief History of Java.
2. List and Explain the various features of Java.
3. Explain with neat diagram the working and execution steps of C++ and Java.
4. With neat diagram explain the various steps involved in execution of java programme.
5. With neat diagram explain the Java Environment Architecture
6. State and explain the terms JDK, JRE & JVM.
7. With example explain how to write a Class in java.
8. Define class and object in java.
9. What are variables with example explain the various types of variables.
10. Write a short note on Java identifiers.
11. Write a short note on Scope in java.
12. List and explain the various operators in java.
13. What is garbage collection in java explain in brief.
14. List and explain the various naming conventions in java.
15. What are various data types in java?

CHAPTER 2

DICISION MAKING AND LOOPS

Objectives:

- To study the execution of if, if – else, nested if else if statement in java program.
- To study the various loop statements like for, while, do while loops in java.
- To study function of switch statement, break statement and continue statement in java.
- To explain how to use comments in java program.

2.1 Java If-else Statement

The Java if statement is used to test such conditions which may result in either true or false. It checks Boolean condition: *true* or *false*. There are diffrentclassification of if statement in Java. In other words if statements are used to check conditions which may result in either true / false or Yes / No. if the condition results in true then if block will execute if the condition result in false the else block will executes. The various combinations of if-else are as follow

- if statement
- if-else statement
- if-else-if ladder
- nested if statement

2.1.1 Java if Statement

The Java if statement tests the condition. It executes the *if block* if the mentioned condition is true else it will exit the if block.

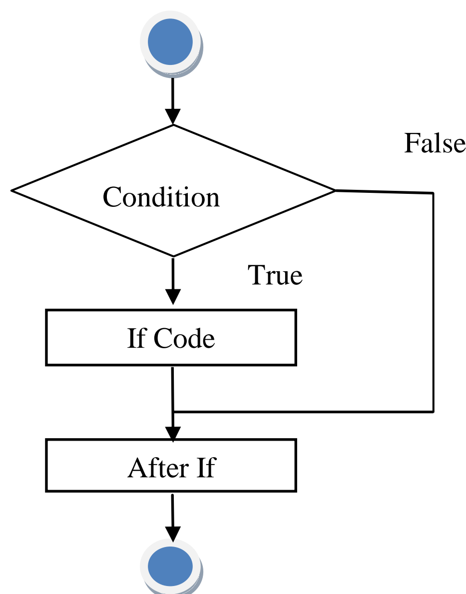


Fig. 2.1 : Java if statement Execution steps

Syntax:

```
if(condition){
//code to be executed  }
```

Example:

```
//Java Program to demonstrate the use of if statement.
public class IfExample {
public static void main(String[] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
   }
}
```

2.1.2 Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if the mentioned condition is true otherwise *else block* is executed. We can use if block alone but we can’t use else block alone we have to use else block always with if or with the combination of if statements. As mentioned above if statements are used to check conditions which may result in either true / false or yes / No.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```

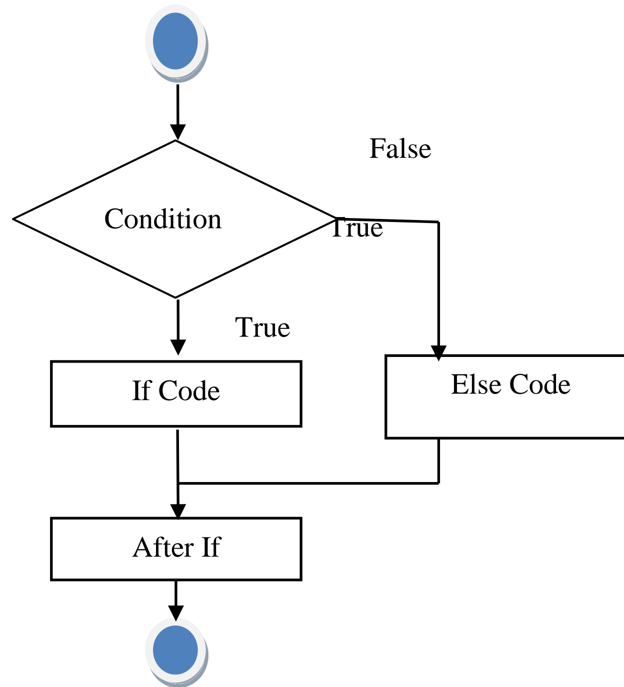


Fig. 2.2 : Java if – else statement Execution steps

Example: Let us consider the following Java Program to understand the use of if-else statement.

```
//It is a program to chech whether given number is odd or even odd and even number.

public class IfElseExample {

public static void main(String[] args) {

   //defining a variable

   int num=31;

   //Check if the number is completely divisible by 2 or not

   if(num%2==0){

      System.out.println("even number");

   }else{

      System.out.println("odd number");

   }

}

}
```

2.1.3. Java if-else-if ladder Statement

The if-else-if hierarchy statement executes only one condition from multiple statements. It can be used when you have to check multiple conditions, and based on satisfied condition need to check associated statements. It will execute the single block of statements associated with condition which get satisfied.

Syntax:

```
if(condition1){

//This code will be executed if condition 1is true

}else if(condition2){

//This code will be  executed if condition2 is true

}

else if(condition3){

//This code will be executed if condition3 is true

}

...

else{

//This code will be executed if all the conditions are false

}
```

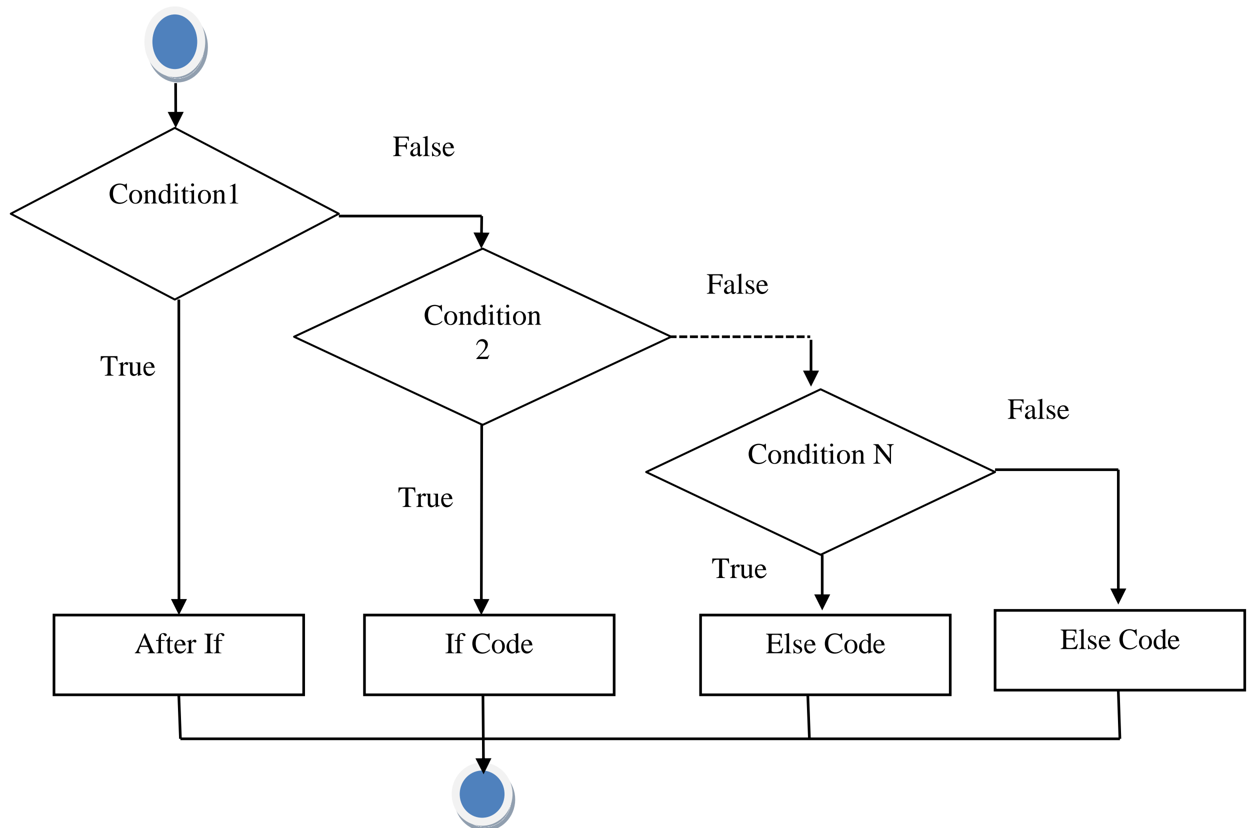


Figure 2.3 : Java if-else-if statement Execution steps

Example:

Consider the followingJava Program to demonstrate the use of If else-if ladder. It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.

```
public class IfElseIfExample {

public static void main(String[] args) {

   int marks=65;


   if(marks<50){

      System.out.println("fail");

   }

   else if(marks>=50 && marks<60){

      System.out.println("D grade");

   }

   else if(marks>=60 && marks<70){

      System.out.println("C grade");
```

```
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

2.1.4 Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
    //code to be executed
        if(condition){
            //code to be executed
    }
}
```

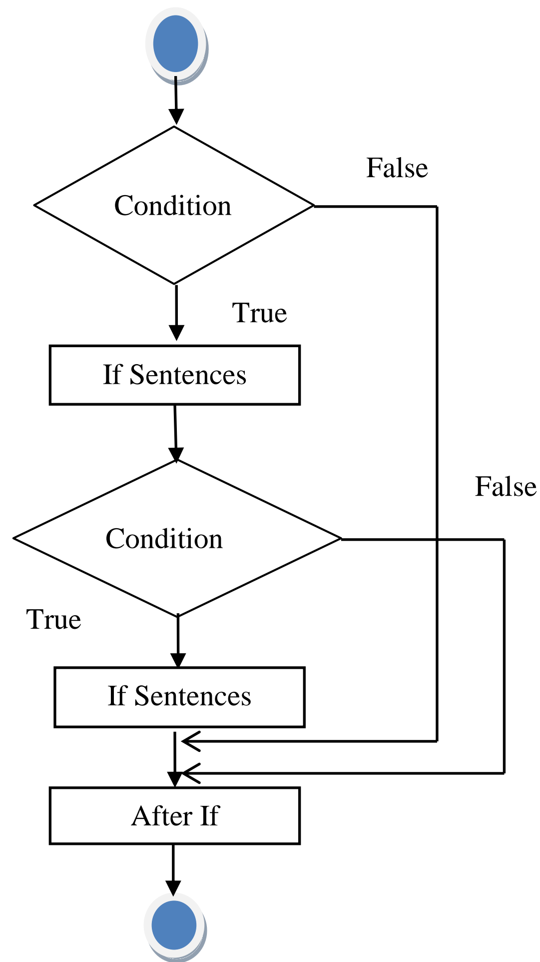


Fig. 2.4 : Java Nested if statement Execution steps

Example:

```
//Java Program to demonstrate the use of Nested If Statement.

public class JavaNestedIfExample {

public static void main(String[] args) {

    //Creating two variables for age and weight

    int age=20;

    int weight=80;

    //applying condition on age and weight

    if(age>=18){

        if(weight>50){

            System.out.println("You are eligible to donate blood");

        }
```

```
    }
}}
```

2.1.5 Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a simplest way to check the condition. If the condition is true then resultsof**'?'**is returnedBut, if the condition is false, the result of **':'** is returned.

Example:

```
public class IfElseTernaryExample {
public static void main(String[] args) {
    int number=13;
    //Using ternary operator
    String output=(number%2==0)?"even number":"odd number";
    System.out.println(output);
}
}
```

2.2 Loops in Java

In programming languages, loops also called as control structures are used to execute a set of statements/ instructions/functions repeatedly when some conditions become true. There are three types of loops in Java. Each are having different syntaxes but there is large similarity between the execution steps of each of the following loops.

1. for loop
2. while loop
3. do-while loop

2.2.1 Java for Loop

The Java *for loop* is used to repeator execute a part of the program several times. It is recommended to use for loop, if the number of iteration are fixed. In other words for loop can be used to execute the same condition and associated statements multiple times.

2.2.1.1 Java Simple for Loop: A simple for loop is the same as C/C++. First initialize the variable, check the required condition and increment/decrement value. It consists of four parts:

1. **Initialization**: in for loop ititialization is the initial condition which is executed only once when the loop starts. In initialization one can initialize the variable, or can use an already initialized variable. It is an optional condition.

2. **Condition**: It is the second part called as condition which is executed each time to test the required condition of the loop. It continues execution until the condition is false. It return boolean value either true or false. It is an optional condition.

3. **Statement**: The statement of the loop is executed each time until the second condition is false.

4. **Increment/Decrement**: It is the fourth and last part of loop used to increments or decrements the variable value. It is an optional condition.

Syntax**:**

```
for(initialization;condition;incr/decr)
{
//statement or code to be executed
}
```

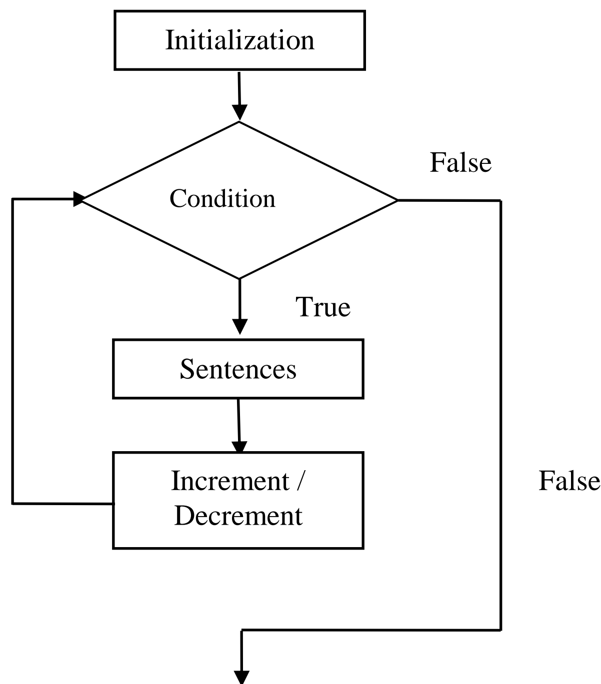


Fig. 2.5 : Java for loop Execution steps

Example:

```
//Java Program to demonstrate the example of for loop

//which prints table of 1

public class ForExample {

public static void main(String[] args) {

   //Code of Java for loop

   for(int i=1;i<=10;i++){

      System.out.println(i);

   }

}

}
```

2.2.1.2 Java Nested For Loop: Writing for loop inside the loop, it is known asnested for loop. The inner loop get executes completely whenever outer loop executes.

Example:

```
public class NestedForExample {
```

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

//loop of i

for(int i=1;i<=3;i++){

//loop of j

for(int j=1;j<=3;j++){

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

}//end of i

}//end of j

}

}
```

2.2.2 Java While Loop

The java *while loop* iterate a part of the program containing executable statements and instructions multiple times. It is recommended to use while loop if the number of iteration is not fixed. Similar to for loop can be used to execute the same condition and associated statements multiple times.

Syntax:

```
while(condition){

//code to be executed

}
```

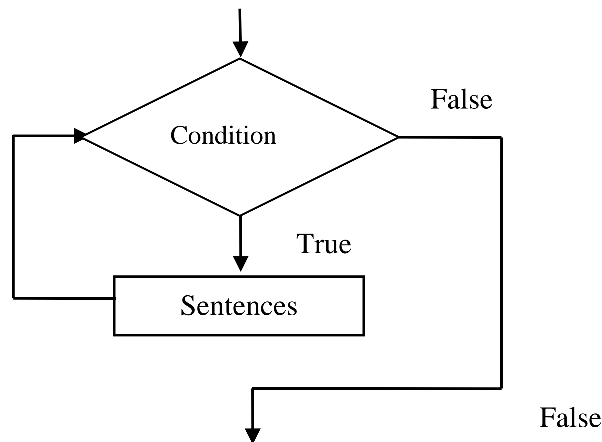


Fig. 2.6 : Java while loop Execution steps

Example:

```
public class WhileExample {
public static void main(String[] args) {
   int i=1;
   while(i<=10){
      System.out.println(i);
   i++;
   }  } }
```

2.2.3 Java do-while Loop

The Java do-while loop iterate a part of the program containing executable statements and instructions multiple times. It is recommended to use do-while loop if the number of iteration is not fixed and you must have to execute the loop at least once. Similar to for loop and while loopdo-while loop can be used to execute the same condition and associated statements multiple times.

The Java *do-while loop* is executed at least once as it executes the loop body first and then condition is checked.

Syntax:

```
Do{
  //code to be executed
  }while(condition);
```

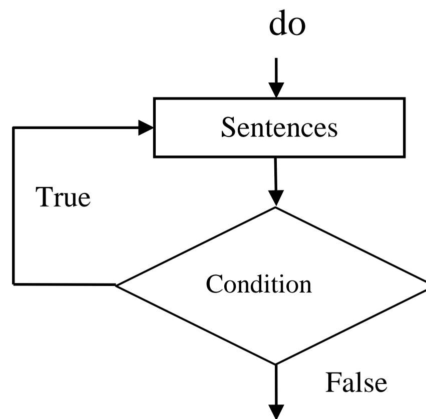


Fig. 2.7 : Java do while loop Execution steps

Example:

```
public class DoWhileExample {
public static void main(String[] args) {
  int i=1;
  do{
    System.out.println(i);
  i++;
  }while(i<=10);
}
}
```

2.3 Java for Loop vs. While Loop vs. Do While Loop

Table 2.1 : Difference between for loop, while Loop and do-while loop

Comparison	for loop	while loop	do while loop
Introduction	It is a control flow statement that iterates a part of the programs multiple timeson satisfying the given boolean condition.	It is a control flow statement that executes a part of the programs repeatedly on satisfying the given	It is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given

		boolean condition.	boolean condition.
When to use	It is recommended to use for loop, If the number of iteration is fixed.	It is recommended to use while loop, If the number of iteration is not fixed.	it is recommended to use the do-while loop, If the number of iteration is not fixed and you must have to execute the loop at least once.

Syntax	for(init;condition;incr/decr){
	// code to be executed
	}
		while(condition){
		//code to be executed
		}
			do{
			//code to be executed
			}while(condition);

Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//while loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do-while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

2.4 Java Switch Statement

The Java *switch statement* executes only one statement from given multiple conditions. It is like if-else-if ladder statement. The switch statement works with different data types such as byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. As well Java 7 onwards strings data type is also allowed in the switch statement.

In simple words, the switch statement checksathe equality of a variable against multiple values and execute the one which is matched.

Points to Remember

- There can be *one or more up to N number of case values* for a switch expression.

- The case value must match with switch expression type. The case value must be *literal or constant*. It doesn't allow variables.

- The case values must be *unique*. In case of duplicate value, it renders compile-time error.

- The Java switch expression support only mentioned data types which are*byte, short, int, long (with its Wrapper type), enums and string*.

- Each case statement can be written with a *break statement* which is not mandatory. When control reaches to the break statement, it jumps the control to the statement after the switch expression. It executes the next case if a break statement is not found.

- The case value can be written with a default label to describe the situation if none of the case executed but it is optional.

Syntax**:**

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

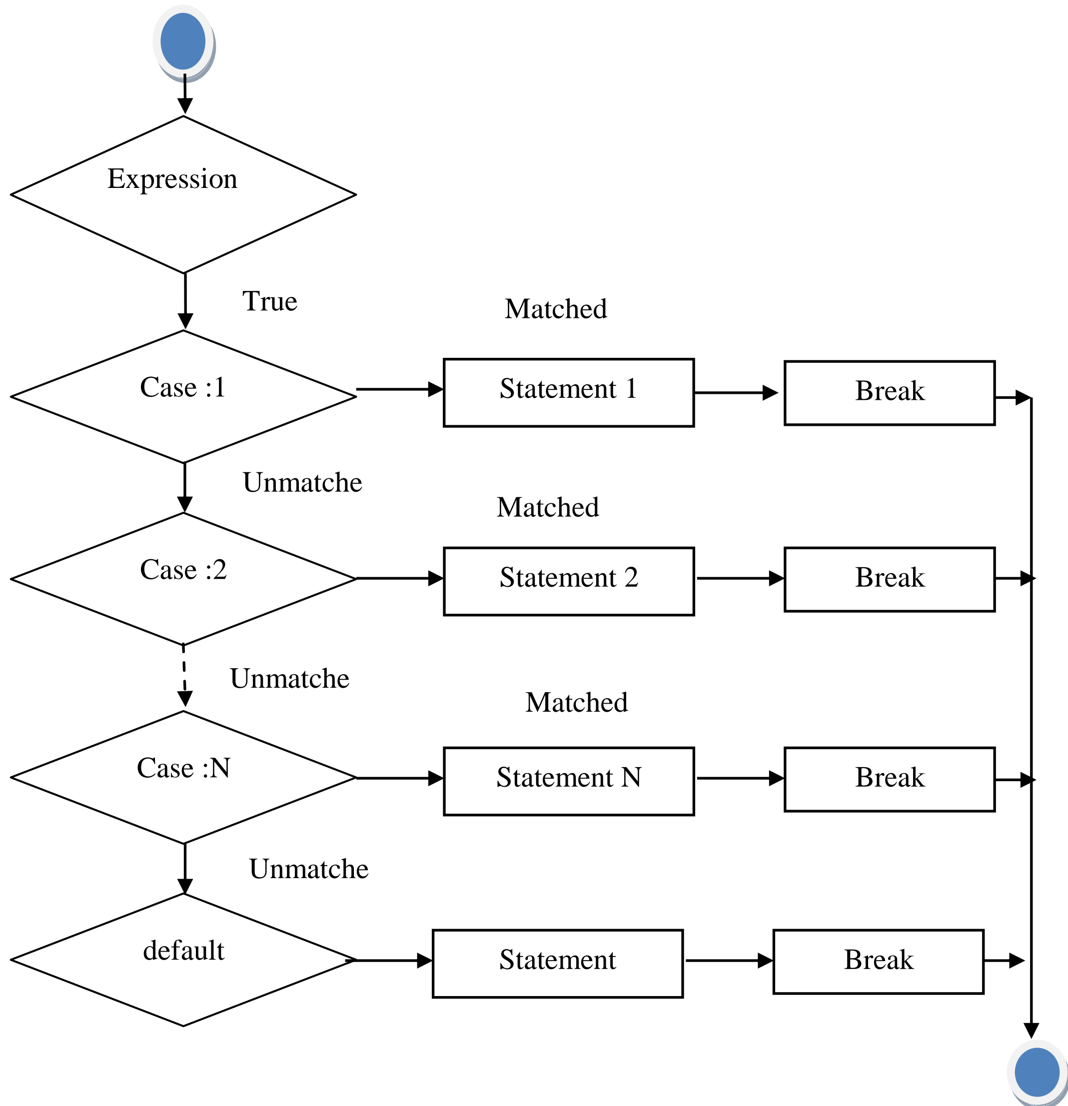


Fig. 2.8 : Java switch statement Execution steps

Example:

```
public class SwitchExample {

public static void main(String[] args) {

    //Declaring a variable for switch expression

    int number=20;

    //Switch expression

    switch(number){

    //Case statements
```

```
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

2.5 Java Break Statement

A break statement is used to terminate the currently executing loop. Onces the break statement encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement is used to break currently executing loop or switch statement. It breaks the current flow of the program on satisfying the specified condition. In case if it is written in inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax**:**

```
jump-statement;
 break;
```

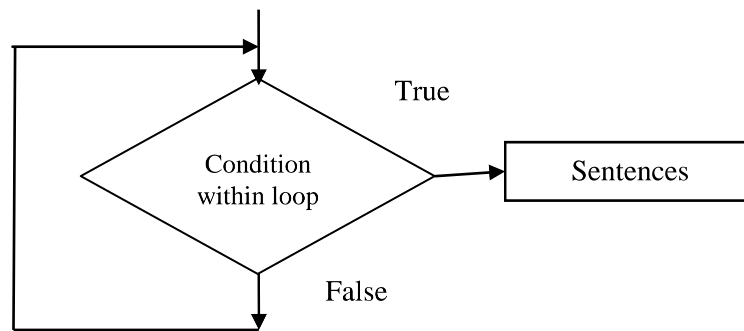


Fig. 2.9: Java break statement Execution steps

Example:

```
//Java Program to demonstrate the use of break statement inside the for loop.

public class BreakExample {

public static void main(String[] args) {

   //using for loop

   for(int i=1;i<=10;i++){

      if(i==5){

         //breaking the loop

         break;

      }

      System.out.println(i);

   }  } }
```

2.6 Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java *continues statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
jump-statement;
continue;
```

Example:

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
   //for loop
   for(int i=1;i<=10;i++){
      if(i==5){
         //using continue statement
         continue;//it will skip the rest statement
      }
      System.out.println(i);
   }  }}
```

2.7 Java Comments

The Java comments are the non executable statements, Compiler and interpreter does not execute these statements. The comments can be used to provide more detailed information or explanation about the variable, method, class or any statement written in a programme. It can also be used to hide program code.

2.7.1 Types of Java Comments

There are three types of comments in Kava.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

2.7.1.1 Java Single Line Comment: The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
    int i=10;//Here, i is a variable
    System.out.println(i);
}
}
```

2.7.1.2 Java Multi Line Comment: The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This  is
multi line
comment
*/
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
    int i=10;
    System.out.println(i);  }}
```

2.7.1.3 Java Documentation Comment: The documentation comment are the type of comment which are used to create documentation API. To create documentation API we require to use javadoc tool.

Syntax:/*This is documentation comment */

Example:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/

public class Calculator {

/** The add() method returns addition of given numbers.*/

public static int add(int a, int b){return a+b;}

/** The sub() method returns subtraction of given numbers.*/

public static int sub(int a, int b){return a-b;}  }
```

Outcomes :

- Students will able to study the execution of if, if – else, nested if else if statement in java program.
- Students will able to study the various loop statements like for, while, do while loops in java.
- Students will able to study function of switch statement, break statement and continue statement in java.
- Students will able to explain how to use comments in java program.

Questions:

1. What is decision making how it is possible in java.
2. What are the loops? List and explain the various types of loops.
3. What if-Else Loop? When it is used ? Explain with suitable example.
4. What you mean by Java if-else-if ladder Statement? Explain with neat diagram and example.
5. With example explain Nested if statement in java.
6. Explain the use of temporary operator with example.
7. List and explain the various loops in java.
8. Compare While and do While loop with suitable example.
9. What is the use of for loop? When is used with suitable example explain for loop.
10. Explain switch statement with suitable example.
11. What is use of break and continue statements in java?
12. What are comments in java what are the various types of comments?

CHAPTER 3

IMPLEMENTATION OF METHODS

Objectives:

- To study the working of class, methods, constructor, types of constructors in java.
- To explain the method overloading, constructor overloading, method overriding in java program.
- To study the inheritance, types of inheritance in java.
- To determine interfaces in java, use of super keyword with class and methods, etc.

3.1 Introduction:

3.1.1 Class

Class is the Entity which binds the all data members and data functions together.

It can also be defined as class is the basic building block of an object-oriented language which is a template that describes the data and behavior associated with instances of that class. When you declare a class you create an object that looks and feels like other instances of the same class.

Objects are created from class which is a user defined blueprint or prototype. Class describe and bind together the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers:** A class can be public or has default access.
2. **Class name:** The class name should be written with an initial letter (capitalized by convention).
3. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, proceeded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Syntax to Write a Class:

```
Keyword class ClassName
{
\\ Variable Declarations
main function ()
{
\\Your Logic comes here
}
}
```

Example:

```
class HelloWorld
{
int i;   // Variable Declaration
public static void main(String [] args)   //Main Function Declaration
{
int j;
System.out.println(" Hello World Welcome to Java");
}
}
Output: Hello World Welcome to Java
```

3.2 Methods in Java

A method is a collection of executable statements or instructions that perform assigned task and return the result to the caller. A method can also perform assigned task without returning anything. Methods allow reuseabillity of code without writting the code again and again. In Java, every method must be member of some class which is different from languages like C, C++, and Python.

Methods allow reusability of code which saves the timeand help us to reuse the code without retyping the code.

3.2.1 Method Declaration

Method declaration are associated with following various terminologies

1. **Modifier:** Defines **access modifier also called access type** of the method i.e. from where it can be accessed in your application and define the access scope. Javasupports four different types of the access specifiers.
2. **public:**this **access type**makes class member accessible in all class in your application.
3. **protected:** accessible within the class in which it is defined and in its **subclass(es)**
4. **private:** accessible only within the class in which it is defined.
5. **default (declared/defined without using any modifier):** accessible within same class and package within which its class is defined.
6. **The return type:** The data type of the value returned by the method or void if does not return a value.
7. **Method Name:** the rules for field names apply to method names as well, but the convention is a little different.

8. **Parameter list:** Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
9. **Exception list:** The exceptions you expect by the method can throw, you can specify these exception(s).
10. **Method body:It is collection of executable statements, it is**collection of instructions you need to be executed to perform your expected operations.It is enclosed between braces..

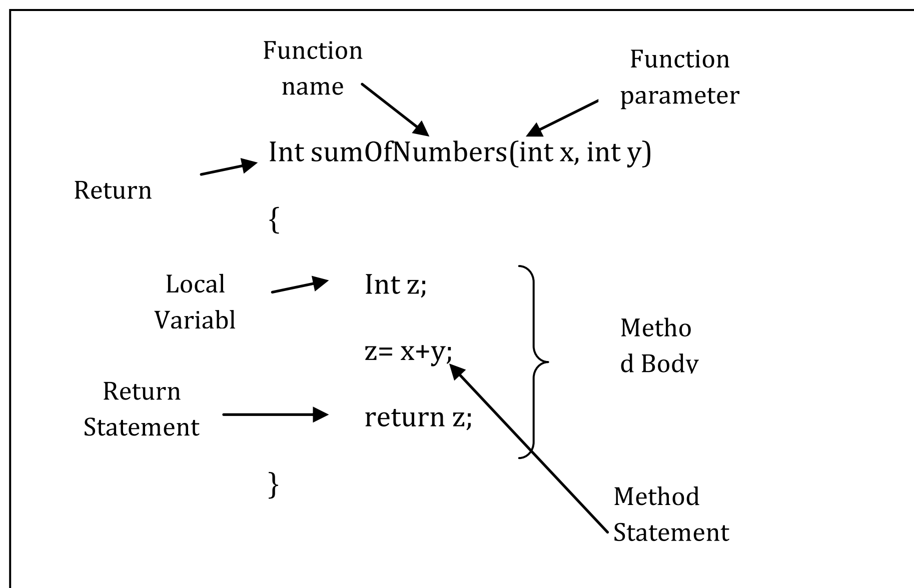


Fig. 3.1 : Method declaration

3.2.2 Method signature

It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type of method and exceptions are not considered as part of method signature.

Method Signature of above function:

```
sumOfNumbers(int x, int y)
```

3.2.3 Calling a method

The method needs to be called for its functionality.A method returns to the caaler or the code that invoked it in any of the following situations:

- It completely executes the all statements in the method
- It reaches a return statement

- Throws an exception

Example:

```
classMain{

publicstaticvoid main(String[] args){
System.out.println("About to encounter a method.");

// method call
      myMethod();

System.out.println("Method was executed successfully!");
}

// method definition
privatestaticvoid myMethod(){
System.out.println("Printing from inside myMethod()!");
}
}
```

Output:

```
About to encounter a method.
Printing from inside myMethod().
Method was executed successfully!
```

Example: Return Value from Method

Let's take an example of method returning a value.

```
classSquareMain{
publicstaticvoid main(String[] args){
int result;
      result = square();
System.out.println("Squared value of 10 is: "+ result);
}
publicstaticint square(){
// return statement
return10*10;
}
}
```

Output:

```
Squared value of 10 is: 100
```

3.3 What is Constructor?

In Java, a *constructor* is a block consisting of executable sentences or instructions just similar to the method. It is called automatically when an instance of the class (object) is created, and memory is allocated for the object. It is named as constructor because it constructs the values at the time of object creation. Writing a constructor for a class is not nessesary, class may have zero or any number of constructors. If your class doesn't have any constructor java compiler creates a default constructor for your class. You can say it is a special type of method which is used to initialize the object.

3.3.1 When Constructor is called

When an object is created using new () keyword the constructor is get called. It calls a default constructor. After execution of object creation statement constructor is get called and starts its execution.

Example:

```
Employee b=new  Employee ();

Employee c=new Employee (20,"Ajay");
```

While calling parametric constructor we need to take an extra care about the type and sequence of parameters defined in a constructor. We need to pass parameters of same type and of same sequence which we have specified in constructor declaration .

3.3.2 Rules for Writing Java constructor

1. Constructor must have the same name as its class name
2. A Constructor does not return any values it must not have explicit return type
3. A Java constructor cannot be created as abstract, static, final, and synchronized
4. Constructor can have access modifiers in declaration to control its access i.e. which other class can call the constructor.

3.3.3 Types of constructor

There are three type of constructor in Java:

3.3.3.1. No-argument constructor:A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates default constructor (with no arguments) for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
Depending on the typeDefault constructor provides the default values to the objectlike 0, null, etc. A constructor which doesn't have ant parameters is called as "Default Constructor".

Syntax of default constructor**:**

```
<class-name> (){}
```

Example of default Constructor

//Java Program to create and call a default constructor//Java Program to create and call a default c onstructor

```
class Employee{

Employee()            //creating a default constructor
{
System.out.println("Employee Class Default Constructor");}
public static void main(String args[]){           //main method
Employee b=new  Employee ();     //calling a default constructor
}  }
```

What is the purpose of a default constructor?

Depending on the type Default constructor provides the default values to the objectlike 0, null, etc.

3.3.3.2. Java Parameterized Constructor:

A constructor which has a finite number of parameters is called a parameterized constructor. It is used if we want to initialize fields of the class with your own values, and then use a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor provides different values to the distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of ConstructorDemo class that has two parameters. We can write any number of parameters in the constructor defination.

```
class ConstrucrorDemo
{
	int a;
	String s;
	ConstrucrorDemo()
	{
		System.out.println("U r in Default Constructor");
		a=10;
		s="Te comp";
	}
	ConstrucrorDemo(int a,String s)
	{
		System.out.println("U r in Parametric Constructor");
		this.a=a;
		this.s=s;
	}
	void display()
	{
		System.out.println("U r in function now");
		System.out.println("The value of A is==>"+a);
		System.out.println("The value of S is==>"+s);
	}
	public static void main(String[] args)
	{
		System.out.println("Hello World!");
		ConstrucrorDemo c=new ConstrucrorDemo();
		System.out.println("Using the ref of Default Constructor Values are==>");
		c.display();
		ConstrucrorDemo c1=new ConstrucrorDemo(20,"T3 Batch");
		System.out.println("Using the ref of Parametric Constructor Values are==>");
		c1.display();
	}
}
```

Output:

```
Hello World!

U r in Default Constructor

Using the ref of Default Constructor Values are==>

U r in function now

The value of A is==>10

The value of S is==>Te comp

U r in Parametric Constructor

Using the ref of Parametric Constructor Values are==>

U r in function now

The value of A is==>20

The value of S is==>T3 Batch
```

3.3.3.3. Copy constructor

A copy constructor is a type of constructor which is a member function used to initialize an object using another object of the same class. A copy constructor has the following general function prototype:

```
class Demo {
      private double r, m;
        // copy constructor
    Demo(Democ) {
        System.out.println("Copy constructor called");
        r = c.r;
        m = c.m;
    }

    // Overriding the toString of Object class
    @Override
    public String toString() {
        return "(" + r + " + " + m + "i)";
    }
  public static void main(String[] args) {
```

```
    Democ1 = new Demo(10, 15);
      // Following involves a copy constructor call
    Democ2 = new Demo(c1);

    System.out.println(c2); // toString() of c2 is called here
} }
```

3.4 Method Overloading

Method Overloading is a object oriented feature that allows to write a class with more than one method having the same name, and diffrent argument list. Method overloading is similar to constructor overloading in Java that allows a class to have more than one constructor with different argument lists.

3.4.1 Advantage of method overloading

1. Default constructor provides the default values to the object depending Method overloading is used to increase the readability of the program.

3.4.2 Different ways to overload the method

The different ways to overload the methods are as follow

1. By changing number of arguments
2. By changing the data type

3.4.2.1 Method Overloading: changing no. of arguments

Consider the following example in which, we have created two add methods, first add() method having two parameters and performs addition of two numbers and second adds method having three parameters and performs addition of three numbers.

```
class AddDemo{
public int add(int a,int b)
{return a+b;}
public int add (int a, int b, int c)
{return a+b+c;}
}
public static void main(String[] args){
                    System.out.println(AddDemo.add(14,10));
```

```
System.out.println(AddDemo.add(21,11,31));
}
```

3.4.2.2 Method Overloading: changing data type of arguments

In this example, we have created two methods that differ in data type. The first add method receives two float arguments and second add method receives two integer arguments.

```
class AddDemo {
int add(float a, float  b)
{return a+b;}
double add(int  a, int  b)
{return a+b;}
}
public static void main(String[] args){
System.out.println(AddDemo.add(31,12));
System.out.println(AddDemo.add(12.3,32.1));
}
```

3.5 Constructor overloading:

Like methods, constructors can also be overloaded. It allows having *more than one constructor inside one Class but with different signature*. Constructor overloading is anobject oriented concept of having more than one constructor with different parameters list, but each constructor is defined toperform a different task. The constructors parameters can be differs in number of parameters, data types of parameters, order of parameters

3.5.1 Important points related to Constructor overloading:

1. Constructor overloading and method overloading are similar concepts in Java.

2. Overloaded constructor can be called by using this() keyword in Java.

3. Overloaded constructor must be called from another constructor only.

4. You have to add no argument that is default constructor because compiler will not add any constructor if you have added another constructor in program.

5. Calling statement to overloaded constructor must be the first statement of constructor in java.

6. Writting one primary constructor and let overloaded constructor callsthat is the best programming practice. This strategy will make your initialization code centralized and will also be easier to test and maintain.

3.6 Method Overriding

Overriding is an object-oriented programming feature that allows a subclass or child class to write a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the samesignature which includes name, parameters and return type (or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.

Method overriding is one of the ways to achieve Run Time Polymorphism. The instance that is used to invoke the method determines the version of a method that is executed. If an instance of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an instance of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* that determines which version of an overridden method will be executed.

Example:

```
class OverridingDemo
{
        public void aMethod()
        {
                System.out.println("Super class method");
        }
}
```

```
class OverridingDemoTest extends OverridingDemo
{
        public void aMethod()
        {
                System.out.println("sub class amethod");
        }
        public static void main(String [] args)
        {
                OverridingDemoTest t=new OverridingDemoTest();
```

```
            t.aMethod();
      }
}
```

Output: sub class method

3.6.1 Rules for method overriding:

1. Overriding and Access-Modifiers: The access modifier for an overriding method can have more, but not less, access than the overridden method. Let us understand this with an example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

2. Final methods cannot be overridden: Declaremethod as final if you don't want a method to be overridden.

3. Static methods cannot be overridden: (Method Overriding vs. Method Hiding): Writinga static method with same signature as a static method in base class, is known as method hiding.

4. Private methods cannot be overridden: Private methods cannot be overridden as they are bonded during compile time. Hence we can't override private methods in a subclass.

5. The overriding method must have same return type (or subtype): From Java 5.0 onwards it is possible to have different return type for an overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomenon is known as covariant return type.

6. Invoking overridden method from sub-class we can call parent class method in overriding method using super keyword.

7. Overriding and constructor: We cannot override constructor as parent and child class can never have constructor with same name (Constructor name must always be same as Class name).

8. Overriding and Exception-Handling: Below are two rules to note when overriding methods related to exception-handling.

Rule#1: If the super-class overridden method does not throws an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

Rule#2: If the super-class overridden method throws an exception, subclass overriding method can only throw either same or subclass exception. If it is throwing parent exception in Exception hierarchy it will lead to compile time error. If subclass overridden method is not throwing any exception then it will not lead to any error.

9 Overriding and abstract method**:** All theAbstract methods in an interface or abstract class are need to be overridden in derived classes otherwise it will lead to a compile-time error.

10. Overriding and synchronized/strictfp method**:** The presence of synchronized/strictfp modifier with method has no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non-synchronized/strictfp one and vice-versa.

3.7 Final Keyword in Java

The final keyword in java is used to limit the user by adding more restrictions. The java final keyword can be used in many contexts.

Final can be:

1. variable
2. method
3. class

The final keyword can be written with the variables.A final variable that is not initialised norhave any value is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. If the blank final variable is made static then it will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

3.7.1. Java final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example of final variable

There is a final variable speed limit, we are going to change the value of this variable, but it can't be changed because final variable once assigned a value can never be changed.

```
class BikeSpeed{
    final int speedlimit=90;//final variable
    void run(){
    speedlimit=400;
    }
 public static void main(String args[]){
    Bike9 obj=new  Bike9();
```

```
obj.run();
}
}//end of class
```

Output: Compile Time Error

3.7.2. Java final method

Final keyword can be used with methods but the final method cannot be override.

Example of final method

```
class Bike{
    final void run(){System.out.println("running");}
}

class Honda extends Bike{
        void run(){System.out.println("running safely with 100kmph");}

    public static void main(String args[]){
    Honda honda= new Honda();
        honda.run();
        }
}
```

Output: Compile Time Error

3.7.3 Java final class

Final keyword can be used with class but the final class is not allowed to extend or inherit in sub class.

Example of final class:

```
final class Bike{}

class Honda1 extends Bike{
    void run(){System.out.println("running safely with 100kmph");}

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

```
Honda1 honda= new Honda1();
 honda.run();
}
}
```

Output: Compile Time Error

3.8 Static keyword in java

The static keyword in Java is used with variables, methods, blocks and nested classes. The purpose of static keyword is for memory management.When static keyword is used itshares the same variable or method of a given class. This is used for a constant variable or a method that is the same for every instance of a class. That's why main method of a class is usually labeled static.

3.8.1 Static variable

The variable declared with static keyword is known as static variable.

- A Static variable is used to fulfill the common requirement. Let us understand with Example the Company name of employees, the college name of students, etc. The Name of the college is common for all students.
- The static variable allocates memory at the time of class loading and only once in the class area.

3.8.1.1 Advantage of static variable: Using a static variable we make our program memory efficient (i.e. itsaves memory).

3.8.1.2 Need of static variable: Let us consider we need to a store record of all students of any institute, in this case, student id or PRN number is unique for every student but institutename is common for all. When we create a static variable as a institute name then only once memory is allocated otherwise it allocates a memory space each time for every student.

Syntax:

```
public static datatype VariableName;

Example: public static int a;
```

3.8.2 Difference between static and final keyword

Static keyword fixed the memory whereas final keyword fixed the value that means it makes variable values constant. Making the variable static means that it will be located only once in the program and making variable final will not allow to change the value of final variable.

3.8.3 Static Method in Java

A method declared wit static keyword is known as static method, suchmethod belongs to the class not to the object of a class. Static methoddoes not require creating an instance of a class for invoking it. It can access and modify static data members.

Syntax:

```
public static void mymethod () {}
```

Example:

```
class A
{
void fun1()
{
System.out.println("Hello I am Non-Static");
}
static void fun2()
{
System.out.println("Hello I am Static");
}
}
class Person
{
public static void main(String args[])
{
A  oa=new  A();
   oa.fun1(); // non static method
   A.fun2();  // static method
}
}
```

```
Output:
Hello I am Non-Static
Hello I am Static
```

3.9 This keyword in java

this is a keyword in java also called as reference variable that refers to the current class object.

3.9.1 Usage of this keyword

- The keyword this can be used to refer current class instance variable.
- The keyword this can also be used be used to invoke current class constructor.
- The keyword this be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.
- To pass as argument in the constructor call this can be used.
- To return the current class instance this can also be used.

3.9.2 Why use this keyword in java?

The need of using this keyword is to differentiate the formal parameter and data members of class. If the formal parameter and data members of the class are similar then jvm get ambiguity (get confused between formal parameter and member of the class as no clear information)

The data member of the class must be preceded by "this" To differentiate them from formal parameter.

3.10 Inheritance in Java

The process of obtaining the data members and methods from one class to another class is known as inheritance. It is one of the fundamental features of object-oriented programming.

3.10.1 Important points

- In the inheritance the class which is extended to inherit to share data members and methods is known as base or super or parent class.

- The class which inheriting (reusing) the data members and methods from extended class is known as sub or derived or child class.
- The data members and methods of a class are known as class members or class features.
- The concept of inheritance allows us re-usability that is we can reuse the data members of super class in sub class.

3.10.2 Why use Inheritance?

- Inheritance offers Method Overriding (used for Runtime Polymorphism).
- Inheritance also allows to implement polymorphism and to be able to reuse code for different classes by putting it in a common super class
- Inheritance also offers code Re-usability that is write code once and use multiple times.

3.10.3 Types of Inheritance

Inheritance can be classified in different types based on the way of inheriting the classes:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

- **3.10.3.1 Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In the example given below, the class Fruit serves as a base class for the derived class Apple.

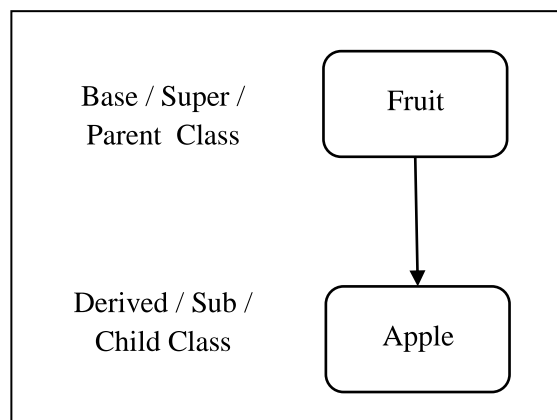


Fig. 3.2 : Single inheritance

Example: Consider the following implementation to understand Single Inheritance.

1. Parent /Super Class

```
class SuperClass
{
        int i;
        String s;
        SuperClass()
        {
                System.out.println("U r in Default Constructor Of Super Class");
                i=10;
                s="Super Class";

        }
        SuperClass(int i,String s)
        {
                System.out.println("U r in Parametric Constructor of Super Class");
                this.i=i;
                this.s=s;

        }
        void display()
        {
                System.out.println("U r in function of Sper Class");
                System.out.println("The Values of I==>"+i);
                System.out.println("The Values of S==>"+s);
        }
        }
```

2. Child/ Sub / derived Class

```
class SubClass extends SuperClass
{
        int j;
        String s1;
        SubClass ()
        {
                System.out.println("U r in Default Constructor Of Sub Class");
                j=10;
                s1="Sub Class";
```

```
	}
	SubClass (int j,String s1,int i,String s)
	{
		super(i,s);
		System.out.println("U r in Parametric Constructor Of Sub Class");
		this.j=j;
		this.s1=s1;


	}
	void display()
	{
		System.out.println("U r in function of Sub Class");
		System.out.println("The Values of I==>"+j);
		System.out.println("The Values of S==>"+s1);
	}
}
```

3. Class Containing Main function which will Create object of Sub class to access the Super class members.

```
class SperSubDemo
{
	public static void main(String[] args)
	{
		System.out.println("Hello World!");
		SubClass s=new SubClass();
		s.display();
		SubClass s1=new SubClass();
		s1.display();
	}
}
```

Output:

```
Hello World!
U r in Default Constructor Of Super Class
U r in Default Constructor Of Sub Class
U r in function of Sub Class
The Values of I==>10
```

```
The Values of S==>Sub Class
U r in Default Constructor Of Super Class
U r in Default Constructor Of Sub Class
U r in function of Sub Class
The Values of I==>10
The Values of S==>Sub Class
```

3.10.3.2 Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. Let us understand this withexample given bellow, the class Vehicle serves as a base class for the derived class Bike, which in turn serves as a base class for the derived class Honda Bike. In Java, a class cannot directly access the grandparent's members.

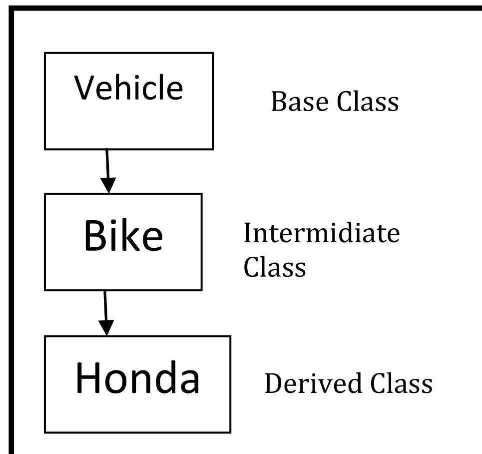


Fig. 3.3 : Multilevel inheritance

Example: Consider the following implementation to understand Multilevel Inheritance.

1. Parent /Super Class

```
class Student
{
        int sid;
        String name;
        String addrs;
        Student ()
        {
                System.out.println("U r in Default Constructor of Student Class");
                sid=10;
                name="Rama";
                addrs="Nashik";
        }
```

```
        Student (int i,String s,String d1)
        {
                System.out.println("U r in Parametric Constructor of Student Class");
                sid=i;
                name=s;
                addrs=d1;
        }

}
```

2. First Level Child/ Sub / derived Class

```
class EnggStudent extends Student
{
        String branch, clg;

        EnggStudent ()
        {
                System.out.println("U r in Default Constructor of Engineering Student Class");
                branch="CSE";
                clg="IIT Bombay";
        }
        EnggStudent (int i,String s,String a,String s1, String s2)
        {
                super(i,s,a);
                System.out.println("U r in Parametric Constructor of Engineering Student
Class");
                branch=s1;
                clg=s2;
        }

}
```

3. Second Level Child/ Sub / derived Class

```
class SecondYearStudent extends EnggStudent
{
        String batch;
```

```
	String result;;
	SecondYearStudent()
	{
		batch="2018-19";
		result="First class";
	}

	SecondYearStudent(int i,String s,String a,String s1, String s2,String d, String b)
	{
		super(i,s,a,s1,s2);
		batch=d;
		result=b;
	}

	public void display()
	{
		System.out.println("\n Student Id==>"+sid);
		System.out.println("\n Student Name==>"+name);
		System.out.println("\n Student College==>"+addrs);
		System.out.println("\n Student Branch==>"+branch);
		System.out.println("\n Student College==>"+clg);
		System.out.println("\n Student Batch==>"+batch);
		System.out.println("\n Student Result==>"+result);
	}
}
```

4. Class Containing Main function which will Create object of Sub class to access the Super class members.

```
class StudentDemo
{
	public static void main(String[] args)
	{
		System.out.println("Hello World!");
		SecondYearStudent s= new SecondYearStudent();
		s.display();
		SecondYearStudent s1= new
SecondYearStudent(10,"Ajay","Jalgaon","Mechanical","IIT Madras","2019-20", "First Class");
		s1.display();
```

```
        }
}
```

Output:

```
Hello World!
U r in Default Constructor of Student Class
U r in Default Constructor of Engineering Student Class
 Student Id==>10
 Student Name==>Rama
 Student College==>Nashik
 Student Branch==>CSE
 Student College==>IIT Bombay
 Student Batch==>2018-19
 Student Result==>First class
U r in Parametric Constructor of Student Class
U r in Parametric Constructor of Engineering Student Class
 Student Id==>10
 Student Name==>Ajay
 Student College==>Jalgaon
 Student Branch==>Mechanical
 Student College==>IIT Madras
 Student Batch==>2019-20
 Student Result==>First Class
```

3.10.3.3 Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.Consider the example given below, the class Car serves as a base class for the derived class Maruti,Hundai and Tata.

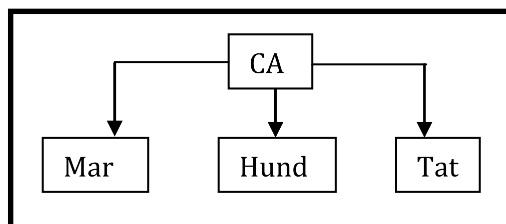


Fig. 3.4 : Hierarchical inheritance

Example: Consider the following implementation to understand Hierarchical Inheritance.

1. Parent /Super Class

```
class Student
{
                    int sid;
                    String name;
                    String addrs;
                    Student ()
                    {
                     System.out.println("U r in Default Constructor of Student Class");
                     sid=10;
                     name="Rama";
                     addrs="Nashik";
                    }
                    Student (int i, String s,String d1)
                    {
                     System.out.println("U r in Parametric Constructor of Student Class");
                     sid=i;
                     name=s;
                     addrs=d1;
                    }
}
```

2. First Child/ Sub / derived Class

```
class MedicalStudent extends Student
{
           String spl;
           String clg;;
           MedicalStudent()
           {
              System.out.println("\n U r in Default Constructor of Medical Student Class");

              spl="Heart";
              clg="JIIM";
           }

           MedicalStudent(int i, String s,String a, String d, String b)
           {
```

```
                super(i,s,a );
                System.out.println("U r in Parametric Constructor of Medical Student Class");
                spl=d;
                clg=b;
            }

            public void display()
            {
                System.out.println("\n Student Id==>"+sid);
                System.out.println("\n Student Name==>"+name);
                System.out.println("\n Student Address==>"+addrs);
                System.out.println("\n Student Specialization==>"+spl);
                System.out.println("\n Student College==>"+clg);
                }
}
```

3. Second Child/ Sub / derived Class

```
class EnggStudent extends Student
{
            String branch,clg;

            EnggStudent ()
            {
              System.out.println("U r in Default Constructor of Engineering Student Class");
              branch="C        SE";
              clg="IIT Bombay";
            }
            EnggStudent (int i,String s,String a,String s1, String s2)
            {
              super(i,s,a);
              System.out.println("U r in Parametric Constructor of Engineering Student
Class");
              branch=s1;
              clg=s2;
            }

             public void display()
```

```
        {
            System.out.println("\n Student Id==>"+sid);
            System.out.println("\n Student Name==>"+name);
            System.out.println("\n Student Address==>"+addrs);
            System.out.println("\n Student Branch==>"+branch);
            System.out.println("\n Student College==>"+clg);

        }
}
```

4. Class Containing Main function which will Create object of Sub class to access the Super class members.

```
class StudentDemo
{
    public static void main(String[] args)
    {
    System.out.println("\n\n Engineering Student Details\n");
    EnggStudent e= new EnggStudent();
    e.display();
    EnggStudent e1= new EnggStudent(10,"Ajay","Jalgaon","Mechanical","IIT Madras");
    e1.display();

    System.out.println("\n\n Medical Student Details\n");
    MedicalStudent m= new MedicalStudent();
    m.display();
    MedicalStudent m1= new MedicalStudent(10,"Vijay","Jalgaon","Brain","IIM");
    m1.display();
    }
}
```

Output:

```
Engineering Student Details
U r in Default Constructor of Student Class
U r in Default Constructor of Engineering Student Class
 Student Id==>10
 Student Name==>Rama
 Student Address==>Nashik
```

```
Student Branch==>CSE
Student College==>IIT Bombay
U r in Parametric Constructor of Student Class
U r in Parametric Constructor of Engineering Student Class
Student Id==>10
Student Name==>Ajay
Student Address==>Jalgaon
Student Branch ==>Mechanical
Student College==>IIT Madras

Medical Student Details
U r in Default Constructor of Student Class
U r in Default Constructor of Medical Student Class
Student Id==>10
Student Name==>Rama
Student Address==>Nashik
Student Specialization==>Heart
Student College==>JIIM
U r in Parametric Constructor of Student Class
U r in Parametric Constructor of Medical Student Class
Student Id==>10
Student Name==>Vijay
Student Address==>Jalgaon
Student Specialization==>Brain
Student College==>IIM
```

3.10.3.4 Multiple Inheritance (Through Interfaces): Multiple inheritance is another type of inheritance, one class can have more than one superclass and inherit features from all parent classes. Java does **not** support multiple inheritance directly that is by extending classes. But we can achieve multiple inheritance only by implementing the Interfaces. In the example given below, Class Child is derived from interface Father and Mother.

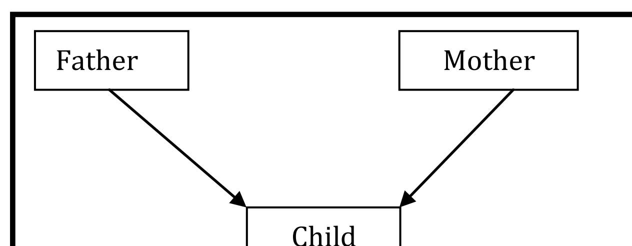


Fig. 3.5 : Multiple inheritance

Example: Consider the following implementation to understand MultipleInheritance.

1. Parent /Super Class

```
class Student
{
                    int sid;
                    String name;
                    String addrs;
                    Student ()
                    {
                     System.out.println("U r in Default Constructor of Student Class");
                     sid=10;
                     name="Rama";
                     addrs="Nashik";
                    }
                    Student (int i,String s,String d1)
                    {
                     System.out.println("U r in Parametric Constructor of Student Class");
                     sid=i;
                     name=s;
                     addrs=d1;
                    }

}
```

2. Interface (Acting as second parent Class)

```
class MedicalStudent extends Student
```

```
{
        String spl;
        String clg;;
        MedicalStudent()
        {
          System.out.println("\n U r in Default Constructor of Medical Student Class");

          spl="Heart";
          clg="JIIM";
        }

        MedicalStudent(int i,String s,String a, String d, String b)
        {
          super(i,s,a );
          System.out.println("U r in Parametric Constructor of Medical Student Class");
          spl=d;
          clg=b;
        }

        public void display()
        {
          System.out.println("\n Student Id==>"+sid);
          System.out.println("\n Student Name==>"+name);
          System.out.println("\n Student Address==>"+addrs);
          System.out.println("\n Student Specialization==>"+spl);
          System.out.println("\n Student College==>"+clg);

        }
}
```

3. Child/ Sub / derived Class

```
class EnggStudent extends Student
```

```
{
        String branch,clg;

        EnggStudent ()
        {
          System.out.println("U r in Default Constructor of Engineering Student Class");
          branch="C        SE";
          clg="IIT Bombay";
        }
        EnggStudent (int i,String s,String a,String s1, String s2)
        {
          super(i,s,a);
          System.out.println("U r in Parametric Constructor of Engineering Student 
Class");
          branch=s1;
          clg=s2;
        }

        public void display()
        {
          System.out.println("\n Student Id==>"+sid);
          System.out.println("\n Student Name==>"+name);
          System.out.println("\n Student Address==>"+addrs);
          System.out.println("\n Student Branch==>"+branch);
          System.out.println("\n Student College==>"+clg);


        }
}
```

4. Class Containing Main function which will Create object of Sub class to access the Super class members.

```
class StudentDemo
{
    public static void main(String[] args)
    {
    System.out.println("\n\n Engineering Student Details\n");
    EnggStudent e= new EnggStudent();
    e.display();
```

```
        EnggStudent e1= new EnggStudent(10,"Ajay","Jalgaon","Mechanical","IIT Madras");
        e1.display();

        System.out.println("\n\n Medical Student Details\n");
        MedicalStudent m= new MedicalStudent();
        m.display();
        MedicalStudent m1= new MedicalStudent(10,"Vijay","Jalgaon","Brain","IIM");
        m1.display();
        }
}
```

Output:

```
Engineering Student Details
U r in Default Constructor of Student Class
U r in Default Constructor of Engineering Student Class
 Student Id==>10
 Student Name==>Rama
 Student Address==>Nashik
 Student Branch==>CSE
 Student College==>IIT Bombay
U r in Parametric Constructor of Student Class
U r in Parametric Constructor of Engineering Student Class
 Student Id==>10
 Student Name==>Ajay
 Student Address==>Jalgaon
 Student Branch ==>Mechanical
 Student College==>IIT Madras
 Medical Student Details
U r in Default Constructor of Student Class
 U r in Default Constructor of Medical Student Class
 Student Id==>10
 Student Name==>Rama
 Student Address==>Nashik
 Student Specialization==>Heart
 Student College==>JIIM
U r in Parametric Constructor of Student Class
U r in Parametric Constructor of Medical Student Class
```

```
Student Id==>10
Student Name==>Vijay
Student Address==>Jalgaon
Student Specialization==>Brain
Student College==>IIM
```

3.10.3.5 Hybrid Inheritance (Through Interfaces): It is a type of inheritance in wich we can combine two or more of the above types of inheritance. But java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, hybrid inheritance can be achived or implemented only by implementing Interfaces.

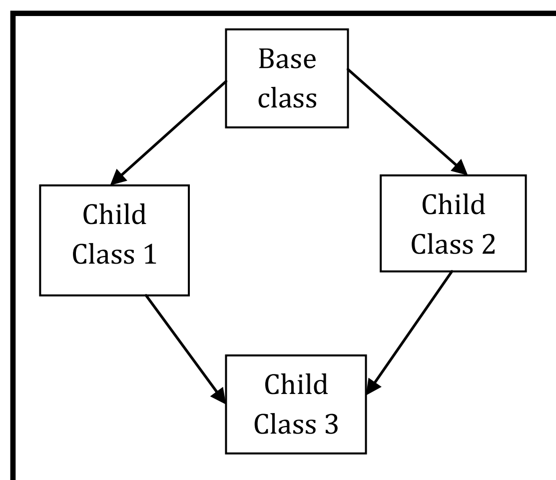


Fig. 3.6 : Hybrid inheritance

3.10.4 Important Rules Regarding Inheritance:

- In java programming one derived class can extends only one base class because java programming does not support multiple inheritance through the concept of classes, but it can be supported through the concept of Interface.
- Whenever we develop any inheritance application first creates an object of bottom most derived class but not for top most base class.
- When we create an object of bottom most derived class, first we get the memory space for the data members of top most base class, and then we get the memory space for data member of other bottom most derived class.
- The lowest derived class or leaf class in a herarchy class contains logical appearance for the data members of all heigher lever classes in herarchy or top most base classes.

- If we do not want to give the features of base class to the derived class then the definition of the base class must be preceded by final hence If the base class is marked as final, then final base classis not allowed to reuse or not to inherit.
- If we do not want to implement or extend some of the features of super class in to sub class or derived class then such features of base class must be markes as private, private features of base class are not allowed to accesse or inherit in derived class.
- Data members and methods of a base class can be inherited into the derived class but constructors of base class cannot be inherited because every constructor of a class is declared for initializing its own data members but not for initializing the data members of other classes.
- An object of base class can contain details about features of same class but an object of base class never contains the details about special features of its derived class (this feature is known as scope of base class object).
- For each and every class in java java.lang.Object is an implicit predefined super class. Because it providers garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.

3.10.5 Interfaces in Java

An interface in java is just similar to class which can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Syntax:

```
interface <interface_name> {
    // declare constant fields
   // declare methods that abstract (by default).
```

```
}
```

Example:

```
interface Player
{
   final int id = 10;
   int move();
}
```

A real-world example:

Let's consider the example of vehicles like bicycle, car, bike, they have common functionalities. So we create an interface toplace all these common functionalities in it. And lets Bicycle, Bike, car .etc. implement all these features and functionalities in the respective class in diffrent way.

```
import java.io.*;
 interface Vehicle {

   // all are the abstract methods.
   void changeGear(int a);
   void speedUp(int a);
   void applyBrakes(int a);
}

class Bicycle implements Vehicle{
  int speed;
   int gear;

    // to change gear
   @Override
   public void changeGear(int newGear){
              gear = newGear;
   }

   // to increase speed
   @Override
   public void speedUp(int increment){
              speed = speed + increment;
   }
```

```
    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
                speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed  " gear: " + gear);
    }
}

class Bike implements Vehicle {
    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){
                gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
                speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
                speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed  + " gear: " + gear);
    }
}
```

```
class GFG {

   public static void main (String[] args) {

       // creating an instance of Bicycle  doing some operations
       Bicycle bicycle = new Bicycle();
       bicycle.changeGear(2);
       bicycle.speedUp(3);
       bicycle.applyBrakes(1);

       System.out.println("Bicycle present state :");
       bicycle.printStates();

       // creating instance of the bike.
       Bike bike = new Bike();
       bike.changeGear(1);
       bike.speedUp(4);
       bike.applyBrakes(3);

       System.out.println("Bike present state :");
       bike.printStates();
   } }
```

```
Output:
Bicycle present state :
speed: 2 gear: 2
Bike present state:
Speed: 1 gear: 1
```

3.11 Super Keyword in Java

The super keyword in java is a keyword is used to refer parent class objects.It is also called as reference variable as is used to refer parent class objects. The keyword “super” came into the use with the concept of Inheritance. It is majorly used in the following contexts:

3.11.1. Use of super with variables:

The super keyword can be used with variabels, when a derived class and base class have same data members. It may cause ambiguity for the JVM. We can understand it more evidently using this code:

```
/* Base class vehicle */
classVehicle
{
   intmaxSpeed = 120;
}

/* sub class Car extending vehicle */
classCar extendsVehicle
{
   intmaxSpeed = 180;

   voiddisplay()
   {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: "+ super.maxSpeed);
   }
}

/* Driver program to test */
classTest
{
   publicstaticvoidmain(String[] args)
   {
      Car small = newCar();
      small.display();
   }
}
```

Output: Maximum Speed: 120

3.11.2. Use of super with methods:

This is used with methods to call them and only when we want to call parent class method. If a parent and child class have same named methods that is overriding methods then to resolve ambiguity we use super keyword. This piece of code helps to understand the use of super keyword.

```
/* Base class Person */
class Person
{
   void message()
   {
```

```
        System.out.println("This is person class");
    }
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

```
Output:

This is Student Class
```

```
This is Person Class
```

3.11.3. Use of super with constructors:

Super keyword can also be used to access the parent class constructor. The ‘’super’keyword can call both parametric as well as non-parametric constructors depending upon the situation. Following is the peace of code to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}



/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

3.11.4 Remember Important Rules while using Super:

1. Call to super () must be first statement in Derived (Student) Class constructor.
2. If a constructor does not explicitly invoke a constructor of superclass, the Java compiler automatically inserts a calling statement to the no-argument constructor of the superclass. If incasethe superclass does not have a no-argument constructor, compile-time error will be thrown by the compiler. Object *does* have such constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*.

Outcomes :

- Students will able to study the working of class, methods, constructor, types of constructors in java.
- Students will able to explain the method overloading, constructor overloading, method overriding in java program.
- Students will able study the inheritance, types of inheritance in java .
- Students will able to determine interfaces in java, use of super keyword with class and methods, etc.

Questions:

1. What is Class? What are various components of class?
2. What are the methods in java? What are various terminologies of methods in java?
3. With proper example explain the concept of calling the Methods.
4. What is Method signature?
5. What are the various types of Method calling? explain each with example.
6. What is constructor? What are the various rules for writing constructor.

7. With neat diagram explain the types of constructor.
8. What is method overloading? Explain with example.
9. What is method overriding? Explain with example.
10. What is difference between method overloading and method overriding?
11. What is Constructor overloading? Explain with example.
12. Write a short note on final keyword in java.
13. Write a short note on static keyword in java.
14. What is this keyword? Write a note on use of this keyword in java.
15. What is difference between static and final methods in java.
16. Explain the concept of inheritance? What are the important points related to inheritance in java.
17. Write a short note on various types of Inheritance.
18. Discuss the use of inheritance. Does multiple inheritance is supported in java? Justify your answer.
19. With suitable example explain multilevel inheritance.
20. With suitable example explain Hierarchical inheritance.
21. Explain in short the concept of interface in java.

CHAPTER 4

WRAPPER CLASSES, ARRAY AND STRING

Objectives:

- To study the function of wrapper classes, types of wrapper classes like integer, float, double, character, short, etc. in java .
- To study the function of constructors and methods of wrapper classes .
- To study multidimensional array in java.
- To study the working of array of objects in java.

4.1 Wrapper Classes in Java

A Wrapper class is a class which can contain a primitive data types, or we can say its object wraps a primitive data types. An object to a wrapper class contains a field and in this field primitive data types can be stored. In simple words, primitive values can be wrap a into a wrapper class object.

4.1.1 Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

2. The classes in java.util package handle only objects and hence wrapper classes help in this case also.

3. To store data elements because some of the Data structures in the Collection framework, such as ArrayList and Vector does not store not primitive types or values, but store only objects (reference types).

4. It is also needed to support synchronization in multithreading.

4.1.2 Various Wrapper Classes in java:

In java you may find various wrapper classes. Some of wrapper classes and their equivalent primitive data types are as follow.

Table 4.1 : Wrapper Classes

Sr. No.	Wrapper Class	Primitive Data Types
1	Character	Char
2	Short	short
3	Byte	byte
4	Integer	int
5	Float	float
6	Double	double

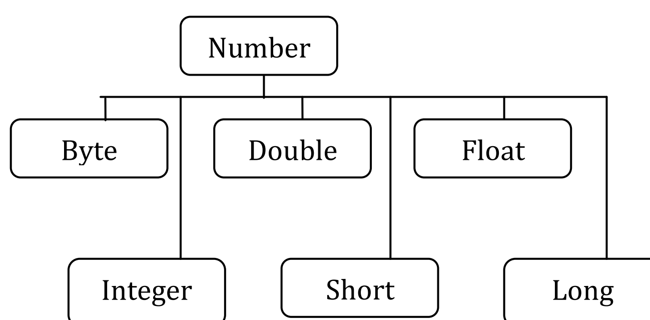
7	Long	long
8	Boolean	Boolean

4.2 Java.lang.Number Class in Java

In java, while working with numbers, we use primitive data types. But, Java also provides various numeric wrapper sub classes under the abstract class Number present in *java.lang* package. The Number class have mainly **six** sub-classes under it. All these sub-classes define some useful methods which are used regularly while dealing with numbers.

The various wrapper classes such as (Integer, Long, Byte, Double, Float, Short) are subclasses or derived classes of the abstract super class Number.

These all derived classes can "wrap" the primitive data type in its corresponding object. Often, the wrapping is done by the compiler. Incase if we use a primitive values at the place where an object is expected, the compiler boxes the primitive values in its wrapper class for you. Similarly, if we a Number object is used at place where a primitive value is expected, the compiler unboxes the object impleciately. This concept is also called Autoboxing and Unboxing.



4.2.1 Constructorof Number Class

Table 4.2 : Number class constructor

Sr. no.	Constructor Description
1	**Number()** This is the Single Constructor.

4.2.2 Methods of Number Class

Following is the list of the instance methods, that are implemented by all the subclasses of the Number class –

Table 4.3 : Number class methods

Sr.No.	Method & Description
1	**xxxValue():**It Converts the value of *this* Number object to the xxx data type and returns it.
2	**compareTo():**It is used to Compares *this* Number object to the argument.
3	**equals():**It willDetermines whether *this* number object is equal to the argument.
4	**valueOf():** It will Returns an Integer object holding the value of the specified primitive.
5	**toString():** Returns a String object representing the value of a specified int or Integer.
6	**parseInt():** This method is used to get the primitive data type of a certain String.
7	**abs():** This method willReturns the absolute value of the argument.
8	**ceil():** This methodReturns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	**floor():** Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	**rint():**Returns the integer that is closest in value to the argument. Returned as a double.
11	**round():** Returns the closest long or int, as indicated by the method's return type to the argument.
12	**min():** This method used toReturns the smaller of the two arguments.
13	**max():**It willReturns the larger of the two arguments.
14	**exp():** Returns the base of the natural logarithms, e, to the power of the argument.
15	**log():** This method Returns the natural logarithm of the argument.
16	**pow():** This method will Returns the value of the first argument raised to the power of the second argument.
17	**sqrt():**This method will Returns the square root of the argument.
18	**sin():**This method willReturns the sine of the specified double value.
19	**cos():**This method will Returns the cosine of the specified double value.
20	**tan():**This method used toReturns the tangent of the specified double value.

21	**asin():**This method will Returns the arcsine of the specified double value.
22	**acos():** This method is usedReturns the arccosine of the specified double value.
23	**atan():** Returns the arctangent of the specified double value.
24	**atan2():** Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
25	**toDegrees():** Converts the argument to degrees.
26	**toRadians():** Converts the argument to radians.
27	**random():** Returns a random number.

Example: Java program to demonstrate xxxValue() method

```
public class Test
{
   public static void main(String[] args)
   {
      // Creating a Double Class object with value "6.9685"
      Double d = new Double("6.9685");

      // Converting this Double(Number) object to
      // different primitive data types
      byte b = d.byteValue();
      short s = d.shortValue();
      int i = d.intValue();
      long l = d.longValue();
      float f = d.floatValue();
      double d1 = d.doubleValue();

      System.out.println("value of d after converting it to byte : " + b);
      System.out.println("value of d after converting it to short : " + s);
      System.out.println("value of d after converting it to int : " + i);
      System.out.println("value of d after converting it to long : " + l);
      System.out.println("value of d after converting it to float : " + f);
      System.out.println("value of d after converting it to double : " + d1);
   }
}
```

Output:

```
value of d after converting it to byte : 6
value of d after converting it to short : 6
value of d after converting it to int : 6
value of d after converting it to long : 6
value of d after converting it to float : 6.9685
value of d after converting it to double : 6.9685
```

Example: Java program to demonstrate compareTo() method

```
public class Test
{
    public static void main(String[] args)
    {
        // creating an Integer Class object with value "10"
        Integer i = new Integer("10");

        // comparing value of i
        System.out.println(i.compareTo(7));
        System.out.println(i.compareTo(11));
        System.out.println(i.compareTo(10));
    }
}
```

```
Output:  1     -1        0
```

Example: Java program to demonstrate equals() method

```
public class Test
{
    public static void main(String[] args)
    {
        // creating a Short Class object with value "15"
        Short s = new Short("15");

        // creating a Short Class object with value "10"
        Short x = 10;

        // creating an Integer Class object with value "15"
        Integer y = 15;

        // creating another Short Class object with value "15"
        Short z = 15;

        //comparing s with other objects
        System.out.println(s.equals(x));
        System.out.println(s.equals(y));
        System.out.println(s.equals(z));
    }
}
```

```
Output:  false   false    true
```

4.3 Java Integer Class

The Java Integer class is a derived class of the **Java.lang.Number** class hence it comes under **Java.lang.Number**package. This class wraps a value of the primitive type int in an object. An object of Integer class contains a single field of type int value.

4.3.1 Constructor of Integer Class

Sr. no.	Constructor Description
1	**Integer(int b):** Creates a Integer object initialized with the value provided.
2	**Integer(String s):** Creates a Integer object initialized with the int value provided by string representation. Defalut radix is taken to be 10

4.3.2 Methods of Integer Class

Table 4.4 : Integer class methods

Sr.No.	Method & Description
1	**toString () :** Returns the string corresponding to the int value.
2	**valueOf() :** returns the Integer object initialised with the value provided.
3	**valueOf(String val,int radix):** Another overloaded function which provides function similar to new Integer(Integer.parseInteger(val,radix))
4	**valueOf(String val)** Another overloaded function which provides function similar to new Integer(Integer.parseInt(val,10))
5	**getInteger() :** returns the Integer object representing the value associated with the given system property or null if it does not exist.
6	**decode () :** returns a Integer object holding the decoded value of string provided.
7	**rotateLeft() :** Returns a primitive int by rotating the bits left by given distance in two's complement form of the value given.
8	**rotateRight() :** Returns a primitive int by rotating the bits right by given distance in the twos complement form of the value given.
9	**byteValue() :** returns a byte value corresponding to this Integer Object
10	**shortValue() :** returns a short value corresponding to this Integer Object.
11	**floatValue() :** returns a float value corresponding to this Integer Object.
12	**intValue() :** returns a value corresponding to this Integer Object.
13	**doubleValue() :** returns a double value corresponding to this Integer Object.
14	**hashCode() :** returns the hashcode corresponding to this Integer Object.

15	**bitcount() :** Returns number of set bits in twos complement of the integer given.
16	**equals() :** Used to compare the equality of two Integer objects
17	**compareTo() :** Used to compare two Integer objects for numerical equality.
18	**compare() :** Used to compare two primitive int values for numerical equality
19	**reverse() :** returns a primitive int value reversing the order of bits in two's complement form of the given int value.
20	**static int max(int a, int b)**: This method returns the greater of two int values as if by calling Math.max.
21	**static int min(int a, int b)**: This method returns the smaller of two int values as if by calling Math.min.

Example: Java program to illustrate various Integer class methods

```
public class Integer_test
{
   public static void main(String args[])
   {
      int b = 55;
      String bb = "45";

      // Construct two Integer objects
      Integer x = new Integer(b);
      Integer y = new Integer(bb);

      // xxxValue can be used to retrieve
      // xxx type value from int value.
      // xxx can be int,byte,short,long,double,float
      System.out.println("bytevalue(x) = " + x.byteValue());
      System.out.println("shortvalue(x) = " + x.shortValue());
      System.out.println("intvalue(x) = " + x.intValue());
      System.out.println("longvalue(x) = " + x.longValue());
      System.out.println("doublevalue(x) = " + x.doubleValue());
      System.out.println("floatvalue(x) = " + x.floatValue());

      int value = 45;

      // bitcount() : can be used to count set bits
      // in twos complement form of the number
      System.out.println("Integer.bitcount(value)=" + Integer.bitCount(value));

      // numberOfTrailingZeroes and numberOfLeaadingZeroes
      // can be used to count prefix and postfix sequence of 0
      System.out.println("Integer.numberOfTrailingZeros(value)=" +
                     Integer.numberOfTrailingZeros(value));
      System.out.println("Integer.numberOfLeadingZeros(value)=" +
                     Integer.numberOfLeadingZeros(value));

      //highestOneBit returns a value with one on highest
      //set bit position
```

```
        System.out.println("Integer.highestOneBit(value)=" +
                            Integer.highestOneBit(value));

        // highestOneBit returns a value with one on lowest
        // set bit position
        System.out.println("Integer.lowestOneBit(value)=" +
                            Integer.lowestOneBit(value));

        // reverse() can be used to reverse order of bits
        // reverseytes() can be used to reverse order of bytes
        System.out.println("Integer.reverse(value)=" +
                            Integer.reverse(value));
        System.out.println("Integer.reverseBytes(value)=" +
                            Integer.reverseBytes(value));

        // signum() returns -1,0,1 for negative,0 and positive
        // values
        System.out.println("Integer.signum(value)=" + Integer.signum(value));

        // hashcode() returns hashcode of the object
        int hash = x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        // equals returns boolean value representing equality
        boolean eq = x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        // compare() used for comparing two int values
        int e = Integer.compare(x, y);
        System.out.println("compare(x,y) = " + e);

        // compareTo() used for comparing this value with some
        // other value
        int f = x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);
    }
}
```

Output :

```
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.0
floatvalue(x) = 55.0
Integer.bitcount(value)=4
Integer.numberOfTrailingZeros(value)=0
Integer.numberOfLeadingZeros(value)=26
Integer.highestOneBit(value)=32
Integer.lowestOneBit(value)=1
```

```
Integer.reverse(value)=-1275068416
Integer.reverseBytes(value)=754974720
Integer.signum(value)=1
hashcode(x) = 55
x.equals(y) = false
compare(x,y) = 1
x.compareTo(y) = 1
Java.Lang.Float class in Java
```

4.4 Java Float Class

Float class is a wrapper class for the primitive type float which contains several methods to effectively deal with a float value like converting it to a string representation, and vice-versa.

4.4.1 Constructor of Float Class

Table 4.5 : Float class constructor

Sr. no.	Constructor Description
1	**Float(float b):** Creates a Float object initialized with the value provided.
2	**Float(String s):** Creates a Float object initialized with the parsed float value provided by string representation. Defalut radix is taken to be 10.

4.4.2 Methods of Float Class

Table 4.6 : Float class methods

Sr.No.	Method & Description
1	**toString ():** Returns the string corresponding to the float value.
2	**valueOf() :** returns the Float object initialised with the value provided.

3	**parseFloat() :** returns float value by parsing the string. Differs from valueOf() as it returns a primitive float value and valueOf() return Float object.
4	**byteValue() :** returns a byte value corresponding to this Float Object.
5	**shortValue() :** returns a short value corresponding to this Float Object.
6	**intValue() :** returns a int value corresponding to this Float Object.
7	**longValue() :** returns a long value corresponding to this Float Object.
8	**doubleValue() :** returns a double value corresponding to this Float Object.
9	**floatValue() :** returns a float value corresponding to this Float Object.
10	**hashCode() :** returns the hashcode corresponding to this Float Object.
11	**isNaN() :** returns true if the float object in consideration is not a number, otherwise false.
12	**isInfinite() :** returns true if the float object in consideration is very large, otherwise false
13	**equals() :** Used to compare the equality of two Float objects.
14	**compareTo() :** Used to compare two Float objects for numerical equality.
15	**compare() :** Used to compare two primitive float values for numerical equality.
16	**toHexString() :** Returns the hexadecimal representation of the argument float value.
17	**IntBitsToFloat() :** Returns the float value corresponding to the long bit pattern of the argument. It does reverse work of the previous two methods.

Example: Java program to illustrate various float class methods of Java.lang class

```
public class GfG
{

   public static void main(String[] args)
   {
      float b = 55.05F;
      String bb = "45";

      // Construct two Float objects
      Float x = new Float(b);
      Float y = new Float(bb);

      // toString()
      System.out.println("toString(b) = " + Float.toString(b));

      // valueOf()
      // return Float object
      Float z = Float.valueOf(b);
```

```
        System.out.println("valueOf(b) = " + z);
        z = Float.valueOf(bb);
        System.out.println("ValueOf(bb) = " + z);

        // parseFloat()
        // return primitive float value
        float zz = Float.parseFloat(bb);
        System.out.println("parseFloat(bb) = " + zz);

        System.out.println("bytevalue(x) = " + x.byteValue());
        System.out.println("shortvalue(x) = " + x.shortValue());
        System.out.println("intvalue(x) = " + x.intValue());
        System.out.println("longvalue(x) = " + x.longValue());
        System.out.println("doublevalue(x) = " + x.doubleValue());
        System.out.println("floatvalue(x) = " + x.floatValue());

        int hash = x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        boolean eq = x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        int e = Float.compare(x, y);
        System.out.println("compare(x,y) = " + e);

        int f = x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);

        Float d = Float.valueOf("1010.54789654123654");
        System.out.println("isNaN(d) = " + d.isNaN());

        System.out.println("Float.isNaN(45.12452) = "
                        + Float.isNaN(45.12452F));

        // Float.POSITIVE_INFINITY stores
        // the positive infinite value
        d = Float.valueOf(Float.POSITIVE_INFINITY + 1);
        System.out.println("Float.isInfinite(d) = "
                    + Float.isInfinite(d.floatValue()));

        float dd = 10245.21452F;
        System.out.println("Float.toString(dd) = "
                          + Float.toHexString(dd));

        int float_to_int = Float.floatToIntBits(dd);
        System.out.println("Float.floatToLongBits(dd) = "
                              + float_to_int);

        float int_to_float = Float.intBitsToFloat(float_to_int);
        System.out.println("Float.intBitsToFloat(float_to_long) = "
                              + int_to_float);
    } }
```

Output :

```
toString(b) = 55.05
valueOf(b) = 55.05
ValueOf(bb) = 45.0
parseFloat(bb) = 45.0
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.04999923706055
floatvalue(x) = 55.05
hashcode(x) = 1113338675
x.equals(y) = false
compare(x,y) = 1
x.compareTo(y) = 1
isNaN(d) = false
Float.isNaN(45.12452) = false
Float.isInfinite(d) = true
Float.toString(dd) = 0x1.4029b8p13
Float.floatToLongBits(dd) = 1176507612
Float.intBitsToFloat(float_to_long) = 10245.215
```

4.5 Java.Lang.Byte class in Java

Byte class is a wrapper class for the primitive type byte which contains several methods to effectively deal with a byte value like converting it to a string representation, and vice-versa.

4.5.1 Constructor of Byte Class

Table 4.7: Byte class constructor

Sr. no.	Constructor Description
1	**Byte(byte b):** Creates a Byte object initialized with the value provided
2	**Byte(String s):** Creates a Byte object initialized with the byte value provided by string representation. Defalut radix is taken to be 10.

4.5.2 Methods of Byte Class

Table 4.8 : Byte class methods

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the byte value.
2	**valueOf() :** returns the Byte object initialised with the value provided.
3	**parseByte() :** returns byte value by parsing the string.
4	**decode() :** returns a Byte object holding the decoded value of string provided.
5	**byteValue() :** returns a byte value corresponding to this Byte Object.
6	**shortValue() :** returns a short value corresponding to this Byte Object.
7	**intValue() :** returns a int value corresponding to this Byte Object.
8	**longValue() :** returns a long value corresponding to this Byte Object.
9	**doubleValue() :** returns a double value corresponding to this Byte Object.
10	**floatValue() :** returns a float value corresponding to this Byte Object.
11	**hashCode() :** returns the hashcode corresponding to this Byte Object.
12	**equals() :** Used to compare the equality of two Byte objects.
13	**compareTo() :** Used to compare two Byte objects for numerical equality.
14	**compare() :** Used to compare two primitive byte values for numerical equality.

Example: Java program to illustrate various methods of Byte class

```
public class Byte_test
{
   public static void main(String[] args)
   {
      byte b = 55;
      String bb = "45";

      // Construct two Byte objects
```

```
Byte x = new Byte(b);
Byte y = new Byte(bb);

// toString()
System.out.println("toString(b) = " + Byte.toString(b));

// valueOf()
// return Byte object
Byte z = Byte.valueOf(b);
System.out.println("valueOf(b) = " + z);
z = Byte.valueOf(bb);
System.out.println("ValueOf(bb) = " + z);
z = Byte.valueOf(bb, 6);
System.out.println("ValueOf(bb,6) = " + z);

// parseByte()
// return primitive byte value
byte zz = Byte.parseByte(bb);
System.out.println("parseByte(bb) = " + zz);
zz = Byte.parseByte(bb, 6);
System.out.println("parseByte(bb,6) = " + zz);

//decode()
String decimal = "45";
String octal = "005";
String hex = "0x0f";

Byte dec=Byte.decode(decimal);
System.out.println("decode(45) = " + dec);
dec=Byte.decode(octal);
System.out.println("decode(005) = " + dec);
dec=Byte.decode(hex);
System.out.println("decode(0x0f) = " + dec);

System.out.println("bytevalue(x) = " + x.byteValue());
System.out.println("shortvalue(x) = " + x.shortValue());
System.out.println("intvalue(x) = " + x.intValue());
System.out.println("longvalue(x) = " + x.longValue());
```

```
        System.out.println("doublevalue(x) = " + x.doubleValue());
        System.out.println("floatvalue(x) = " + x.floatValue());

        int hash=x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        boolean eq=x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        int e=Byte.compare(x, y);
        System.out.println("compare(x,y) = " + e);

        int f=x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);
    }
}
```

Output:

```
toString(b) = 55
valueOf(b) = 55
ValueOf(bb) = 45
ValueOf(bb,6) = 29
parseByte(bb) = 45
parseByte(bb,6) = 29
decode(45) = 45
decode(005) = 5
decode(0x0f) = 15
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.0
floatvalue(x) = 55.0
hashcode(x) = 55
x.equals(y) = false
compare(x,y) = 10
x.compareTo(y) = 10
```

4.6 Java.Lang.Short class in Java

Short class is a wrapper class for the primitive type short which contains several methods to effectively deal with a short value like converting it to a string representation, and vice-versa.

4.6.1 constructors of Short Class

Table 4.9 : Short class constructor

Sr.No.	Constructor & Description
1	**Short(short value):** This constructs a newly allocated Short object that represents the specified short value.
2	**Short(String s):** This constructs a newly allocated Short object that represents the short value indicated by the String parameter.

4.6.2 Methods of Short Class

Table 4.10 : Short class methods

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Short value.
2	**valueOf() :** returns the Short object initialised with the value provided.
3	**parseShort() :** returns short value by parsing the string.
4	**decode() :** returns a Short object holding the decoded value of string provided.
5	**byteValue() :** returns a byte value corresponding to this short Object.
6	**shortValue() :** returns a short value corresponding to this Short Object.
7	**intValue() :** returns a int value corresponding to this Short Object.
8	**longValue() :** returns a long value corresponding to this Short Object.
9	**doubleValue() :** returns a double value corresponding to this Short Object.
10	**floatValue() :** returns a float value corresponding to this Short Object.
11	**hashCode() :** returns the hash code corresponding to this Short Object.

12	**equals() :** Used to compare the equality of two Short objects.
13	**compareTo() :** Used to compare two Short objects for numerical equality.
14	**compare() :** Used to compare two primitive short values for numerical equality.

Example: Java program to illustrate various methods of Short class

```
public class Short_test
{

    public static void main(String[] args)
    {

        short b = 55;
        String bb = "45";

        // Construct two Short objects
        Short x = new Short(b);
        Short y = new Short(bb);

        // toString()
        System.out.println("toString(b) = " + Short.toString(b));

        // valueOf()
        // return Short object
        Short z = Short.valueOf(b);
        System.out.println("valueOf(b) = " + z);
        z = Short.valueOf(bb);
        System.out.println("ValueOf(bb) = " + z);
        z = Short.valueOf(bb, 6);
        System.out.println("ValueOf(bb,6) = " + z);

        // parseShort()
        // return primitive short value
        short zz = Short.parseShort(bb);
        System.out.println("parseShort(bb) = " + zz);
        zz = Short.parseShort(bb, 6);
        System.out.println("parseShort(bb,6) = " + zz);

        //decode()
        String decimal = "45";
        String octal = "005";
        String hex = "0x0f";

        Short dec = Short.decode(decimal);
        System.out.println("decode(45) = " + dec);
        dec = Short.decode(octal);
        System.out.println("decode(005) = " + dec);
        dec = Short.decode(hex);
        System.out.println("decode(0x0f) = " + dec);
```

```
        System.out.println("bytevalue(x) = " + x.byteValue());
        System.out.println("shortvalue(x) = " + x.shortValue());
        System.out.println("intvalue(x) = " + x.intValue());
        System.out.println("longvalue(x) = " + x.longValue());
        System.out.println("doublevalue(x) = " + x.doubleValue());
        System.out.println("floatvalue(x) = " + x.floatValue());

        int hash = x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        boolean eq = x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        int e = Short.compare(x, y);
        System.out.println("compare(x,y) = " + e);

        int f = x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);


        short to_rev = 45;
        System.out.println("Short.reverseBytes(to_rev) = " + Short.reverseBytes(to_rev));
    }
}
```

Output :

```
toString(b) = 55
valueOf(b) = 55
ValueOf(bb) = 45
ValueOf(bb,6) = 29
parseShort(bb) = 45
parseShort(bb,6) = 29
decode(45) = 45
decode(005) = 5
decode(0x0f) = 15
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.0
floatvalue(x) = 55.0
hashcode(x) = 55
x.equals(y) = false
compare(x,y) = 10
```

```
x.compareTo(y) = 10
Short.reverseBytes(to_rev) = 11520
```

4.7 Java.Lang.Long class in Java

Long class is a wrapper class for the primitive type long which contains several methods to effectively deal with a long value like converting it to a string representation, and vice-versa.

4.7.1 Long Class constructors

Table 4.11 : Long class constructor

Sr.No.	Constructor & Description
1	**Long(long b):** Creates a Long object initialized with the value provided.
2	**Long(String s):** Creates a Long object initialized with the long value provided by string representation. Defalut radix is taken to be 10.

4.7.2 Methods of Long Class

Table 4.12 : Long class methods

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Long value.
2	**valueOf() :** returns the Long object initialised with the value provided.
3	**parse Long () :** returns Long value by parsing the string.
4	**decode() :** returns a Long object holding the decoded value of string provided.
5	**byteValue() :** returns a byte value corresponding to this Long Object.
6	**shortValue() :** returns a short value corresponding to this Long Object.
7	**intValue() :** returns a int value corresponding to this Long Object.
8	**longValue() :** returns a long value corresponding to this Long Object.
9	**doubleValue() :** returns a double value corresponding to this Long Object.
10	**floatValue() :** returns a float value corresponding to this Long Object.
11	**hashCode() :** returns the hash code corresponding to this Long Object.

12	**equals() :** Used to compare the equality of two Long objects.
13	**compareTo()** : Used to compare two Long objects for numerical equality.
14	**compare() :** Used to compare two primitive long values for numerical equality.
15	**bitcount() :** Returns number of set bits in twos complement of the long given.

Example: Java program to illustrate various Long methods

```
public class Long_test
{
   public static void main(String args[])
   {
      long b = 55;
      String bb = "45";

      // Construct two Long objects
      Long x = new Long(b);
      Long y = new Long(bb);

      // xxxValue can be used to retrieve
      // xxx type value from long value.
      // xxx can be int,byte,short,long,double,float
      System.out.println("bytevalue(x) = " + x.byteValue());
      System.out.println("shortvalue(x) = " + x.shortValue());
      System.out.println("intvalue(x) = " + x.intValue());
      System.out.println("longvalue(x) = " + x.longValue());
      System.out.println("doublevalue(x) = " + x.doubleValue());
      System.out.println("floatvalue(x) = " + x.floatValue());

      long value = 45;

      // bitcount() : can be used to count set bits in twos complement form of the number
      System.out.println("Long.bitcount(value)=" + Long.bitCount(value));

      // numberOfTrailingZeroes and numberOfLeaadingZeroes
      // can be used to count prefix and postfix sequence of 0
      System.out.println("Long.numberOfTrailingZeros(value)=" +
                    Long.numberOfTrailingZeros(value));
      System.out.println("Long.numberOfLeadingZeros(value)=" +
                    Long.numberOfLeadingZeros(value));

      // highestOneBit returns a value with one on highest
      // set bit position
      System.out.println("Long.highestOneBit(value)=" +
                        Long.highestOneBit(value));

      // highestOneBit returns a value with one on lowest
      // set bit position
      System.out.println("Long.lowestOneBit(value)=" +
                        Long.lowestOneBit(value));
```

```
        // reverse() can be used to reverse order of bits
        // reverseytes() can be used to reverse order of bytes
        System.out.println("Long.reverse(value)=" + Long.reverse(value));
        System.out.println("Long.reverseBytes(value)=" +
                            Long.reverseBytes(value));

        // signum() returns -1,0,1 for negative,0 and positive
        // values
        System.out.println("Long.signum(value)=" + Long.signum(value));

        // hashcode() returns hashcode of the object
        int hash = x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        // equals returns boolean value representing equality
        boolean eq = x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        // compare() used for comparing two int values
        int e = Long.compare(x, y);
        System.out.println("compare(x,y) = " + e);

        // compareTo() used for comparing this value with some
        // other value
        int f = x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);
    }
}
```

Output :

```
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.0
floatvalue(x) = 55.0
Long.bitcount(value)=4
Long.numberOfTrailingZeros(value)=0
Long.numberOfLeadingZeros(value)=58
Long.highestOneBit(value)=32
Long.lowestOneBit(value)=1
Long.reverse(value)=-5476377146882523136
Long.reverseBytes(value)=3242591731706757120
Long.signum(value)=1
hashcode(x) = 55
x.equals(y) = false
compare(x,y) = 1
```

x.compareTo(y) = 1

4.8 Java.Lang.Double class in Java

Double class is a wrapper class for the primitive type double which contains several methods to effectively deal with a double value like converting it to a string representation, and vice-versa.

4.8.1 Constructors of Double Class

Table 4.13 : Double class constructor

Sr.No.	Constructor & Description
1	**Double(double b) :**Creates a Double object initialized with the value provided.
2	**Double(String s) :**Creates a Double object initialized with the parsed double value provided by string representation. Defalut radix is taken to be 10.

4.8.2 Methods of Double Class

Table 4.14 : Double class methods

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Double value.
2	**valueOf() :** returns the Double object initialised with the value provided.
3	**parse**Double() **:** returns Double value by parsing the string.
4	**decode() :** returns a Double object holding the decoded value of string provided.
5	**byteValue() :** returns a byte value corresponding to this Double Object.
6	**shortValue() :** returns a short value corresponding to this Double Object.
7	**intValue() :** returns a int value corresponding to this Double Object.
8	**longValue() :** returns a long value corresponding to this Double Object.
9	**doubleValue() :** returns a double value corresponding to this Double Object.
10	**floatValue() :** returns a float value corresponding to this Double Object.
11	**hashCode() :** returns the hash code corresponding to this Double Object.

12	**equals() :** Used to compare the equality of two Double objects.
13	**compareTo() :** Used to compare two Double objects for numerical equality.
14	**compare() :** Used to compare two primitive double values for numerical equality.
15	**toHexString() :** Returns the hexadecimal representation of the argument double value.

Example: Java program to illustrate various Double class methods of java.lang class

```
public class Double_test
{

    public static void main(String[] args)
    {

        double b = 55.05;
        String bb = "45";

        // Construct two Double objects
        Double x = new Double(b);
        Double y = new Double(bb);

        // toString()
        System.out.println("toString(b) = " + Double.toString(b));

        // valueOf()
        // return Double object
        Double z = Double.valueOf(b);
        System.out.println("valueOf(b) = " + z);
        z = Double.valueOf(bb);
        System.out.println("ValueOf(bb) = " + z);

        // parseDouble()
        // return primitive double value
        double zz = Double.parseDouble(bb);
        System.out.println("parseDouble(bb) = " + zz);

        System.out.println("bytevalue(x) = " + x.byteValue());
        System.out.println("shortvalue(x) = " + x.shortValue());
        System.out.println("intvalue(x) = " + x.intValue());
        System.out.println("longvalue(x) = " + x.longValue());
        System.out.println("doublevalue(x) = " + x.doubleValue());
        System.out.println("floatvalue(x) = " + x.floatValue());

        int hash = x.hashCode();
        System.out.println("hashcode(x) = " + hash);

        boolean eq = x.equals(y);
        System.out.println("x.equals(y) = " + eq);

        int e = Double.compare(x, y);
        System.out.println("compare(x,y) = " + e);
```

```
        int f = x.compareTo(y);
        System.out.println("x.compareTo(y) = " + f);

        Double d = Double.valueOf("1010.54789654123654");
        System.out.println("isNaN(d) = " + d.isNaN());

        System.out.println("Double.isNaN(45.12452) = " + Double.isNaN(45.12452));

        // Double.POSITIVE_INFINITY stores
        // the positive infinite value
        d = Double.valueOf(Double.POSITIVE_INFINITY + 1);
        System.out.println("Double.isInfinite(d) = " +
                                Double.isInfinite(d.doubleValue()));

        double dd = 10245.21452;
        System.out.println("Double.toString(dd) = " + Double.toHexString(dd));

        long double_to_long = Double.doubleToLongBits(dd);
        System.out.println("Double.doubleToLongBits(dd) = " + double_to_long);

        double long_to_double = Double.longBitsToDouble(double_to_long);
        System.out.println("Double.LongBitsToDouble(double_to_long) = " +
                                long_to_double);
    }

}
```

Output :

```
toString(b) = 55.05
valueOf(b) = 55.05
ValueOf(bb) = 45.0
parseDouble(bb) = 45.0
bytevalue(x) = 55
shortvalue(x) = 55
intvalue(x) = 55
longvalue(x) = 55
doublevalue(x) = 55.05
floatvalue(x) = 55.05
hashcode(x) = 640540672
x.equals(y) = false
compare(x,y) = 1
x.compareTo(y) = 1
isNaN(d) = false
Double.isNaN(45.12452) = false
Double.isInfinite(d) = true
```

```
Double.toString(dd) = 0x1.4029b7564302bp13
Double.doubleToLongBits(dd) = 4666857980575363115
Double.LongBitsToDouble(double_to_long) = 10245.21452
```

4.9 Java.lang.Character Class in Java

Java provides a wrapper class **Character** in java.lang package. An object of type Character contains a single field, whose type is char.

4.9.1 Constructor of Character Class

Table 4.15 : Character class constructor

Sr.No.	Constructor & Description
1	**Character('a'):** statement creates a Character object which contain ‘a’ of type char. This is only the constructor in Character class which expect an argument of char data type**.**

4.9.2 Methods of Character Class

Table 4.16 : Character class methods

Sr.No.	Method & Description
1	**toString(char ch)** : This method returns a String class object representing the specified character value(ch)
2	**char toLowerCase(char ch)** : It returns the lowercase of the specified char value(ch).
3	**char toUpperCase(char ch)** : This method returns the uppercase of the specified char value(ch).
4	**boolean isLowerCase(char ch)** : It determines whether the specified char value(ch) is lowercase or not.
5	**boolean isUpperCase(char ch)** : This method determines whether the specified char value(ch) is uppercase or not.
6	**char charValue()**: This method returns the value of this Character object.
7	**static int compare(char x, char y)**: This method compares two char values numerically.
8	**int compareTo(Character anotherCharacter)**: This method compares two Character objects numerically.
9	**static int digit(char ch, int radix)**: This method returns the numeric value of the character ch in the specified radix.
10	**boolean equals(Object obj)**: This method compares this object against the specified object.

Example: Java program to demonstrate isLetter() method

```
publicclassTest
{
   publicstaticvoidmain(String[] args)
   {
      System.out.println(Character.isLetter('A'));
      System.out.println(Character.isLetter('0'));
    }
}
```

```
Output: true    false
```

Example: Java program to demonstrate isUpperCase() method

```
publicclassTest
{
 publicstaticvoidmain(String[] args)
 {
   System.out.println(Character.isUpperCase('A'));
   System.out.println(Character.isUpperCase('a'));
   System.out.println(Character.isUpperCase(65));
 }
}
```

```
Output:   true              false                         true
```

Example: Java program to demonstrate toLowerCase() method

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println(Character.toLowerCase('A'));
    System.out.println(Character.toLowerCase(65));
    System.out.println(Character.toLowerCase(48));
  }
}
```

Output:	a	97	48

4.10 Type conversion / Type Casting

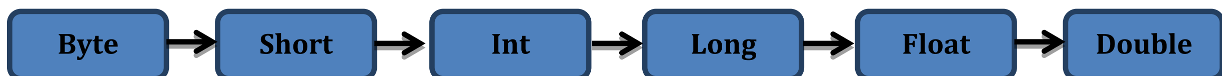
In java we can assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible with each other, then Java will perform the impleciate conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. Let us understand with an example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening is type of conversion which takes place when two data types are automatically converted or get converted impliciately. This happens when:

1. The two data types are compatible.
2. When we assign value of a smaller data type to a bigger data type.

Let us understand with an example, in java the int datatype is a smaller insize can be covrted in to large size long data type automatically,and both are compatible with each other.



Example:

```
class Test
{
   public static void main(String[] args)
   {
      int i = 100;
         // automatic type conversion
      long l = i;
         // automatic type conversion
      float f = l;
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
```

```
        System.out.println("Float value "+f);
    }
}
```

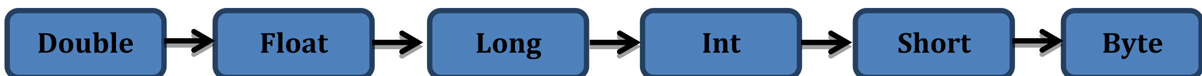
```
Output:

        Int value 100
        Long value 100
        Float value 100.0
```

4.11 Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type which does not get converted automatically or not compatible with each other, we have to perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.



Let us understand with example, numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, and are not compatible with each other.

Example:

```
/Java program to illustrate explicit type conversion
class Test
{
   public static void main(String[] args)
   {
      double d = 100.04;

      //explicit type casting
      long l = (long)d;

      //explicit type casting
      int i = (int)l;
      System.out.println("Double value "+d);

      //fractional part lost
      System.out.println("Long value "+l);

      //fractional part lost
      System.out.println("Int value "+i);
   }
}
```

```
Output:

        Double value 100.04
        Long value 100
        Int value 100
```

4.12 Autoboxing & Unboxing:

Autoboxing**:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

Example: Java program to convert primitive into objects Auto boxing example of int to Integer

```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}}
```

```
Output:20 20 20
```

Unboxing: Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing, in simple words we can say It is just the reverse process of autoboxing.Let us consider with an example – conversion of Integer to int, Long to long, Double to double etc.

Example: Java program to convert object into primitives Unboxing example of Integer to int

```
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

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

```
Output:     3 3 3
```

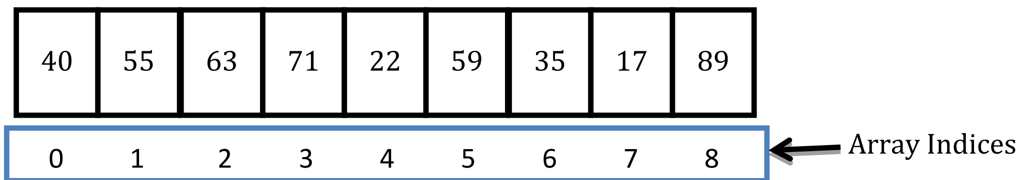
4.13 Arrays in Java

An array is a linear data structure which is a collection of similar type but multiple elements which have a contiguous memory location.

Java array is an object which contains elements of a similar data type. The elements of an array are stored in a contiguous also called as sequential memory location. It is a data structure where we store similar types of elements. In java array allows to store only a fixed set of elements.

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The Object is a superclass of an array type.

Array can contain primitive data types as well as objects of a class depending on the definition of array. For primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.



Array Length: 9
First Index: 0
Last Index: 8

4.13.1 Advantages

- **Code Optimization:** It makes the code optimized, It is usefull to retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

4.13.2 Disadvantages

- **Size Limit:** array allows storing only the fixed size of elements in the array. It doesn't grow its size at runtime. This problem can be solved by using the collection framework, which grows automatically.

4.13.3 Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

4.13.3.1 Single Dimensional Array in Java

Syntax:

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

```
Output:

10      20      70      40      50
```

4.13.3.2 Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

```
Output:
        1 2 3
        2 4 5
        4 4 5
```

4.13.4 Array Of Objects In Java

The array of objects, as defined by its name, stores an array of objects. An object represents a single record in memory, and thus for multiple records, an array of objects must be created. It

must be noted, that the arrays can hold only references to the objects, and not the objects themselves.

4.13.4.1 Declaring an Array Of Objects In Java

Array of Object can be declared by useing the class name Object, followed by square brackets to declare an Array of Objects.

Object[] JavaObjectArray;

Another declaration can be as follows:

4.13.4.2 Declaring an Array Objects With Initial Values

Java also allows Declaration of an array of object by adding initial values. Here, we create an array with initial string values as “Hello World”, as well as an integer with the value 5.

```
public class Main {

public static void main(String[] args) {

Object[] ObjectArray = {"Hello World", new Integer(5)};

System.out.println( ObjectArray[0] );

System.out.println( ObjectArray[1] );

}

}

Output:

Hello World

5
```

Outcomes :

- Students will able to study the function of wrapper classes, types of wrapper classes like integer, float, double, character, short, etc. in java.
- Students will able to study the function of constructors and methods of wrapper classes.
- Students will able to study multidimensional array in java.
- Students will able to study the working of array of objects in java .

Questions:

1. Write a brief note on wrapper class in java.
2. Describe in short the need of wrapper class in java.
3. Discuss the use of number class with some of important methods of number class.
4. Write a short note on use of Integer class with example and some of important methods of Integer class.
5. Explain with neat diagram hierarchy of number class.
6. Write a short note on use of Float class with some of important methods of Float class.
7. Discuss the use of Short class with example and some of important methods of Short class.
8. Discuss the use of Byte with example and some of important methods of Byte class.
9. Write a short note on use of Long class with example and some of important methods of Long class.
10. Discuss the use of Double class with example and some of important methods of Double class.
11. Write a short note on use of Character class with example and some of important methods of Character class.
12. With brief note explain the concept of type casting/Conversion.
13. List and explain with example the various types of conversion.
14. Explain the concept of boxing and auto boxing in java.
15. What is array? Explain need of array with advantages and disadvantages.
16. Explain with example types of array in java.

CHAPTER 5

STRING AND EXCEPTIONS HANDLING

Objectives:

- To study java string, string class, stringBuffer classes, their constructors and methods.
- To study the immutable string in java.
- To study exception handling, exception methods in java.

5.1 Java String

String in java is special type of object that represents sequence of charecter values. The String in java is work similar to aarays of characters.

Strings Objects in java are strongly supported by a char array. Since arrays are immutable that means cannot grow, hence Strings are alsoimmutable In **Java** programming language, **strings** are treated as objects. The **Java** offers the inbuilt **String** class to create and manipulate **string objects**. With every change in a string object causes to create a entire new string. Every string in java is end with “\0” symbol which is treated as last character or end of string. This character is placed at end of every string by default by compiler.

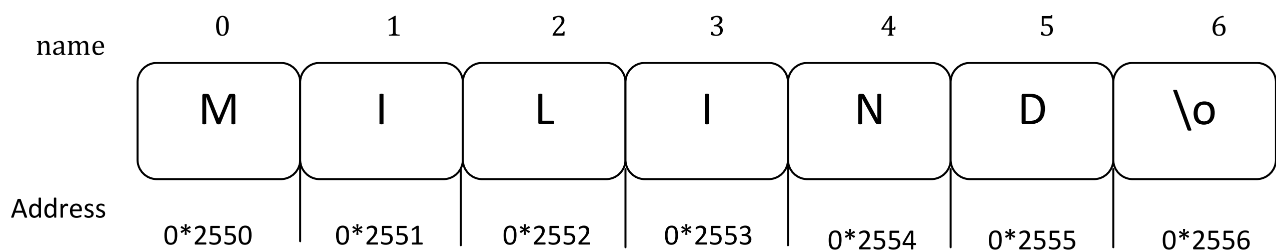


Fig. 5.1 : Representation of string

In above diagram you can see the representation of string along with indexes and memory address and as mentioned it ends with “\0”.

Example 1:

```
        char[] ch={'j','a','v','a','w','o','r','l','d'};
                String s=new String(ch);

is same as:
        String s="javaworld";
```

Example 2:

```
public class StringDemo{
public static void main(String args[]){
String s1="Hello";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("Demo");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

```
Output:HellostringsDemo
```

5.1.1 String Methods

Here is the list of methods supported by String class –

Table 5.1 : String class methods

Sr.No.	Method & Description
1	**char charAt(int index)**: Returns the character at the specified index.
2	**int compareTo(Object o)**: Compares this String to another Object.
3	**intcompareTo(String anotherString)**: Compares two strings lexicographically.
4	**int compareToIgnoreCase(String str)**: Compares two strings lexicographically, ignoring case differences.
5	**String concat(String str)**: Concatenates the specified string to the end of this string.
6	**boolean contentEquals(StringBuffer sb)**: Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.

7	**static String copyValueOf(char[] data)**: Returns a String that represents the character sequence in the array specified.
8	**static String copyValueOf(char[] data, int offset, int count)**: Returns a String that represents the character sequence in the array specified.
9	**boolean endsWith(String suffix)**: Tests if this string ends with the specified suffix.
10	**boolean equals(Object anObject)**: Compares this string to the specified object.
11	**boolean equalsIgnoreCase(String anotherString)**: Compares this String to another String, ignoring case considerations.
12	**byte getBytes()**: Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	**byte[] getBytes(String charsetName)**: Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**: Copies characters from this string into the destination character array.
15	**int hashCode()**: Returns a hash code for this string.
16	**int indexOf(int ch)**: Returns the index within this string of the first occurrence of the specified character.
17	**int indexOf(int ch, int fromIndex)**: Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	**int indexOf(String str)**: Returns the index within this string of the first occurrence of the specified substring.
19	**int indexOf(String str, int fromIndex)**: Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	**String intern()**: Returns a canonical representation for the string object.
21	**int lastIndexOf(int ch)**: Returns the index within this string of the last occurrence of the specified character.
22	**int lastIndexOf(int ch, int fromIndex)**: Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	**int lastIndexOf(String str)**: Returns the index within this string of the rightmost occurrence of the specified substring.

24	**int lastIndexOf(String str, int fromIndex)**: Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	**int length()**: Returns the length of this string.
26	**boolean matches(String regex)**: Tells whether or not this string matches the given regular expression.
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** Tests if two string regions are equal.
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)**: Tests if two string regions are equal.
29	**String replace(char oldChar, char newChar)**: Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	**String replaceAll(String regex, String replacement**: Replaces each substring of this string that matches the given regular expression with the given replacement.
31	**String replaceFirst(String regex, String replacement)**: Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	**String[] split(String regex)**: Splits this string around matches of the given regular expression.
33	**String[] split(String regex, int limit)**: Splits this string around matches of the given regular expression.
34	**boolean startsWith(String prefix)**: Tests if this string starts with the specified prefix.
35	**boolean startsWith(String prefix, int toffset)**: Tests if this string starts with the specified prefix beginning a specified index.
36	**CharSequence subSequence(int beginIndex, int endIndex)**: Returns a new character sequence that is a subsequence of this sequence.
37	**String substring(int beginIndex)**: Returns a new string that is a substring of this string.
38	**String substring(int beginIndex, int endIndex)**: Returns a new string that is a substring of this string.
39	**char[] toCharArray()**: Converts this string to a new character array.
40	**String toLowerCase()**: Converts all of the characters in this String to lower case using the rules

	of the default locale.
41	**String toLowerCase(Locale locale)**: Converts all of the characters in this String to lower case using the rules of the given Locale.
42	**String toString()**: This object (which is already a string!) is itself returned.
43	**String toUpperCase()**: Converts all of the characters in this String to upper case using the rules of the default locale.
44	**String toUpperCase(Locale locale)**: Converts all of the characters in this String to upper case using the rules of the given Locale.
45	**String trim()**: Returns a copy of the string, with leading and trailing whitespace omitted.
46	**static String valueOf(primitive data type x)**: Returns the string representation of the passed data type argument.

5.2 StringBuffer and StringBuilder class

5.2.1 StringBuffer:

StringBuffer class provides much of the functionality of strings class, and is knows as a peer class of String. Unlike the String class StringBuffer is a mutable and allows the rewritable character sequences. Hence StringBuffer represents growable and writable character sequences, but opposite to that String represents fixed-length, immutable character sequences.

- A string buffer is simiar to String in functionalities, but can be modified.
- It offers particular sequence of characters, bnd content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- Every string buffer has a capacity.

5.2.1.1 StringBuffer Class constructors

Table 5.2 : StringBuffer class constructors

Sr.No.	Constructor & Description
1	**StringBuffer():** This constructs a string buffer with no characters in it and an initial capacity of 16 characters.

2	**StringBuffer(CharSequence seq):** This constructs a string buffer that contains the same characters as the specified CharSequence.
3	**StringBuffer(int capacity):** This constructs a string buffer with no characters in it and the specified initial capacity.
4	**StringBuffer(String str):** This constructs a string buffer initialized to the contents of the specified string.

Syntax:

```
StringBuffer s = new StringBuffer("Hello Friends");
```

5.2.1.2 StringBuffer Class methods

Table 5.3: StringBuffer class methods

Sr.No.	Method & Description
1	**StringBuffer append(boolean b):**This method appends the string representation of the boolean argument to the sequence
2	**StringBuffer append(char c):** This method appends the string representation of the char argument to this sequence.
3	**StringBuffer append(char[] str):** This method appends the string representation of the char array argument to this sequence.
4	**StringBuffer** append(char[] str, int offset, int len): This method appends the string representation of a subarray of the char array argument to this sequence.
5	**StringBuffer append(CharSequence s):** This method appends the specified CharSequence to this sequence.
6	**StringBuffer append(CharSequence s, int start, int end):** This method appends a subsequence of the specified CharSequence to this sequence.
7	**StringBuffer append(double d):** This method appends the string representation of the double argument to this sequence.
8	**StringBuffer append(float f):** This method appends the string representation of the float argument to this sequence.
9	**StringBuffer append(int i):** This method appends the string representation of the int argument to this sequence.
10	**StringBuffer append(long lng):** This method appends the string representation of the long argument to this sequence.
11	**StringBuffer append(Object obj):** This method appends the string representation of the Object argument.

12	**StringBuffer append(String str):** This method appends the specified string to this character sequence.
13	**StringBuffer append(StringBuffer sb):** This method appends the specified StringBuffer to this sequence.
14	**StringBuffer appendCodePoint(int codePoint):** This method appends the string representation of the codePoint argument to this sequence.
15	**int capacity():** This method returns the current capacity.
16	**char charAt(int index):** This method returns the char value in this sequence at the specified index.
17	**int codePointAt(int index):** This method returns the character (Unicode code point) at the specified index
18	**int codePointBefore(int index):** This method returns the character (Unicode code point) before the specified index
19	**int codePointCount(int beginIndex, int endIndex):** This method returns the number of Unicode code points in the specified text range of this sequence
20	**StringBuffer delete(int start, int end):** This method removes the characters in a substring of this sequence.
21	**StringBuffer deleteCharAt(int index):** This method removes the char at the specified position in this sequence
22	**void ensureCapacity(int minimumCapacity):** This method ensures that the capacity is at least equal to the specified minimum.
23	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin):** This method characters are copied from this sequence into the destination character array dst.
24	**int indexOf(String str):** This method returns the index within this string of the first occurrence of the specified substring.
25	**int indexOf(String str, int fromIndex):** This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
26	**StringBuffer insert(int offset, boolean b):** This method inserts the string representation of the boolean argument into this sequence.
27	**StringBuffer insert(int offset, char c):** This method inserts the string representation of the char argument into this sequence.
28	**StringBuffer insert(int offset, char[] str):** This method inserts the string representation of the char array argument into this sequence.
29	**StringBuffer insert(int index, char[] str, int offset, int len):** This method inserts the string representation of a subarray of the str array argument into this sequence.
30	**StringBuffer insert(int dstOffset, CharSequence s) :** This method inserts the specified

	CharSequence into this sequence.
31	**StringBuffer insert(int dstOffset, CharSequence s, int start, int end):** This method inserts a subsequence of the specified CharSequence into this sequence.
32	**StringBuffer insert(int offset, double d):** This method inserts the string representation of the double argument into this sequence.
33	**StringBuffer insert(int offset, float f):** This method inserts the string representation of the float argument into this sequence.
34	**StringBuffer insert(int offset, int i):** This method inserts the string representation of the second int argument into this sequence.
35	**StringBuffer insert(int offset, long l):** This method inserts the string representation of the long argument into this sequence.
36	**StringBuffer insert(int offset, Object obj):** This method inserts the string representation of the Object argument into this character sequence.
37	**StringBuffer insert(int offset, String str):** This method inserts the string into this character sequence.
38	**int lastIndexOf(String str):** This method returns the index within this string of the rightmost occurrence of the specified substring.
39	**int lastIndexOf(String str, int fromIndex):** This method returns the index within this string of the last occurrence of the specified substring.
40	**int length():** This method returns the length (character count).
41	**int offsetByCodePoints(int index, int codePointOffset):** This method returns the index within this sequence that is offset from the given index by codePointOffset code points.
42	**StringBuffer replace(int start, int end, String str):** This method replaces the characters in a substring of this sequence with characters in the specified String.
43	**StringBuffer reverse():** This method causes this character sequence to be replaced by the reverse of the sequence.
44	**void setCharAt(int index, char ch):** The character at the specified index is set to ch.
45	**void setLength(int newLength):** This method sets the length of the character sequence.
46	**CharSequence subSequence(int start, int end):** This method returns a new character sequence that is a subsequence of this sequence.
47	**String substring(int start):** This method returns a new String that contains a subsequence of characters currently contained in this character sequence
48	**String substring(int start, int end):** This method returns a new String that contains a subsequence of characters currently contained in this sequence.

49	**String toString():** This method returns a string representing the data in this sequence.
50	**void trimToSize():** This method attempts to reduce storage used for the character sequence.

Example 1: for append() method: concatenates the given argument with this string.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);
}
}
```

Output: Hello Java

Example 2: forThe reverse() method: StringBuilder class reverses the current string.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);
}
}
```

Output: olleH

Example 3: forThe replace() method: replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);
} }
```

Output: HJavalo

5.2.2 StringBuilder:

The StringBuilder is the object having features similar to String class in Java but it offers a mutable sequence of characters. In java String Class produces immutable sequence of characters, but where as the StringBuilder class offersansubstitute to String Class, as it generates a mutable sequence of characters.

Syntax:

```
StringBuilder str = new StringBuilder();
str.append("GOD");
```

5.2.2.1 Important Constructors of StringBuilder class

Table 5.4 : StrinBuilder class constructors

Sr. No.	Constructor Description
1	**StringBuilder():** creates an empty string Builder with the initial capacity of 16.
2	**StringBuilder(String str):** creates a string Builder with the specified string.
3	**StringBuilder(int length):** creates an empty string Builder with the specified capacity as length.

5.2.2.2 Important methods of StringBuilder class

Table 5.5 : StringBuilder class methods

Sr. No	Methods & Descriptions
1	**public StringBuilder append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2	**public StringBuilder insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3	**public StringBuilder replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4	**public StringBuilder delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.

5	**public StringBuilder reverse():** is used to reverse the string.
6	**public int capacity():** is used to return the current capacity.
7	**public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8	**public char charAt(int index):** is used to return the character at the specified position.
9	**public int length():** is used to return the length of the string i.e. total number of characters.
10	**public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11	**public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

Example 1: The StringBuilder append() method concatenates the given argument with this string.

```
class StringBuilderDemo{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);
}
}
```

```
Output:  Hello Java
```

Example 2: The StringBuilder insert() method inserts the given string with this string at the given position.

```
class StringBuilderDemo{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

```
Output:  HJavaello
```

Example 3: StringBuilder class having the method capacity() which returns the current capacity of the Builder. The Initial and by default capacity of the Builder is 16. If the number of character increases more than its current capacity, it increases the capacity by (oldcapacity*2)+2. Let us Consider with an example if your current capacity is 16, it will be changed to (16*2)+2=34.

```
class StringBuilderDemo{
public static void main(String args[]){
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my most favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

```
Output:  34
```

5.2.3. StringTokenizer:

StringTokenizer class in Java is used to breakdown a string into small sequences of charecters called as token.

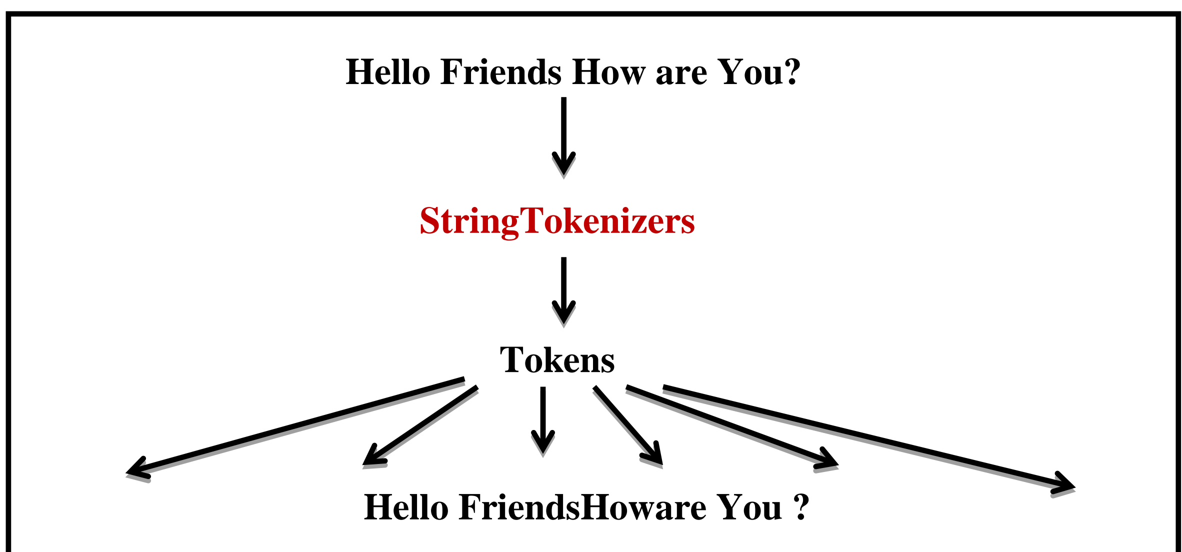
- This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.
- The methods of StringTokenizers do not differentiatebetweennumbers, identifiers, and quoted strings.
- This class methods do not even recognize and skip comments.

A StringTokenizer object internally maintains a current position within the string to be tokenized.

Some operations advance this current position past the characters processed.

To create the StringTokenizer objectsubstring of the string is used and then the token is returned.

Example:



5.2.3.1 StringTokenizer Class constructors

Table 5.6 : StringTokenizer class constructor

Sr.No.	Constructor & Description
1	**StringTokenizer(String str):** This constructor a string tokenizer for the specified string.
2	**StringTokenizer(String str, String delim):** This constructor constructs string tokenizer for the specified string.
3	**StringTokenizer(String str, String delim, boolean returnDelims):** This constructor constructs a string tokenizer for the specified string.

5.2.3.2 StringTokenizer Class methods

Table 5.7 : StringTokenizer class methods

Sr.No.	Method & Description
1	**int countTokens():** This method calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
2	**boolean hasMoreElements():** This method returns the same value as the hasMoreTokens method.
3	**boolean hasMoreTokens():** This method tests if there are more tokens available from this tokenizer's string.
4	**Object nextElement():** This method returns the same value as the nextToken method, except that its declared return value is Object rather than String.
5	**String nextToken():** This method returns the next token from this string tokenizer.
6	**String nextToken(String delim):** This method returns the next token in this string tokenizer's string.

Example 1:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
      StringTokenizer st = new StringTokenizer("my name is rama");
      While(st.hasMoreTokens())
     System.out.println(st.nextToken());
  }
```

```
}
```

```
Output:my  name      is              Rama
```

Example 2:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
     StringTokenizer st = new StringTokenizer("my,name,is,khan");
          // printing next token
    System.out.println("Next token is : " + st.nextToken(","));
  }
}
```

```
Output: Next token is : my
```

5.2.4 StringJoiner:

In java java.util package provide a class StringJoiner which is used to construct a string or sequence of characters separated by a delimiter. It is optionally starting with a supplied prefix and ending with a supplied suffix. This class takes the help of StringBuilder class to append delimiter after each string; StringJoiner does not required to erite a code and is ansimplest way to do that.

Syntax:

```
public StringJoiner(CharSequence delimiter)
```

5.2.4.1 StringJoiner Constructors

Table 5.8 : StringJoinerr class constructors

Sr. no	Constructor Description
1	**Public StringJoiner(CharSequence delimiter):** It constructs a StringJoiner with no characters in it, with no prefix or suffix, and a copy of the supplied delimiter. It throws NullPointerException if delimiter is null.
2	**Public StringJoiner(CharSequence delimiter,CharSequence prefix,CharSequence suffix):** It constructs a StringJoiner with no characters in it using copies of the supplied prefix, delimiter and suffix. It throws NullPointerException if prefix, delimiter, or suffix is null.

5.2.4.2 StringJoiner Methods

Table 5.9 : StringJoiner class methods

Sr. No.	Method Descriptions
1	**Public StringJoiner add(CharSequence newElement):** It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added.
2	**Public StringJoiner merge(StringJoiner other):** It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect.
3	**Public int length():** It returns the length of the String representation of this StringJoiner.
4	**Public StringJoiner setEmptyValue(CharSequence emptyValue):** It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty.

Example 1:

```
import java.util.StringJoiner;
public class StringJoinerExample {
   public static void main(String[] args) {
      StringJoiner joinNames = new StringJoiner(","); // passing comma(,) as delimiter
              // Adding values to StringJoiner
      joinNames.add("Rahul");
      joinNames.add("Raju");
      joinNames.add("Rama");
      joinNames.add("Patil");

      System.out.println(joinNames);
   }
}
```

```
Example 1:  [Rahul,Raju, Rama, Patil]
```

Example 2: The merge() method merges two StringJoiner objects excluding of prefix and suffix of second StringJoiner object.

```
import java.util.StringJoiner;
public class StringJoinerExample {
  public static void main(String[] args) {

      StringJoiner joinNames = new StringJoiner(",", "[", "]");   // passing comma(,) and square-
brackets as delimiter

      // Adding values to StringJoiner
      joinNames.add("Rahul");
      joinNames.add("Raju");

      // Creating StringJoiner with :(colon) delimiter
      StringJoiner joinNames2 = new StringJoiner(":", "[", "]");  // passing colon(:) and square-
brackets as delimiter

      // Adding values to StringJoiner
      joinNames2.add("Rama");
      joinNames2.add("Patil");

      // Merging two StringJoiner
      StringJoiner merge = joinNames.merge(joinNames2);
      System.out.println(merge);
  }
}
```

```
Output: [Rahul,Raju,Rama:Patil ]
```

5.2.5 Difference between String and StringBuffer

There are many differences between String and StringBuffer. The differences between String and StringBuffer are as mentioned below:

Table 5.10 : Difference between String and StringBuffer

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	Stringconsumes more memory when	StringBuffer consumes less memory while

	you concat too many strings as every time it creates new instance, hence it is comparatively slow.	cancating the strings, hence is fast.
3)	String class inherits the equals() method of Object class. This can help to compare the contents of two strings by equals() method.	The equals() method of Object class is not inherited in StringBuffer.

5.2.6 Difference between StringBuffer and StringBuilder

Java offersdifferent methods torepresent a sequence of characters, whichare three different classes String, StringBuffer, and StringBuilder. The String class hold the spectial feature called as immutable. whereas StringBuffer and StringBuilder classes are mutable. There are some of the diffrences than can be used to compare StringBuffer and StringBuilder classes. The StringBuilder class is introduced First in JDK 1.5.

There are many differences between StringBuilder and StringBuffer. Thedifferences between StringBuilder and StringBuffer are as mentioned below:

Table 5.11: Difference between StringBuilder and StringBuffer

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. Synchronised feature does not allow two threads to call the methods simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

5.3 Immutable String in Java

In java, **string objects are immutable**. Immutable simply means un-modifiable or unchangeable. Once string object is created then it does not allow to change its data or state but allow to create a new string object.

Example: Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Future";
```

```
  s.concat(" World");//concat() method appends the string at the end
  System.out.println(s);//will print Future because strings are immutable objects
 }
}
```

```
Output: Future
```

Now let us understand the concept with the help of diagram given below. Here Future is not changed but a new object is created with Future World. And hence the string is known as immutable. As you can see in the above figure that two objects are created but s reference variable still refers to "Future" not to "Future World".

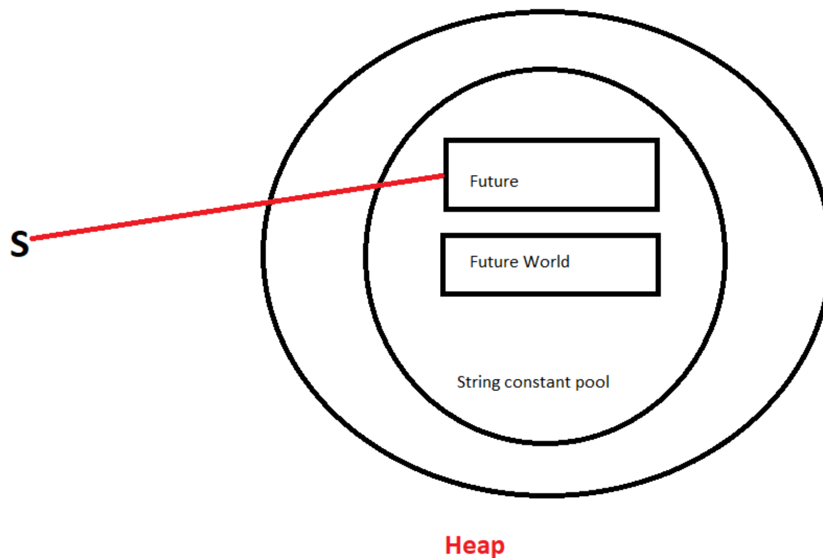


Fig. 5.2 :Example of Immutable string in Java

But it will refer to "Future World" object, if we explicitly assign it to the reference variable.

Example:

```
class Testimmutablestring1{
 public static void main(String args[]){
  String s="Future";
  s=s.concat(" World");
  System.out.println(s);
 }
}
```

```
Output: Future World
```

In such case, s points to the "Future World". Please notice that still Future object is not modified.

5.3.1 Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there aremultiple reference variables let us say 5, all referes to one object "Future". It will be affected to all the reference variables, if one reference variable changes the value of the object. Which makes string objects as immutable in java.

5.4 Exception Handling

5.4.1 Exception

An Exception is an unexpected or undesirablesituation that disturbs the normal flow of the program. The program execution gets terminated when ever the excepton is occurred. This produces a system generated error message for us. The java allows us to handle or controlexceptions, which allows us to provide the meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

5.4.2 Why an exception occurs?

The exception can be caused because of severals reasons. Let us consider with an situation: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

5.4.3 How to Handle Exception?

If an exception dose not handled by programmer and if it occurs then program execution gets terminated and a system generated error message to inform to user.

5.4.4 Advantage of exception handling

Exception handling guarantees that the flow of the program will continue even after an exception occurs. For example, if a program has bunch of statements and an exception occurs mid-way after

executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

Exception handling make assured that all the statements execute and the flow of program doesn't break.

5.4.5 Exception Hierarchy in Java

Exception in java is classified in various types such as Checked and unchecked. Following diagram shows the hierarchy of Exception Class in java along with its all types and few examples.

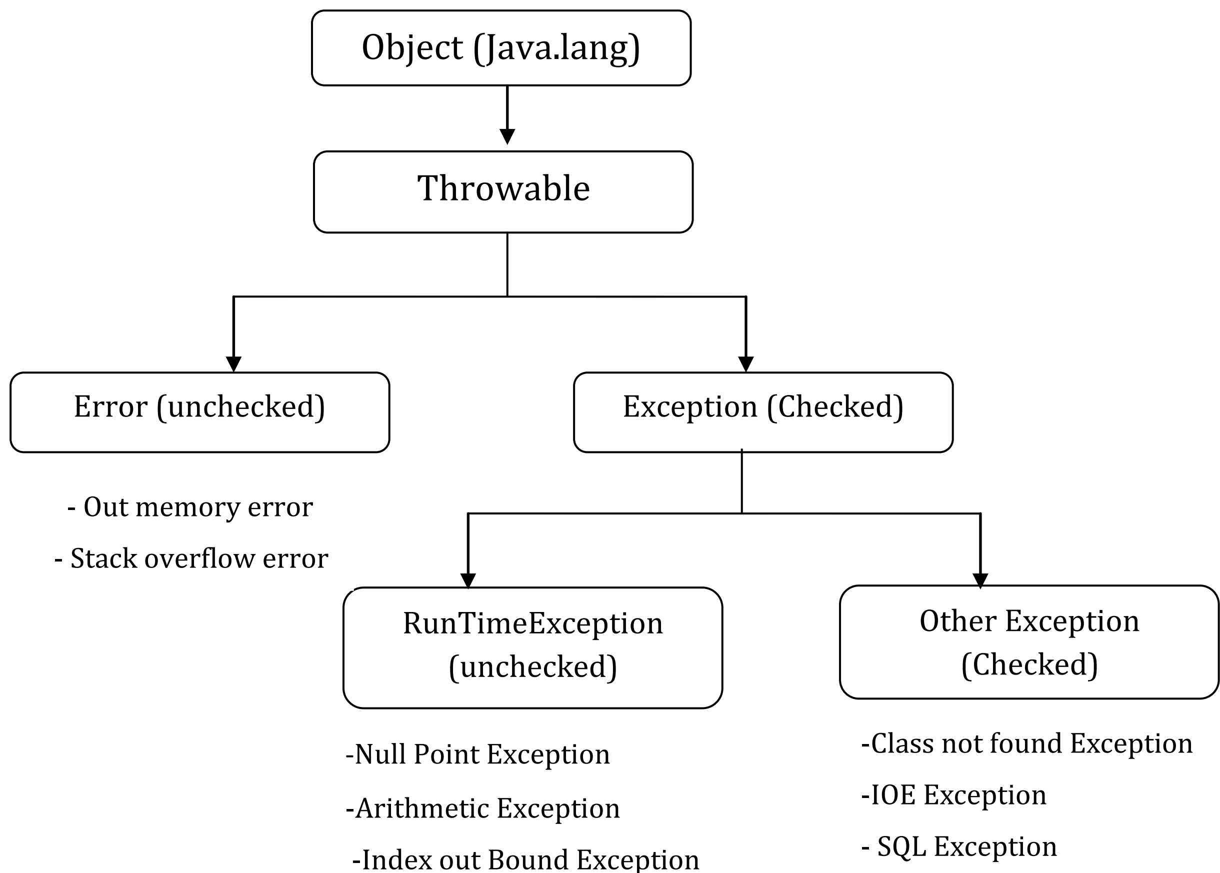


Fig. 5.3 : Exception Class hierarchy

The diagram shows the exception hierarchy, All errors and exceptions in Java has a comman superclass called as*Throwable* class. Objects those are instances of *Throwable* or one of its subclasses are thrown by JVM or by Java throw statement.

5.4.5.1 Error: An Error is a subclass of *Throwable* that represents serious errors that can't be handled. A throws clause does not required declaring by methodfor Error or any of its subclasses

for the uncaught errors thrown during execution of the method. Error and its subclasses are unchecked exceptions.
Examples of errors: OutOfMemoryError, StackOverflowError, AssertionError, IOError, NoClassDefFoundError etc.

5.4.5.2 Exception: Exception is a subclass of *Throwable*. Exception and its subclasses represent the problems. Program has to recover and should be handle the occurred problems. Exception has two sub type:

1. RunTimeException or Unchecked Exception: These exception are need not be handled before compilation of the source code. Complier doesn’t check whether the exceptions are handled before compilation or not that’s why they are called as Unchecked exceptions. The classes that extends *RunTimeException* are called unchecked Exception. An unchecked exception can’t be checked at compile time as it occurs at runtime, an application should handle these exceptions.
For example: NullPointerException, ArithmeticException, IllegalArgumentException, IndexOutOfBoundException etc.

2. Checked Exception: Checked exception are checked at compile time and application should handle these exceptions before the compilation of program. If these exceptions are not handled it will not allow you to even compile the source code. That’s why they are called as Checked Exception. All the classes in exception hierarchythat extends Throwable class except RunTimeExpception and Error are called Checked Exception.
Some of the examples of Checked Exceptions are: IOException, SQLException, FileNotFoundException, ClassNotFoundException, NoSuchMethodException etc.

3. User-defined custom exception: We can also create our own exception. Here are few rules:

1. Throwable must be the superclass of all exceptions.
2. One needs to extend the RunTimeException class to to create a RuntimeException.
3. One needs to extend the Exception class to create the checked exception.

5.4.6 Exceptions Methods

Following is the list of important methods available in the Throwable class.

Table 5.12 : Exceptions Methods

Sr. No.	**Method & Description**
1	**public String getMessage():** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause():** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString():** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace():** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace():** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace():** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

5.5 Terminology Related to Exceptions

5.5.1 Try Block:

The "try" is a keyword which is used to mention a block in which exception code or risky code can be placed. It means we write the The code which may cause exception (risky code) in try block. If exception occure then suddenly compiler will terminate the excection of try block and will start the execution of catch or finally block. The try block must be used with the combination of either catch or finally. In simple words, try block alone can not be used.

5.5.2 Catch:

The "catch" block is used to handle the exception. It must be used with and after try block. In simple words catch block alone can not be used. If the exception is occurred in try block then only catch block will be excuted. In catch block we write a code to handle the exception or to describe more information about exception to user. It may or may not be used with finally block.

Syntax:

```
try {
  // Risky code
} catch (ExceptionType e1) {
  // Catch block
}
```

5.5.2.1 Multiple Catch Blocks:A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

The Example mentioned abovedetermines three catch blocks, but you can have any number of catch blocks with a single try. If an exception occurs, then is thrown to the first catch block in the list.In will be handeled in block one, if the data type of the exception thrown matches ExceptionType1. Otherwise the exception passes down to the next catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

5.5.3 Finally:

The "finally" block is used either after try or catch block to execute the important code of the program. It is executed even an exception is handled or even is not handled. It guarantees that the code written in finally block will surely execute even if exception occur or doesn’t occur.

Syntax:

```
try {
   // Risky code
} catch (ExceptionType e1) {
   // Catch block
}
finally {
   // The finally block always executes.
}
```

5.5.4 Throw:

The "throw" keyword is used to create and throw an Exception. It means user can manually create or generate the exception using throw keywords.

5.5.5 Throws:

The "throws" keyword is used to announceexceptions. It doesn't throw an exception. It indicates that there may occur an exception in the method. It informs the compiler that the specified methods may throws the exception, in that case user dosent have to handle the exception that will be handled by the compiler. The throws will always be used along with method signature.

See the below diagram to understand the flow of the call stack.

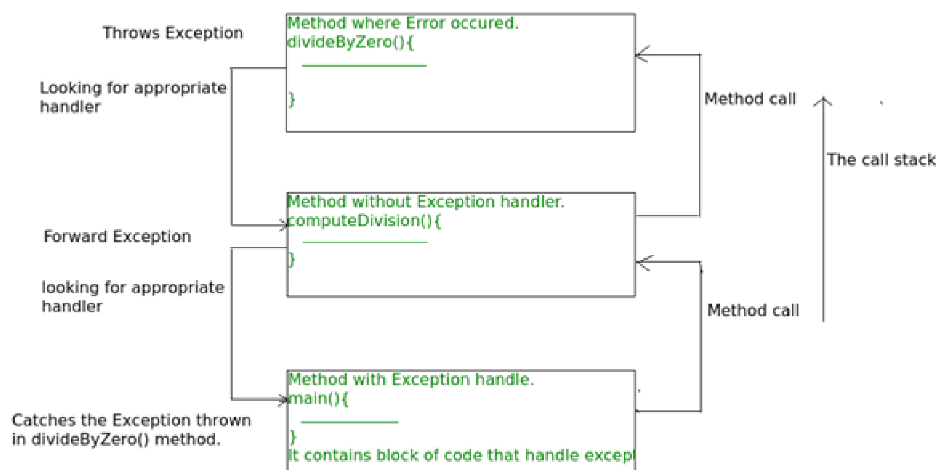


Fig. 5.4 : Flow of call stack

Java Exception Handling Example

Let us try to understand the concept of Exception Handling by using a try-catch statement to handle the exception.

Example 1:

```
public class JavaExceptionDemo{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("Exception is Occurred");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
Exception is Occurred...
```

Example 2: Using Finaly

```
class ExceptioDemo
{
	public static void main(String[] args)
	{
		int a=10,b;
		System.out.println("Hello World!");
		try
		{
			System.out.println("You are in Try Block");
			b=1/0;
		}
```

```
		catch (Exception e)
		{
			System.out.println("You are in Exception Block");
			e.printStackTrace();
		}
		finally
		{
			System.out.println("You are in finally Block");
		}
	}
}
```

Output:

```
You are in Try Block.
You are in Exception Block.
You are in finally Block.
```

Example 3: Example of throw

```
class ExeceptionDemo
{
	public static void main(String[] args) throws Exception
	{
		System.out.println("Hello World!");
		int a=10,b;
		throw new Exception();

	}
}
```

Output:

5.6 Important Points to remember:

1. In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler inside own **catch** block for each of them.
2. If in try an exception occure, then the exception handler associated with it will handle that exception. To associate exception handler by putting the **catch** block after it. Try block can have more than one exception handlers. Every **catch** block work as an exception handler that handles the exception of the type indicated by its argument. The argument and Exception Type defines the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
3. For each try block there can be zero or more catch blocks, but **only one** finally block.
4. The finally block is optional.It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And it may be executed after the **try** block, if exception does not occur. The finally block in java is used to put important codes that need to be executed in any of the situations. Some of the situations that can be executed in finaly block are such as clean up code e.g. closing the file or closing the connection.

Outcomes:

- Students will able to study java string, string class , stringBuffer classes ,their constructors and methods..
- Students will able to study the immutable string in java.
- Students will able to exception handling, exception methods in java.

Questions:

1. Write a brief note on String class in java.
2. List and explain some of the important methods of string class.
3. Write a java program to measure length of string and to print the string in reverse order.
4. With suitable example explain the need of StringBuffer class.
5. With suitable example explain the need of StringBuilder class.
6. Compare StringBuffer and StringBuillder class.
7. List and explain some of the important methods StringBuilder class with example.
8. List and explain some of the important methods StringBuffer class with example.
9. Describe with example the purpose of StringTokenizer class. List the important methods in class.

10. Write a Brief note on StringJoiner class.
11. Differentiate between StringBuffer and String Class.
12. Differentiate between StringBuilder and String Class.
13. Describe with short note concept of Imutable string in java.
14. What is Exception? Write a short note on Exception handling
15. Describe the causes of Exception occurring?
16. Write a short note on Exception handling what are the advantages of Exception handling.
17. With neat diagram explain the Exception Hierarchy.
18. Compare Checked Vs Unchecked Exception.
19. List explains the terminologies related to exception handling.
20. Write a program to demonstrate the use of Try, catch and finally.

CHAPTER 6

PACKAGE AND DEFERRED IMPLIMENTATIONS

Objectives:

- To study Packages in java, working of packages, subpackages in java.
- To study types of packages, like user defined and built-in packages,access protection packages.
- To study abstraction and interfaces, difference between abstract class and interfaces, etc.
- To study generalization, specialization and inheritance in java.

6.1 What is Package in Java?

A Package is a collection of related classes. It helps arrange your classes into a folder structure and make it simple to trace and use them. More importantly, it helps improve re-usability.

A package in Java allows compressing a group of classes, interfaces, enumerations, annotations, and sub-packages. In simple word, you can consider of java packages as being similar to different folders on your computer.

When software is created in any programming language, it can be consisting of hundreds or even thousands of individual classes. It makes efficient tomentain things structured by placing related classes and interfaces into packages.

Each package arranges its classes and interfaces into a separate namespace, or name group and each one has its unique name.But interfaces and classes in the same package may not have the same name, Butthey can appear in different packages. This can be acchived by allocating a separate namespace to each package.

- Preventing naming conflicts. Let consider with an example there can be two classes with name Students in two packages, college.stud.cse.Studentsand college.stud.ee.Students

- Pacakage Make easir to search and locate and usage of classes, interfaces, enumerations and annotations.

- Providing controlled access: The classes having protected and defaultaccess specifier have package level access control. Members with protected access specifier areaccessible by classes in

the same package and its subclasses. A memberswithout any access specifier called asdefault member are accessible by classes in the same package only.

- Packages can be considered as a way of data hiding this feature is called as data encapsulation.

While creating the classes we need to put related classes into packages. Then by using an import statement classes from existing packages can be used used it in our program. A package is just like acontainer, which consist of a group of related classes where some of the classes are accessible and are exposed and others are kept for internal purpose.Package allows us to reuse existing classes from the packages as many time as we need it in our program.

6.2 How packages work?

Package names and directory structure are closely related. Let us consider with an example if a package name is company.staff.testing, then there are three directories, company, staff and testingsuch that testingis present in staff and staff is present company. Also, the directory companyis accessible through CLASSPATH variable, i.e., path of parent directory of companyis present in CLASSPATH. The concept is to make easy to locate and search the classes.

6.2.1 Adding a class to a Package:

We can add any number of classes to aexisting or newly created package by using package name at the starting of the program and saving it in the package directory. We need to create new **java** file to define a public class, other method is to add the new class to an existing **.java** file and recompile it.

6.2.2 Subpackages:

Packages that are defined inside another package are called as **subpackages**. These packages need to importexplicitly; they can not be imported implicietly. The members of a subpackage have no access specifiers, i.e., they are considered as different package for protected and default access specifiers.

Example :

```
import java.sql.*;
```

sql is a newly created subpackage inside java package.

6.2.3 Accessing classes inside a package

Consider following two statements :

```
// import the Vector class from util package.

import java.util.vector;

// import all the classes from sql package

import java.sql.*;
```

6.3 Types of packages:

In java packages can be of following two types.

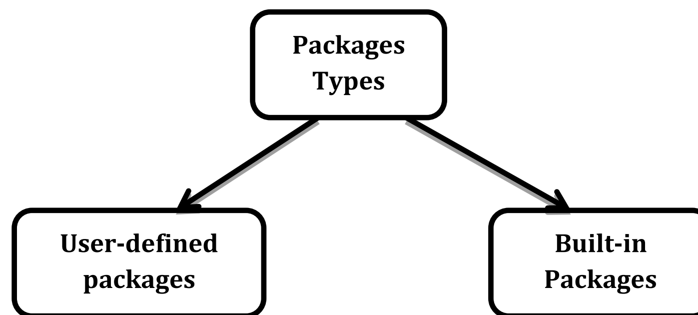


Fig 6.1: Packages Types

6.3.1 Built-in Packages

The package consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) java.lang: This is a default package, i.e this package is automatically imported and contains language support classes such as classes which defines primitive data types, math operations etc.

2) java.io: This package contains classes that support various input / output operations.

3) java.util: This package contains utility classes which can be used to implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) java.applet: This package contains classes for creating Applets.

5) java.awt: This package contains classes for implementing the components for graphical user interfaces like button, frames, menus etc.

6) java.net: This package contains classes for supporting networking operations.

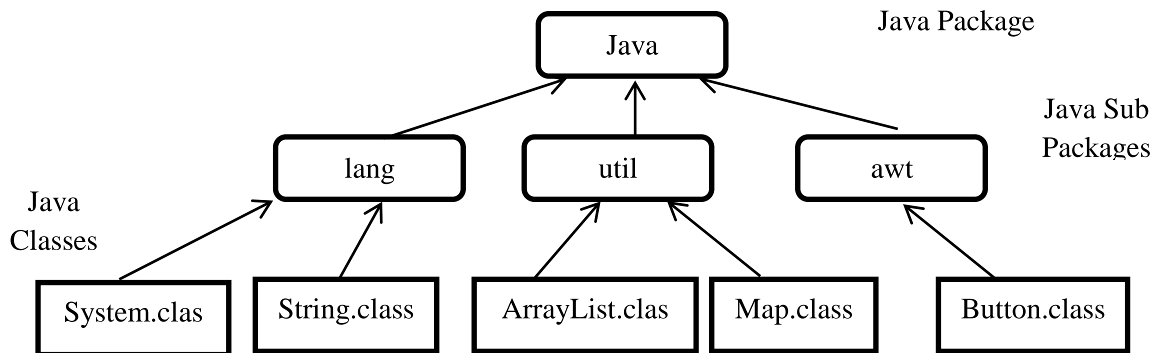


Fig. 6.2:Built-in packages

6.3.2 User-defined packages

These are the packages that are created by the user. Let us consider with an example by create a directory myPackage (name should be same as the name of the package). Then create public class MyClass inside the directorymyPackage with the first statement being the package names.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;
public class MyClass
{
   public void getNames(String s)
   {
      System.out.println(s);
   }
}
```

Now we can use the MyClass class in our program by importing 'MyClass' class from 'names' myPackage

```
import myPackage.MyClass;

public class PrintName

{

  public static void main(String args[])

  {

    // Initializing the String variable

    // with a value

    String name = "Welcome to My Package";

        // Creating an instance of class MyClass in

    // the package.

    MyClass obj = new MyClass();

        obj.getNames(name);

  }

}
```

Note : MyClass.java must be saved inside the myPackage directory since it is a part of the package.

```
Output: Welcome to My Package.
```

Example-2: In this example we will understand creating user defined package

Creating our first package: File name – ClassOne.java

```
package package_one;

 public class ClassOne {

   public void methodClassOne() {

      System.out.println("Hello there its ClassOne");

   } }
```

Creating our second package:

File name – ClassTwo.java

```
package package_two;

 public class ClassTwo {

   public void methodClassTwo(){
```

```
        System.out.println("Hello there i am ClassTwo");
    }
}
```

Making use of both the created packages:

File name – Testing.java

```
import package_one.ClassTwo;
import package_two.ClassOne;
  public class Testing {
    public static void main(String[] args){
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

```
Output:
Hello there i am ClassTwo
Hello there its ClassOne
```

Compileing java package: If IDE is not used then,For compiling the java package you need to follow the syntax given below:

```
Syntax:    javac -d directory javafilename
For example : javac -d . Simple.java
```

The -d switch defines the location where to put the generated class file. One can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. Incase to place the package within the same directory, you can use . (dot).

Running java package program: To run java package you need to use fully qualified name e.g. mypack.Simple etc to run the class.

```
To Compile: javac -d . Simple.java
To Run: java mypack.Simple
```

```
Output:Welcome to package
```

The -d is a switch that defines the location for compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three diffrent ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use packagename like package.* then all the classes and interfaces of this specified package will be accessible but not subpackages and classes in subpackages.

The import keyword allows the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
  package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

3) Using fully qualified name

If fully qualified name is used to import package then only declared class of this package will be accessible. But for accessing any class or interface of the packageone need to use fully qualified name every time.

This method is generally used when two packages have same class name, consider an example java.util and java.sql packages contain Date class.

Let us consideran following block of code to understand use of package by importing fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }  }
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

6.4 Import a Package

There are many packages to choose from. Consider an example to import Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package in an existing class include an asterisk sign (*) at the end of import statement. Let us consider the the following statement will import ALL the classes in the java.util package:

```
Example   :   import java.util.*;
```

6.5 Access Protection in Java Packages

Java offers diffrent access control mechanism and its access specifiers. Packages in Java add another layer to access control. The collection of classes and packages together called as data encapsulation. The packages in java act as containers for classes, interfaces and other subordinate packages, where as classes in package act as containers for data and code. This relationship between packages and classes inJava, packages offer four categories of visibility for class members:

- Sub-classes in the same package
- Non-subclasses in the same package
- Sub-classes in different packages
- Classes that are neither in the same package nor sub-classes

The table mentioned bellow describe different typesof possible access between classes and packages:

Table 6.1 : Access Protection in Java Packages

	Private	**No Modifier**	**Protected**	**Public**
Same Class	Yes	Yes	Yes	Yes
Same Package Subclasses	No	Yes	Yes	Yes
Same Package Non-Subclasses	No	Yes	Yes	Yes
Different Packages Subclasses	No	No	Yes	Yes
Different Packages Non- Subclasses	No	No	No	Yes

We can simplify the data in the above table as follows:

6.5.1 Setting CLASSPATH:

CLASSPATH can be set by diffrent ways as mentioned bellow:

To set CLASSPATH permanently in the Windows environment: choose control panel → System → Advanced→Environment Variables → choose “System Variables” (for all the users) or “User Variables” (only the currently login user) ?choose “Edit” (if CLASSPATH already exists) or “New” → Enter “CLASSPATH” as the variable name → Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-

api.jar”). Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH.

To check the current CLASSPATH, run the following command:

```
> SET CLASSPATH
```

To set the CLASSPATH temporarily for that particular CMD shell session by issuing the following command:

```
> SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
```

The CLASSPATH can also be set without using CLASSPATH environment variable, It can also be done by using the command-line option -classpath or -cp of the javac and java commands, for example,

```
> java –classpath c:\javaproject\classes com.abc.project1.subproject2. MyClass3
```

6.5.2 Important Points to Remember

- Every class is part of some package. The class names are put into the default package if import statement is not mentioned.
- Class can have more than one import package statements, but A class can have only one package statement.
- The package name must have same name as the directory under which the file is saved
- Package declaration must be the first statement, followed by package import.

6.6 Abstract Classes and Interfaces

A class declared using “**abstract**” keyword is known as abstract class. It can have methods without method body called as abstract methods as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. It can have abstract and non-abstract methods. This abstract class needs to be extended and its method need tto be implemented by class who is extending the class. That is the abstract class can be extended by any other class which have to implement all the abstract method declared in abstract class. It cannot be instantiated.

6.6.1 Abstraction

It is an object oriented feature which allows hiding the implementation details and showing only functionality to the user. It hides the internal details and shows only essential things to the user,

Let us consider for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction allows you consentrate on what the object does but not on how it does it.

Ways to achieve Abstraction

The different methods to achieve abstraction in java are as follow

Abstract class (0 to 100%)

Interface (100%)

Syntax:

```
//Declaration using abstract keyword
abstract class A{
   //This is abstract method
   abstract void myMethod();
   //This is concrete method with body
   void anotherMethod(){
      //Does something
   }}
```

Example 1: In this example we will try to understand Abstract class Implementation

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{
   public void sound(){
            System.out.println("Woof");
   }
   public static void main(String args[]){
```

```
            Animal obj = new Dog();

            obj.sound();

    }

}
```

```
Output: Woof
```

Example-2: Consider the following example, In given example Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.
In this example you can observe the implementation class which is hidden to the end user, and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. In this example, the isstance created for any class will call the relevant draw method, suppose if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{

abstract void draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape{

void draw(){System.out.println("drawing rectangle");}

}

class Circle1 extends Shape{

void draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

class TestAbstraction1{

public static void main(String args[]){

Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape()
method

s.draw();

}

}
```

```
Output:   drawing circle
```

Example-3: Example of an abstract class that has abstract and non-abstract methods

```
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
Output:

                bike is created
 running safely..
gear changed
```

6.6.2 Points to Remember

1. An abstract keyword can be used to declarean abstract class.
2. Abstract classcan have abstract and non-abstract methods.
3. It cannot be instantiated.
4. Abstract class can have constructors and static methods also.
5. It can have final methods which will force the subclass not to change the body of the method.

6. The methods which are marked as abstract need to be implemented in a class who is extending the abstract class.

6.6.3 Difference between abstract class and interface

Abstraction can be achived by either abstract class or interface, and both can be used to achieve abstraction where we can declare the abstract methods. But remember abstract class and interface both can't be instantiated.

But still both are different from each other, following are the some of the diffrences between Abstract class and Interface.

Table 6.2: Difference between abstract class and interface

Abstract class	**Interface**
Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
The implementation of interface can be provided by abstract class.	The implementation of abstract class can't be provided by interface.
To declare abstract classabstract keyword is used.	To declare interface the interface keyword is used.
An abstract class can implement multiple Java interfaces and can extend another Java class.	An interface canonlyinherit another Java interface by extending it.
An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
A Java abstract class can have class members with different access specifiers like private, protected, etc.	In Java interface all members are public by default.

Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example: Example to understand abstract class and interface in Java.

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the me
thods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Output:

```
I am a
I am b
```

```
I am c
I am d
```

6.7 Interfaces in Java

An interface can have methods and variables just like class can have, but the methods declared in an interface are by default abstract, that is only method signature can be specified, but method body is not allowed.

- An interface defines what a class must do and what must not do. It can be considered as a Xerox copy of the class.
- An Interface is about capabilities, just understand with an example like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- The class must be declared abstract, If a class implements an interface and does not provide method bodies for all functions specified in the interface.

To declare an interface, use **interface** keyword. It is used to provide total abstraction. In simple words all the methods in an interface are declared with an empty body,as well all methods are public and all fields are public, static and final by default. All the methods declared in the interface must be implemented by a class that implement interface. To implement interface use **implements** keyword. Interface is used to implement multiple and hybrid inheritance in java, as one class cannot extend more than one class but can implement many classes and can extend single class.

Syntax:

```
interface <interface_name> {
    // declare constant fields
   // declare methods that abstract (by default).
}
```

Example:

```
interfacePlayer
{
   finalintid = 10;
   intmove();
```

```
}
```

Let’s understand the consept with real world example, Consider the example of vehicles like bicycle, car, bike, they have common functionalities. So to put all these common functionalities let us make an interface. And lets different classes like Bicycle, Bike, car ….etc allows to implement all these functionalities in their own class in their own way.

```
import java.io.*;
 interface Vehicle {
        // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){
                gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
                speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
               speed = speed - decrement;
    }

    public void printStates() {
```

```
        System.out.println("speed: " + speed   + " gear: " + gear);
    }
}

class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
                speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
                speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed  + " gear: " + gear);
    }
 }
class GFG {

    public static void main (String[] args) {

        // creating an inatance of Bicycle  doing some operations
        Bicycle bicycle = new Bicycle();
```

```
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

```
Output;
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1
```

6.7.1 Implementing multiple interfaces

Java directly does not allow implementingmultiple inheritances but we can achieve the impact of multiple inheritances using interfaces. In interfaces, a class can implement more than one interface by using implement keywords followed by interface names separated by comma. Let's say we have two interfaces with same method with name demo and different return types say float and double.

```
publicinterfaceInterfaceA
{
   publicfloat demo ();
}
publicinterfaceInterfaceB
{
```

```
   publicdoubledemo ();
}
```

Now, Let us consider the class that implements both those interfaces:

```
publicclassTesting implementsInterfaceA, InterfaceB
{
publicdoubledemo ()
   {
      return11.5;
   }
}
```

Example of multiple interfaces Implementation in Java

```
interfaceReadable {
        voidread();
}
interface Eatable {
        void eat();
}
```

Bird class will implement both interfaces

```
Class Person implements Readable, Eatable {

        public void read() {
                System.out.println("Person Reading");
        }
        public void eat() {
                System.out.println("Person eats");
        }
        // It can have more own methods.
}
```

Test multiple interfaces example

```
public class Sample {
        public static void main(String[] args) {
                Person b = new Person ();
                b.eat();
                b.read();
        }
}
```

Output:
Person eats
Person Reading

6.8 Generalization, Specialization, and Inheritance

6.8.1 Generalization

Generalization is the process, which extracts the shared features from two or more classes, and combines the extracted features into a generalized superclass. Shared features can be attributes, associations, or methods. The process of Generalization is the bottom-up process of abstraction, in which we club the differences among entities according to the common feature and generalize them into a single superclass. The original entities are then becomes subclasses of it.

The process of converting a subclass type into a superclass type is called '**Generalization**'because we are making the subclass to become more general and its scope is widening.

For example, if we say Apple is a Fruit, there will be no objection

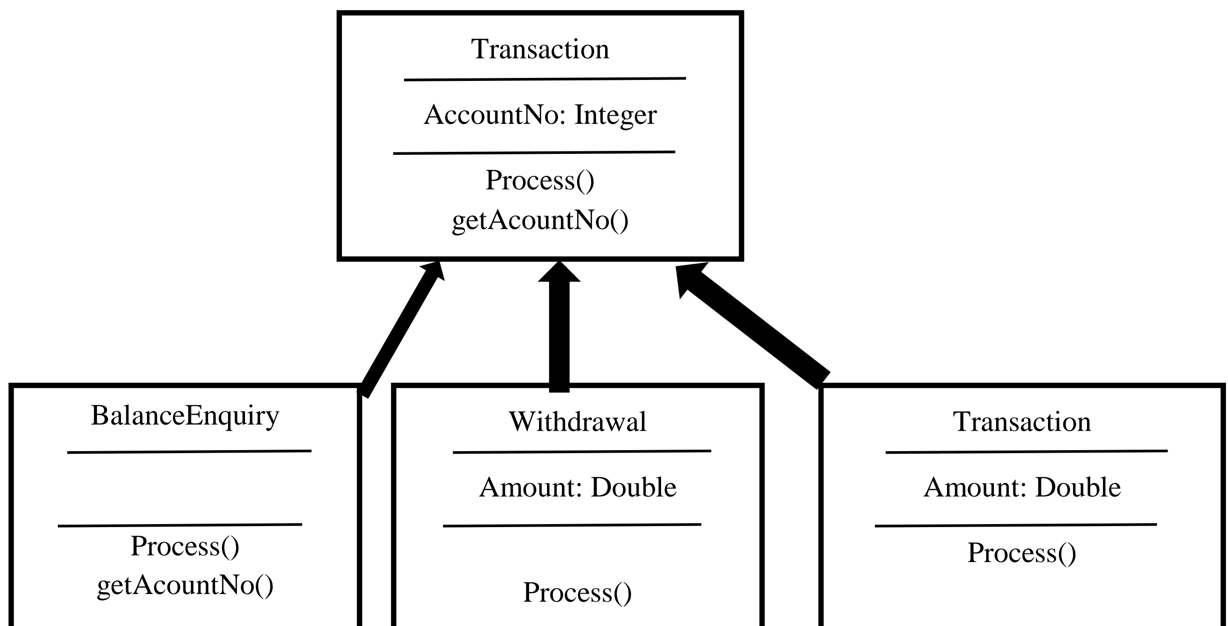


Figure.6.3:Generalization Example

6.8.2 Specialization

Converting a super class type into a sub class type is called ‘Specialization‘. In specialization, we are coming down from more general form to a specific form and hence the scope is narrowed. Hence, this is called narrowing or down-casting. Specialization is defined as the process of subclassing a superclass entity on the basis of some distinguishing characteristic of the entity in the superclass.

As the classes will become more and more specific thus giving rise to more and more doubts and hence narrowing is not safe. For example if we say Fruit is aApple we need a proof.

The reason for creating such hierarchical relationship is Certain attributes of the superclass may apply to some but not all entities of the superclass. These extended subclasses then can be used to encompass the entities to which these attributes apply.

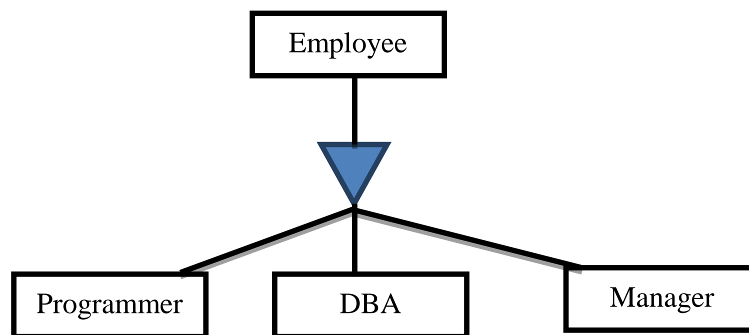


Figure.6.4: Specialization Example

Specialization means creating new subclasses from an existing class, which is exactly opposite to generalization. If it turns out that certain attributes, associations, or methods only apply to some of the objects of the class, a subclass can be created. The most inclusive class in a generalization/specialization is called the superclass and is generally located at the top of the diagram. The more specific classes are called subclasses and are generally placed below the superclass.

Outcomes:

- Students will able to study Packages in java, working of packages, subpackages in java.
- Students will able tostudy types of packages, like user defined and built-in packages,access protection packages.

- Students will able tostudy abstraction and interfaces ,difference between abstract class and interfaces, etc.
- Students will able tostudy generalization, specialization and inheritance in java.

Questions:

1. What is package in java? How packages work?
2. How to create the packages? Explain with suitable example.
3. Write a short note on types of packages.
4. With suitable example explain how to run java package program?
5. Write a short note about how to import java packages.
6. Write a short note on access protection in java package. Describe various access levels in packages.
7. What is abstraction? What are the various ways to achieve abstraction?
8. With suitable example explain the concept of interface; also describe the important points related to interface.
9. With suitable example explain the concept of abstract class; also describe the important points related to abstract class.
10. Compare abstract class and interface.
11. With suitable example explain how multiple inheritance can be achieved in java using interface?
12. Write a short note on generalization.
13. Write a short note on Specialization.
14. Compare generalization and specialization, and explain how they are related to inheritance.

CHAPTER 7

JAVA INPUT-OUTPUT

Objectives:

- To study file class in java, its constructors and methods in java.
- To study working of read and write files using stream in java.
- To study FileiInputStream class, FileiOutputStream class, their constructors and methods.
- To study BufferReader class, BufferWriter class, its constructors and methods.
- To study serialization and deserialization in java.

7.1 Introduction

File handling is an essential part of any application.Java provides several methods for handling the files operations such as creating, reading, updating, and deleting files. The java.io package provides File class which allows us to work with files. One needs to create an object of the class, and specify the filename or directory name in order to use the File class. The File class is Java's representation of a file or directory path name. A simple string is not adequate to name them because file and directory names have different formats on different platforms. The File class have several methods that can be used for working with the path name, listing the contents of a directory, deleting and renaming files, creating new directories, and determining several common attributes of files and directories.

7.2 File Class in Java

The File class from the java.io package, allows us to work with files. To use the File class, create an object of the class, and specify the filename or directory name. The File class is Java's representation of a file or directory path name. A simple string is not adequate to name them because file and directory names have different formats on different platforms. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

7.2.1 File Class Constructors

Table 7.1: File Class Constructors

Sr. No.	Constructor	Description
1	File(File parent, String child)	Creates a new File instance from a parent abstract pathname and a child pathname string.
2	File(String pathname)	Creates a new File instance by converting the given pathname string into an abstract pathname.
3	File(String parent, String child)	Creates a new File instance from a parent pathname string and a child pathname string.
4	File(URI uri)	Creates a new File instance by converting the given file: URI into an abstract pathname.

7.2.2 File Class Methods

Table 7.2 : File Class Methods

Sr. No	Methods & Descriptions
1	boolean canExecute() : Tests whether the application can execute the file denoted by this abstract pathname.
2	boolean canRead() : Tests whether the application can read the file denoted by this abstract pathname.
3	boolean canWrite() : Tests whether the application can modify the file denoted by this abstract pathname.
4	int compareTo(File pathname) : Compares two abstract pathnames lexicographically.
5	boolean createNewFile() : Atomically creates a new, empty file named by this abstract pathname .
6	static File createTempFile(String prefix, String suffix) : Creates an empty file in the default temporary-file directory. boolean delete() : Deletes the file or directory denoted by this abstract pathname.
7	boolean equals(Object obj) : Tests this abstract pathname for equality with the given object.
8	boolean exists() : Tests whether the file or directory denoted by this abstract pathname exists.
9	String getAbsolutePath() : Returns the absolute pathname string of this abstract pathname.
10	long getFreeSpace() : Returns the number of unallocated bytes in the partition .
11	String getName() : Returns the name of the file or directory denoted by this abstract

	pathname.
12	String getParent() : Returns the pathname string of this abstract pathname's parent.
13	File getParentFile() : Returns the abstract pathname of this abstract pathname's parent.
14	String getPath() : Converts this abstract pathname into a pathname string.
15	boolean isDirectory() : Tests whether the file denoted by this pathname is a directory.
16	boolean isFile() : Tests whether the file denoted by this abstract pathname is a normal file.
17	boolean isHidden() : Tests whether the file named by this abstract pathname is a hidden file.
18	long length() : Returns the length of the file denoted by this abstract pathname.
19	String[] list() : Returns an array of strings naming the files and directories in the directory .
20	File[] listFiles() : Returns an array of abstract pathnames denoting the files in the directory.
21	boolean mkdir() : Creates the directory named by this abstract pathname.
22	boolean setExecutable(boolean executable) : A convenience method to set the owner's execute permission.
23	boolean setReadable(boolean readable) : A convenience method to set the owner's read permission.
24	boolean setReadable(boolean readable, boolean ownerOnly) : Sets the owner's or everybody's read permission.
25	boolean setReadOnly() : Marks the file or directory named so that only read operations are allowed.
26	boolean setWritable(boolean writable) : A convenience method to set the owner's write permission.

Program 1: Program to create a file.

```
import java.io.*;
public class FileDemo {
   public static void main(String[] args) {


      try {
         File file = new File("javaFile123.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
```

```
        }
      } catch (IOException e) {
          e.printStackTrace();
      }


   }
}
```

```
Output: New File is created!
```

Program 2 : Program to insure if a file or directory physically exist on drive or not.

In this program, we accept a file or directory name from, command line arguments. Then the program can be implemented to check if that file or directory physically exist on drive or not and it can also be implemented to displays the property of that file or directory.

```
*import java.io.File;
class fileProperty
{
    public static void main(String[] args) {
        //accept file name or directory name through command line args
        String fname =args[0];
          //pass the filename or directory name to File object
        File f = new File(fname);
          //apply File class methods on File object
        System.out.println("File name :"+f.getName());
        System.out.println("Path: "+f.getPath());
        System.out.println("Absolute path:" +f.getAbsolutePath());
        System.out.println("Parent:"+f.getParent());
        System.out.println("Exists :"+f.exists());
        if(f.exists())
        {
            System.out.println("Is writeable:"+f.canWrite());
```

```
            System.out.println("Is readable"+f.canRead());

            System.out.println("Is a directory:"+f.isDirectory());

            System.out.println("File Size in bytes "+f.length());

        }

    }

}
```

```
Output:
File name :file.txt
Path: file.txt
Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt
Parent:null
Exists :true
Is writeable:true
Is readabletrue
Is a directory:false
File Size in bytes 20
```

7.3 Reading and Writing Files

A stream in java can be defined as a sequence of charecters or data. The Java provides **InputStream class which** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.

- Object
  - InputStream
    - FileInputStream
    - ByteArrayInputStream
    - FilterInputStream
      - BufferedInputStram
      - DataInputStram
    - ObjectInputStream
  - OutputStream
    - FileOutputStream
    - ByteArrayOutputStrea
    - FilterOutputStream
      - BufferedOutputStram
      - DataOutputStram
    - ObjectOutputStream

Fig.7.1: Hierarchy of Object

7.4 Stream

A stream in java can be defined as a sequence of charecters or data . There are two kinds of Streams −

1. InputStream − The InputStream is used to read data from a source.
2. OutputStream − The OutputStream object is used for writing data to a destination.

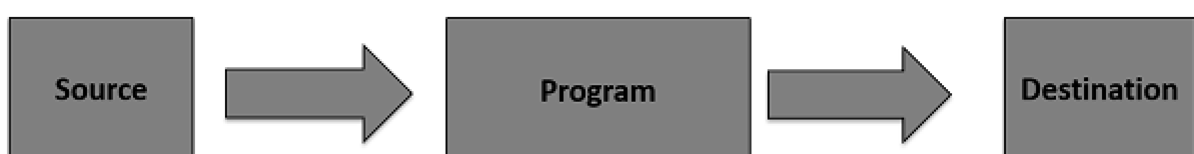


Fig. 7.2: Stream

The two important streams offered in java are FileInputStream and FileOutputStream, which would be discussed as follow.

7.4.1 FileOutputStream

FileOutputStream is used to write a data to destination, it create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.
FileOutputStream object can be created in two different ways by usingtwo constructors in FileOutputStream.

Sr.No.	Methods& Descriptions
1	FileOutputStream("C:/java/hello"): This constructor takes string as a parameter to create an outputstream object to write the file. The string can representsa file name.
2	FileOutputStream(f): This constructor takes a file object to create an output stream object to write the file. File object can be created as follow File f = new File("C:/java/hello");

Once we create have OutputStream object, need to use list of helper methods, which can be used to write to stream or to do other operations on the stream.

7.4.1.1 FileOutputStream Class Methods

Table 7.3 : FileOutputStream Class Methods

Sr.No.	Methods& Descriptions
1	public void close() throws IOException{} : This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {}: This method cleans up the connection to the file. When there are no more references to this stream, then close method of this file output stream is called. Throws an IOException.
3	public void write(int w)throws IOException{} : This methods writes the specified byte to the output stream.
4	public void write(byte[] w) : Writes the stream of w.length bytes from the mentioned byte array to the OutputStream.

Example:

```
import java.io.FileOutputStream;
```

```
public class FileOutputStreamExample {
    public static void main(String args[]){
          try{
            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
            String s="Welcome to java Files.";
            byte b[]=s.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
           }catch(Exception e){System.out.println(e);}
      }
}
```

```
Output:  Success...
The content of a text file testout.txt is set with the data Welcome to Java Files.
```

7.4.2 FileInputStream

This stream is called as input straemas is used for reading data from the files. Objects of FileInputStream can be created using the keyword **new** and there are different types of constructors available.

Sr.No.	Methods& Descriptions
1	FileInputStream("C:/java/hello");This constructor takes string as a parameter to create an input stream object to write the file. The string can representsa file name.
2	FileInputStream(f): constructor takes a file object to create an input stream object to read the file. File object can be created as follow File f = new File("C:/java/hello");

Once we create have InputStream object, and then list of helper methods provided by class can be used to read to stream or to do other operations on the stream.

7.4.2.1 Methods of FileInputStream Class:

Table 7.4 : FileInputStream Class Methods

Sr.No.	Methods & Descriptions
1	public void close() throws IOException{}: This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {}: This method cleans up the connection to the file. When there are no more references to this stream, then close method of this file output stream is called. Throws an IOException.
3	public int read(int r)throws IOException{}: This method reads the specified byte of data from the InputStream. Returns the next byte of data and if it's the end of the file then will return -1. Having int as return type.
4	public int read(byte[] r) throws IOException{}: This method reads r.length bytes from the input stream into an array. Returns the next byte of data and if it's the end of the file then will return -1. Having int as return type.
5	public int available() throws IOException{}: Gives the number of bytes that can be read from this file input stream. Returns an int.

Example:

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }          }
```

```
Output:Welcome to java Files
```

7.5 Java BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. Efficient writing of various objects like single arrays, characters, and stringsprovided by the buffering characters.

7.5.1 Constructor of BufferedWriterClass

Table 7.5 : BufferedWriter Class Constructors

Constructor	Description
BufferedWriter(Writer wrt)	It is used to create a buffered character output stream that uses the default size for an output buffer.
BufferedWriter(Writer wrt, int size)	It is used to create a buffered character output stream that uses the specified size for an output buffer.

7.5.2 Methods of BufferedWriterClass

Table 7.6 : BufferedWriter Class Methods

Method	Description
void newLine()	This method adds a new line by writing a line separator.
void write(int c)	This method is used to write a single character.
void write(char[] cbuf, int off, int len)	This method writes a portion of an array of characters.
void write(String s, int off, int len)	This method is used to write a portion of a string.
void flush()	This methodused to flushes the input stream.
void close()	This method is used to closes the input stream

Example: Program to Demonstrate some methods of BufferedWriter class

```
import java.io.*;

public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
   FileWriter writer = new FileWriter("D:\\testout.txt");
```

```
    BufferedWriter buffer = new BufferedWriter(writer);
    buffer.write("Welcome to java Files.");
    buffer.close();
    System.out.println("Success");
    }
}
```

```
Output:success
The content of a text file testout.txt is set with the data Welcome to Java Files.
```

7. 6 Java BufferedReader Class

Java BufferedReader class read the text from a character-based input stream. To read data line by line it use thereadLine() method. It makes the performance fast. It inherits Reader class.

7.6.1 Java BufferedReader class declaration

Let's see the declaration for Java.io.BufferedReader class:

1. public class BufferedReader extends Reader

7.6.2 Java BufferedReader class constructors

Table 7.7 : BufferedReader Class Constructors

Constructor	Description
BufferedReader(Reader rd)	It is used to create a object of BufferedReader that can buffered character input stream that uses the default size for an input buffer.
BufferedReader(Reader rd, int size)	It is used to create a object of BufferedReader buffered character input stream that uses the specified size for an input buffer.

7.6.3 Java BufferedReader class methods

Table 7.8 : BufferedReader Class Methods

Method	Description
int read()	This methodis used for reading a single character.
int read(char[] cbuf, int off, int len)	This method is used for reading characters into a portion of an array.
boolean markSupported()	This methodtest the input stream support for the mark and reset method.
String readLine()	This methodused for reading a line of text.
boolean ready()	This method tests whether the input stream is ready to be read.
long skip(long n)	This methodskip the characters specified by position number passed to it.
void reset()	This method repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	To mark the present position in a stream mark method is used.
void close()	This method is used to releases any of the system resources associated with the stream and also closes the input stream.

Example: **In this example, we are reading the data from the text file** testout.txt **using Java BufferedReader class.**

```
package com.mypackage;
import java.io.*;
public class BufferedReaderExample {
   public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        BufferedReader br=new BufferedReader(fr);

        int i;
        while((i=br.read())!=-1){
        System.out.print((char)i);
        }
        br.close();
        fr.close();
   }
```

```
}
```

Output: Here, we are assuming that you have following data in "testout.txt" file:

Welcome to Java Files.

```
Output:

Welcome to java Files.
```

7.7 Reading data from console using InputStreamReader and BufferedReader

Let us consider the example to understand the BufferedReader and InputStreamReader, Let us connect the BufferedReader stream with the InputStreamReader stream, so that it can read the line by line data from the keyboard.

```
package com.mypackage;
import java.io.*;
public class BufferedReaderExample{
public static void main(String args[])throws Exception{
   InputStreamReader r=new InputStreamReader(System.in);
   BufferedReader br=new BufferedReader(r);
   System.out.println("Enter your name");
   String name=br.readLine();
   System.out.println("Welcome "+name);
}
}
```

Output:

```
Enter your name
Rama
Welcome Rama
```

7.8 Serialization and Deserialization in Java

Java Serialization is a mechanism used to write the state of an object into a byte-stream. The exact reverse process of serialization is called deserialization, which convertbyte-stream into an object. These both processes are knows because of their platform-independent feature, it means an object serealized on one platform can be deserialized in different platform. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

One need to call the writeObject() method of ObjectOutputStream for serializing the object, and for deserialization call the readObject() method of ObjectInputStream class.

7.8.1 Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.
3. After Serializing anyObject it can be stored in File or database and can be desterilized when ever it isnecessary.

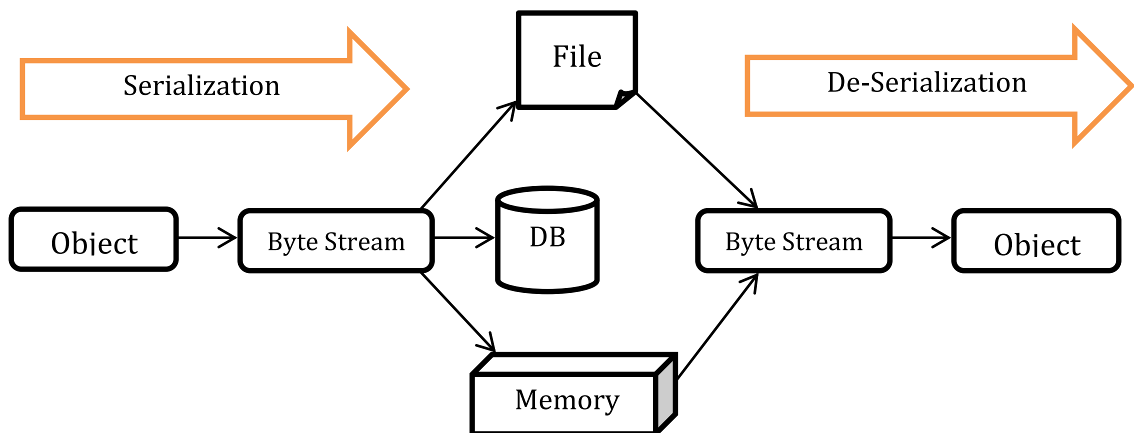


Fig. 7.3(a): Serialization and Deserialization

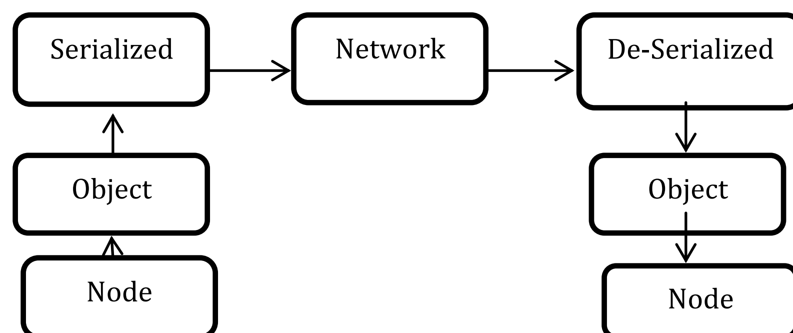


Fig. 7.3(b): Serialization and Deserialization

Only the objects of those classes can be serialized which are implementing **java.io.Serializable** Interface.

7.8.2 java.io.Serializable interface

Serializable is a marker interface that means it does not have data member and method. It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.The class whose object you want to persist, mult implement this interface.All the wrapper classes as well as String class implement the *java.io.Serializable* interface by default.

7.9 ObjectOutputStream class

To write primitive data typesObjectOutputStream class is used, and Java objects to write an OutputStream. The objects can be written to streams if that support the java.io.Serializable interface.

7.9.1 Constructors of ObjectOutputStream class

Table 7.9 : ObjectOutputStream Class Constructors

Constructor	**Description**
public ObjectOutputStream(OutputStream out) throws IOException {}	This Constructor creates an instance of ObjectOutputStream that writes to the specified OutputStream.

7.9.2 Important MethodsObjectOutputStream class

Table 7.10 : ObkectOutputStream Class Methods

Method	**Description**
public final void writeObject(Object obj) throws IOException {}	This method is used to writes the specified object to the ObjectOutputStream.
public void flush() throws IOException {}	This method is used to flush the current output stream.

public void close() throws IOException {}	closes the current output stream.

7.10 ObjectInputStream class

An ObjectInputStream can be used to deserialize the objects and primitive data written using an ObjectOutputStream.

7.10.1 Constructors of ObjectInputStream class

Table 7.11 : ObjectInputStream Class Constructors

Constructor	Description
public ObjectInputStream(InputStream in) throws IOException {}	This Constructor creates an ObjectInputStream that reads from the specified InputStream.

7.10.2Important Methods of ObjectInputStream class

Table 7.12 : ObjectOutputStream Class Methods

Method	Description
public final Object readObject() throws IOException, ClassNotFoundException{}	This method reads an object from the input stream.
public void close() throws IOException {}	This method closes ObjectInputStream.

Let's see the example given below:

```
import java.io.Serializable;
public class Employee implements Serializable{
 int id;
 String name;
 public Employee  (int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

In the above example we can observe and understand thatEmployee class is implementing Serializable interface. Now its objects areable to convert into stream.

7.11 Example of Java Serialization

Let us consider the example, to serialize the object of Employee class, for this purpose usewriteObject() method of ObjectOutputStream class which allows to serialize the object. We are saving the state of the object in the file named f.txt.

```
import java.io.*;
class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Employees1 =new Employee(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
Output: Object is serialized Successfully
```

7.12 Example of Java Deserialization

Deserialization is the exact recerse process of serialization, which reconstruct the object from the serialized state. It is the reverse operation of serialization. Let's us try to understand the concept with example where we are reading the data from a deserialized object.

```
import java.io.*;
```

```
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Employeee=(Employee)in.readObject();
  //printing the data of the serialized object
  System.out.println("EmployeeID:"+e.id);
System.out.println("EmployeeName:  "+e.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
Output:
EmployeeID: 11
Employeename: Rama
```

7.13 Java Scanner

Scanner class in Java is part of java.util package. In Java the java.util.Scanner class is one of themethodto read input from the keyboard.The Java Scanner class divide the input into diffrent tokens with the help of a delimiter which is whitespace by default. To read and parse various primitive values various methods are offered by it.The Java Scanner class uses a regular expression to parse text for strings and primitive types. Java Scanner is the easiest way to accept input in Java. Scanner in Java support por help us to get input from the user in primitive types such as int, long, double, byte, float, short, etc.The Java Scanner class has Object class as Super class and this class also inherit the interfaces Iterator and Closeable by implementing them.

The Java Scanner class provides different version of nextXXX() methods to return the type of value such asnextInt(), nextByte(), nextLine(), nextDouble(), nextFloat(), nextBoolean(),nextShort(), next(), etc. Some methods from the scanner class, such asnext().charAt(0) can be used which returns a single character.

7.13.1 Java Scanner Class Constructors

Table 7.13 : Scanner Class Constructors

SN	Constructors	Description
1)	Scanner(File source)	It is used to produce values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs aobject of new Scanner to produce values scanned from the specified file.
3)	Scanner(InputStream source)	It usedto produce values scanned from the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It is used to produce values scanned from the specified input stream this constructor is used.
5)	Scanner(Readable source)	This constructor produce values scanned from the Sourse specified in parameter.
6)	Scanner(String source)	This constructor produce values scanned from the String specified in parameter.
7)	Scanner(ReadableByteChannel source)	This constructor produce values scanned from the channel specified in parameter.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs aobject of new Scanner to produce values scanned from the specified channel.
9)	Scanner(Path source)	This constructor produce values scanned from the file specified in parameter.
10)	Scanner(Path source, String charsetName)	To produce values scanned from the specified file this constructor can be used.

7.13.2 Java Scanner Class Methods

The following are the list of Scanner methods:

Table 7.14 : Scanner Class Methods

SN	Methods	Description
1)	void close()	It is used to close this scanner.
2)	pattern delimiter()	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
3)	Stream<MatchResult> findAll()	It is used to find a stream of match results that match the provided pattern string.
4)	String findInLine()	Can be used To find the next occurrence of a pattern constructed from the specified string, ignoring delimiters,

5)	string findWithinHorizon()	This method is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean hasNext()	If this scanner has another token in its input then this method is used to returns true
7)	boolean hasNextBigDecimal()	This mrthod used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
8)	boolean hasNextBigInteger()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
9)	Boolean hasNextBoolean()	This method is used to check if the next token in this scanner's input can be interpreted as a Boolean using the nextBoolean() method or not.
10)	boolean hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte using the nextBigDecimal() method or not.
11)	boolean hasNextDouble()	To check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean hasNextFloat()	This method is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean hasNextInt()	To check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean hasNextLine()	This method is to check if there is another line in the input of this scanner or not.
15)	boolean hasNextLong()	This method is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
16)	boolean hasNextShort()	This method isused to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.
17)	IOException ioException()	It is used to return the IOException last thrown by this Scanner's readable.
18)	Locale locale()	It is used to get a Locale of the Scanner class.
19)	MatchResult match()	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String next()	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal nextBigDecimal()	This method is used toacceptthe next token of the input as a BigDecimal.
22)	BigInteger nextBigInteger()	This method is used toacceptthe next token of the input as a BigInteger.
23)	boolean nextBoolean()	This method is used toacceptthe next token of the input into a boolean value and returns that value.

24)	byte nextByte()	This method is used toacceptthe next token of the input as a byte.
25)	double nextDouble()	This method is used toacceptthe next token of the input as a double.
26)	float nextFloat()	This method is used to acceptthe next token of the input as a float.
27)	int nextInt()	This method is used toacceptthe next token of the input as an Int.
28)	String nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	long nextLong()	This method is used toacceptthe next token of the input as a long.
30)	short nextShort()	This method is used to accept the next token of the input as a short.
31)	int radix()	This method is used to get the default radix of the Scanner use.
32)	void remove()	when remove operation is not supported by this implementation of Iterator then this method can be used.
33)	Scanner reset()	This method is used to reset the Scanner which is in use.
34)	Scanner skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.
36)	String toString()	It is used to get the string representation of Scanner using.
37)	Scanner useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner useLocale()	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner useRadix()	It is used to set the default radix of the Scanner .

Example 1: Let's see a simple example of Java Scanner where we are getting a single input from the user. Here, we are asking for a string through in.nextLine() method.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
```

```
        in.close();
        }
}

Output:
Enter your name: sonoo jaiswal
Name is: sonoo jaiswal
```

Example 2:

```
import java.util.*;
public class ScannerClassExample1 {
    public static void main(String args[]){
        String s = "Hello, This is JavaTpoint.";
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " +scan.nextLine());
        scan.close();
        System.out.println("--------Enter Your Details-------- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name: " + name);
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
        in.close();
        }
}
```

```
Output:
Boolean Result: true
```

```
String: Hello, This is JavaTpoint.
-------Enter Your Details---------
Enter your name: Abhishek
Name: Abhishek
Enter your age: 23
Age: 23
Enter your salary: 25000
Salary: 25000.0
```

Outcomes:

- Students will able to study file class in java, its constructors and methods in java.
- Students will able to study working of read and write files using stream in java.
- Students will able to study FileiInputStream class, FileiOutputStream class,their constructors and methods.
- Students will able to study BufferReader class, BufferWriter class,its constructors and methods.
- Students will able to study serialization and deserialization in java.

Questions:

1. Write a short note on file class in java?
2. Explain some of the important methods of file class with suitable example?
3. Define stream? List and explain various types of stream?
4. write a short note on file InputStream class?
5. List and explain in short various classes related to Inputstream and OutputStream classes.
6. What is FileOutputStream? What is the purpose of it in java?
7. Compare FileOutputStream and FileInputStream.
8. Explain the need of BufferedReader class in java. List the important methods in BufferedReader class.
9. What is serialization in java? What are the advantages of serialization?
10. With neat diagram explain the concept of serialization and deserialization.
11. Write a short note on ObjectOutputStream class to explain the need of it in java.
12. Write a short note on ObjectInputStream class to explain the need of it in java.
13. Write a java program to understand the concept of java serialization.
14. Write a java program to understand the concept of java deserialization.
15. What is Scanner class in java? To which package this class belongs? List and explain some of the important methods of scanner class

CHAPTER-8

THREAD, GENERICS AND COLLECTIONS

Objectives:

- To study threading, multithreading, threadcreation in java.
- To study lifecycle of thread and various states in that cycle of thread in java.
- To study thread class, constructors of thread class and its methods in java.
- To study synchronization in java, java genrics, its methods and class, java collections, its methods and classes, etc.

8.1 Introduction to Threading

A thread is a process or task in execution. The Java Virtual Machine allows an application to execute multiple threads concurrently. Such as paint, calculator, Microsoft word or any process running on computer is a thread.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread which is currently executing may or may not be marked as a daemon. When a new `Thread` object is created, the priority of new thread is initially set equal to the priority of the creating thread.A newly created thread is a daemon thread if and only if the creating thread is a daemon.

8.2 Multithreading in Java

Multithreading is a Java feature that allows two or more part of program called as thread to execute concurrently or parallelly for maximum utilization of CPU. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:

1. Extending the Thread class

2. Implementing the Runnable Interface

8.1.1 Thread creation by extending the Thread class:

Thread class can be created by extending the java.lang.Thread class. The class extending thisclass need to override the run() method available in the Thread class. A thread begins its life inside run() method. To start the execution of a thread, need tocreate an object of our new class and call start() method. The start() method is used to invokes the run() method on the Thread object.

Example: Java code for thread creation by extending the Thread class

```
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");
          }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
```

```
}
```

Output:

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

8.1.2 Thread creation by implementing the Runnable Interface:

Thread can also be created in a new class by implementing java.lang.Runnable interface and override run() method. Then a Thread object need to initiate to call start() method on this object.

Example: Java code for thread creation by implementing the Runnable Interface

```
class MultithreadingDemo implements Runnable
{
 public void run()
   {
       try
       {
           // Displaying the thread that is running
           System.out.println ("Thread " +
                               Thread.currentThread().getId() +
                               " is running");
         }
       catch (Exception e)
       {
           // Throwing an exception
           System.out.println ("Exception is caught");
       }
   }
```

```
}
 // Main Class
class Multithread
{
   public static void main(String[] args)
   {
      int n = 8; // Number of threads
      for (int i=0; i<n; i++)
      {
         Thread object = new Thread(new MultithreadingDemo());
         object.start();
      }
   }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread
11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is runningThread 15 is running
```

8.3 Lifecycle and States of a Thread in Java

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. Born
2. Ready
3. Running
4. Blocked
5. Waiting
6. Timed Waiting (Sleeping)

7. Suspended
8. Dead

The diagram shown below represents various states of a thread at any instant of time.

8.3.1 New Thread (Born):

When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

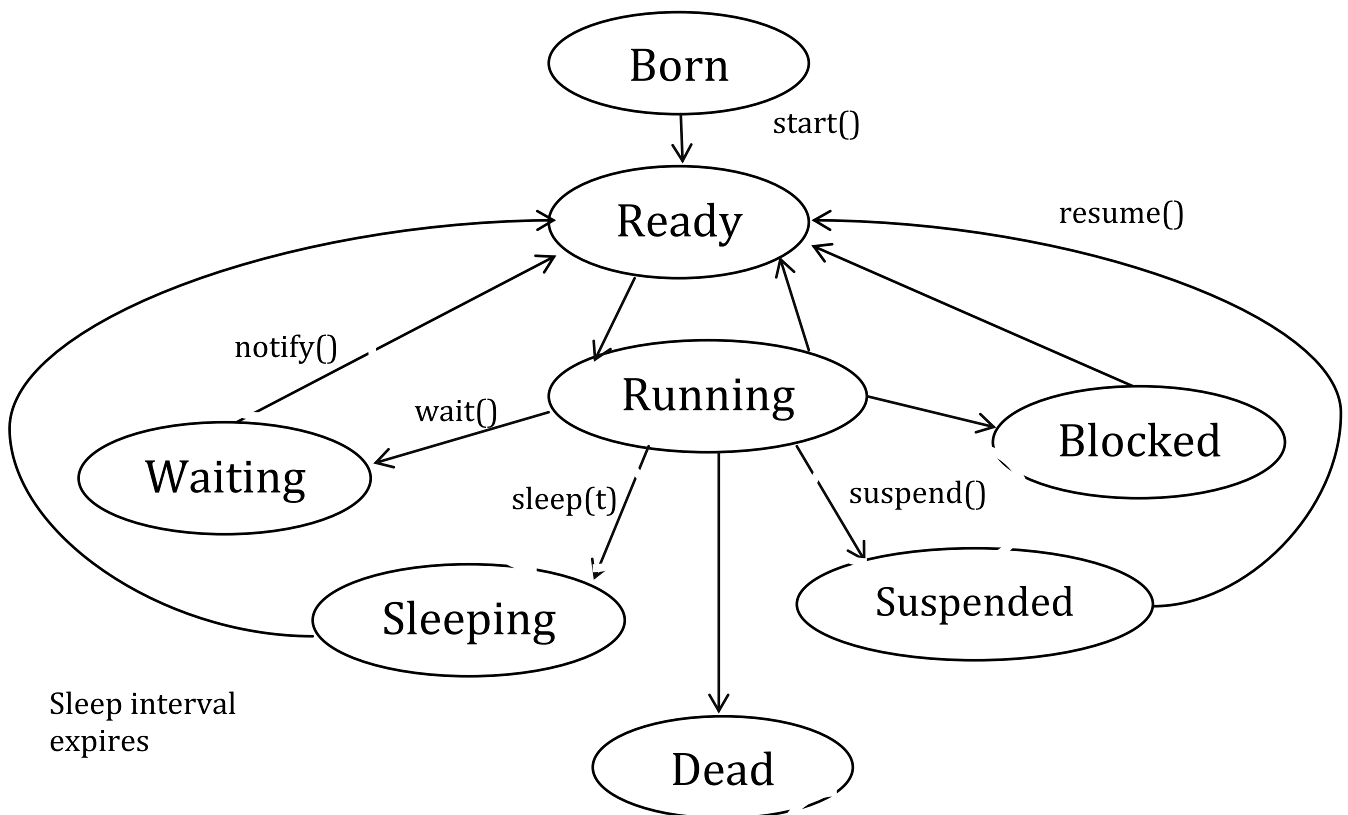


Fig. 8.1 : Life Cycle of a thread

8.3.2 Runnable State (Ready):

A thread that is already in ready to run state after calling start() function is moved to runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. Thread scheduler takes theresponsibility to assign a time to thread to run. Thread enter in to runnable state after calling start() function from thread class.

8.3.3 Running:

When run() function is called from thread function is enter in to running state. Each individual thread gets assigned a fixed amount of time to execute by a multi-threaded program. Each and every thread runs for a short while and then pauses and surrenders the CPU to another thread, so that other threads can get a chance to run. After this, all threads that are ready to run, waiting for the CPU are come into the running state.

When a thread is not in executionor temporarily inactive, then it may lie one of the following states:

8.3.4 Blocked:

Thread lies in the blocked state,whenit is waiting for I/O to complete its execution. Thread scheduler takes the responsibility to reactivate and schedule a blocked thread. A thread which is currently blocked and is in this state, that cannot continue its execution any further until I/O resourses get available and it is moved to runnable state. Any thread in these states does not consume any CPU cycle. When the I/O recourses get available then thread enters into ready to run state.

Onather reason which causes a thread to enter in the blocked state is when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks and moves any of the thread to runnable state.

8.3.5Waiting State:

A thread is in the waiting state when another thread makes it wait on some condition. Another thread may call wait() function. When this condition is satisfied, the scheduler is informed by notify() function and the waiting thread is moved to runnable state.

If any of the currently running thread is moved to blocked/waiting state, another thread waiting to run in runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

8.3.6 Timed waiting (sleeping):

A thread enters in timed waiting state when it calls a method sleep(t) with a time out parameter in nanoseconds. A thread remains in this state till the timeout is completed or until a notification is received. In simple words when a thread calls sleep or a conditional wait, it is entered in to a timed waiting state.

8.3.7 Suspended State:

A thread is in the suspended state when it is suspended for no any reason and no any time limit using a suspend() function. When this condition is fulfilled, resume() function can be called and the suspended thread is moved to runnable state.

8.3.8 Terminated (Dead) State:

A thread may terminate its execution because of any of the following reasons:

- Because it exists normally. This occurs when the code of thread has completely executed by the program.
- Because of some unusual specious event occured, such as segmentation fault or an unhandled exception.
- A thread that lies in a terminated state does no longer consume any cycles of CPU.

8.4 Constructors in Thread Class:

Table 8.1 : Thread Class Constructors

Sr. No.	Constructor & Descriptions
1	**Thread():** Allocates a new Thread object
2	**Thread(Runnable target):** Allocates a new Thread object
3	**Thread(Runnable target, String name):**It will allocates a new Thread object
4	**Thread(String name):** Allocates a new Thread object
5	**Thread(ThreadGroup group, Runnable target):**It will allocates a new Thread object
6	**Thread(ThreadGroup group, Runnable target, String name**): It will allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group
7	**Thread(ThreadGroup group, Runnable target, String name, long stackSize):**It will allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size
8	**Thread(ThreadGroup group, String name):** CAllocates a new Thread object

8.5 Methods in Thread Class:

Table 8.2 : Thread Class Methods

Sr. No.	Method & Descriptions
1	**activeCount():**This method returns an estimate of the number of active threads in the current thread's thread group and its subgroups
2	**checkAccess():**This method determines if the currently running thread has permission to modify this thread
3	**clone():**Throws CloneNotSupportedException as a Thread can not be meaningfully cloned
4	**currentThread():**This method returns a reference to the currently executing thread object
5	**dumpStack():**It will Prints a stack trace of the current thread to the standard error stream
6	**getAllStackTraces():** This method returns a map of stack traces for all live threads
7	**getId():** Returns the identifier of this Thread
8	**getName():**Returns this thread's name
9	**getPriority():**Returns this thread's priority
10	**getStackTrace():**This method Returns an array of stack trace elements representing the stack dump of this thread
11	**getState():**Returns the state of this thread
12	**getThreadGroup():**This method Returns the thread group to which this thread belongs
13	**interrupt():**Interrupts this thread
14	**interrupted():**Tests whether the current thread has been interrupted
15	**isAlive():**Tests if this thread is alive
16	**join():** Waits for this thread to join
17	**run():**If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns
18	**yield():**A hint to the scheduler that the current thread is willing to yield its current use of a processor
19	**toString():** This method Returns a string representation of this thread, including the thread's name, priority, and thread group
20	**start():** This method Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread
21	**sleep(long millis):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers

22	**setPriority(int newPriority): j** Changes the priority of this thread
23	**setPriority(int newPriority):** Changes the priority of this thread
24	**Suspend():**
25	**Wait():**

8.6 Java Thread Priority in Multithreading

In a Multi-threading, thread scheduler assigns processor to a thread based on priority of thread. Every thread createed in Java has some priority assigned to it. This Priority can either be assigned by JVM while creating the thread or it can be given by programmer explicitly.

The Valid value of priority for a thread is in between 1 to 10. The diffrent static variables defined in Thread class for priority are of 3 types.

public static int MIN_PRIORITY: This indicate minimum priority that a thread can have. Value for this 1.

public static int NORM_PRIORITY: This indicatedefault priority of a thread if do not explicitly define it. Value for this is 5.

public static int MAX_PRIORITY: This indicatemaximum priority of a thread. Value for this is 10.

Get and Set Thread Priority:

1. public final int getPriority(): java.lang.Thread.getPriority() method returns priority ofgiven thread.

2. public final void setPriority(int newPriority): java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Examples of getPriority() and set

```
// Java program to demonstrate getPriority() and setPriority()
import java.lang.*;
  class ThreadDemo extends Thread
{
    public void run()
```

```
    {
        System.out.println("Inside run method");
    }
    public static void main(String[]args)
    {
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();
          System.out.println("t1 thread priority : " +  t1.getPriority()); // Default 5
        System.out.println("t2 thread priority : " +  t2.getPriority()); // Default 5
        System.out.println("t3 thread priority : " +   t3.getPriority()); // Default 5
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        System.out.println("t1 thread priority : " +  t1.getPriority());
        System.out.println("t2 thread priority : " +  t2.getPriority());
        System.out.println("t3 thread priority : " +  t3.getPriority());
          // Main thread
        System.out.print(Thread.currentThread().getName());
        System.out.println("Main thread priority : "  + Thread.currentThread().getPriority());
          // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority : "  + Thread.currentThread().getPriority());
    }
}
```

```
Output:
t1 thread priority : 5
```

```
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Main thread priority : 5
Main thread priority : 10
```

8.7 Synchronization in Java:

Multi-threaded programs may often enter in to a situation where multiple threads try to access the same resources and finally produce invalid and unforeseen results.

So synchronization method nned to make confirm that only one thread can access the resource at a given point of time.

Java provides steps and ensured method of creating threads and synchronizing their task by using synchronized blocks. The synchronized keyword can be used in jave to define Synchronized. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads struggling to enter the synchronized block are remains in waiting until the thread inside the synchronized block exits the block.

```
Following is the general form of a synchronized block:
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
```

```
}
```

In Java synchronization is implemented with a concept called monitors.A monitorcan be owed by only one thread 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.

Following is an example of multi-threading with synchronized.

Example: A Java program to demonstrate working of synchronized.

```
import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
    public void send(String msg)
    {
        System.out.println("Sending\t"  + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread  interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}
```

Class for send a message using Threads

```
class ThreadedSend extends Thread
{
    private String msg;
    Sender  sender;
     // Recieves a message object and a string message to be sent
    ThreadedSend(String m,  Sender obj)
    {
        msg = m;
        sender = obj;
    }
     public void run()
    {
        // Only one thread can send a message at a time.
        synchronized(sender)
        {
            // synchronizing the snd object
            sender.send(msg);
        }
    }
}
```

```
// Driver class
class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );
          // Start two threads of ThreadedSend type
        S1.start();
```

```
        S2.start();
          // wait for threads to end
        try
        {
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

```
Output:
Sending     Hi
 Hi Sent
Sending     Bye
 Bye Sent
```

8.8 Java Generics (Generic Methods and Classes):

Java Genericis is an advanced feature of java that can be implemented with methods and classes called as generic method and generic class respectively. Generic methods and generic classes permit programmers to write a single method declaration, which can be used to specify a set of related methods, and write a single class declaration to specify, a set of related types of classes.
Generics in java also offer compile-time type safety, whichpermits programmers to catch invalid types at compile time.
Generics in Java are exactlyanalogous to templates in C++. The Concept is to allow type (Integer, String, etc. and user defined types) to be a parameter to methods, classes and interfaces.

Java Generics are a language feature that allows fthe programmer to define and use of generic types classes and methods." Generic types are instantiated to form parameterized types by providing actual type arguments that replace the formal type parameters. A class like ArrayList<E> is a generic type, that has a type parameter E.

8.8.1 Advantage of Java Generics

1) Type-safety: It is a feature of generic that doesn't allow storing other objects; we can hold only a single type of objects in generics.
2) Type casting is not required: Typecasting of object is not required.
3) **Compile-Time Checking:** The type of generic parameter is checked at compile time so problem will not occur at runtime.To handle the problem at compile time than runtime is the good programming strategy.
4) Code Reusability: it allows code reusability as you can reuse the same code for different Types of data.

8.9 Generic Methods

Method that can be called with arguments of different types is called as generic method; the parameters in generic method can be replaced by any types. The compiler identifies and handles diffrent method call appropriately, Based on the types of the parameters passed to the generic method. Let us discuss some of the rules to define Generic Methods −

1. The generic method declarations have a argument type which is delimited in angle brackets (< and >) and method is precedesby return type (< E > in the next example).

2. parameter section contains one or more no of parameters with different types separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

3. The type parameters can be used to act as placeholders for the types of the arguments passed to the generic method as well cn also be used to declare the return type, which are also known as actual type arguments.

4. A generic method's body is declared just like that of any normal method. Remember that type parameters can represent only reference types, not primitive types (like int, double and char).

Example: Following example illustrates how we can print an array of different type using a single Generic method –

```
publicclassGenericMethodTest{
// generic method printArray
publicstatic< E >void printArray( E[] inputArray ){
// Display array elements
for(E element : inputArray){
System.out.printf("%s ", element);
}
System.out.println();
}
```

```
publicstaticvoid main(String args[]){
// Create arrays of Integer, Double and Character
Integer[] intArray ={1,2,3,4,5};
Double[] doubleArray ={1.1,2.2,3.3,4.4};
Character[] charArray ={'H','E','L','L','O'};

System.out.println("Array integerArray contains:");
    printArray(intArray);// pass an Integer array

System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);// pass a Double array

System.out.println("\nArray characterArray contains:");
    printArray(charArray);// pass a Character array
}
}
```

```
Output
Array integer Array contains: 1 2 3 4 5
```

```
Array double Array contains:  1.1 2.2 3.3 4.4

Array character Array contains: H E L L O
```

8.10 Generic Classes

A generic class declaration is just similar to look like a non-generic class declaration, But just the difference is that the class name is followed by a type parameter section.
In generic Class, the type parameter section of a generic class we canspecify different parameters separated by commas. As these classes can accept one or more parametershence are known as parameterized classes or parameterized types.

Example-1: Following example illustrates how we can define a generic class −

```
public class Box<T> {

  private T t;

  public void add(T t) {

    this.t = t;

  }

  public T get() {

    return t;

  }

  public static void main(String[] args) {

    Box<Integer> integerBox = new Box<Integer>();

    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n", integerBox.get());

    System.out.printf("String Value :%s\n", stringBox.get());

  }

}
```

Output:

```
Integer Value :10

String Value :Hello World
```

Example-2: Following program implements the Generic class and generic methods.

```
class Generic<T , V>
{
        T ob;
        V ob1;
        Generic(T o, V o1)
        {
            ob=o;
            ob1=o1;
        }

        T getob()
        {
            return ob;
        }
        V getob1()
        {
            return ob1;
        }
        public static void main(String[] args)
        {
            Generic<Integer, String> gi=new Generic<Integer,String>(100,"Hiiii.......");
            System.out.println(gi.getob());
            System.out.println(gi.getob1());
            System.out.println("Hello World!");
            Generic<String,Double> gi1=new Generic<String,Double>("Vipin",123.9);
            System.out.println(gi1.getob());
            System.out.println(gi1.getob1());
        }
}
```

```
Output:   100                    Hiiii.......
```

Hello World!	Vipin	123.9

8.11 Collections in Java:

A collection in java, as name indicates, is assembly or group of objects. **Java Collections framework** is consist of the diffrent interfaces and classes which can be used in implementation with different types of data structures as a collections such as **lists, sets, maps, stacks and queues** etc.

These inbuild collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects. The besic operations offered by collection are **add, remove, update, sort, search** and more complex algorithms. These collection classes offereasy and very clearprovision for all such operations using **Collections APIs**.

8.11.1 Java Collections Hierarchy:

The Collections framework can be easyly understood with the help of **core interfaces**. The collections classes implement these interfaces and provide concrete functionalities.

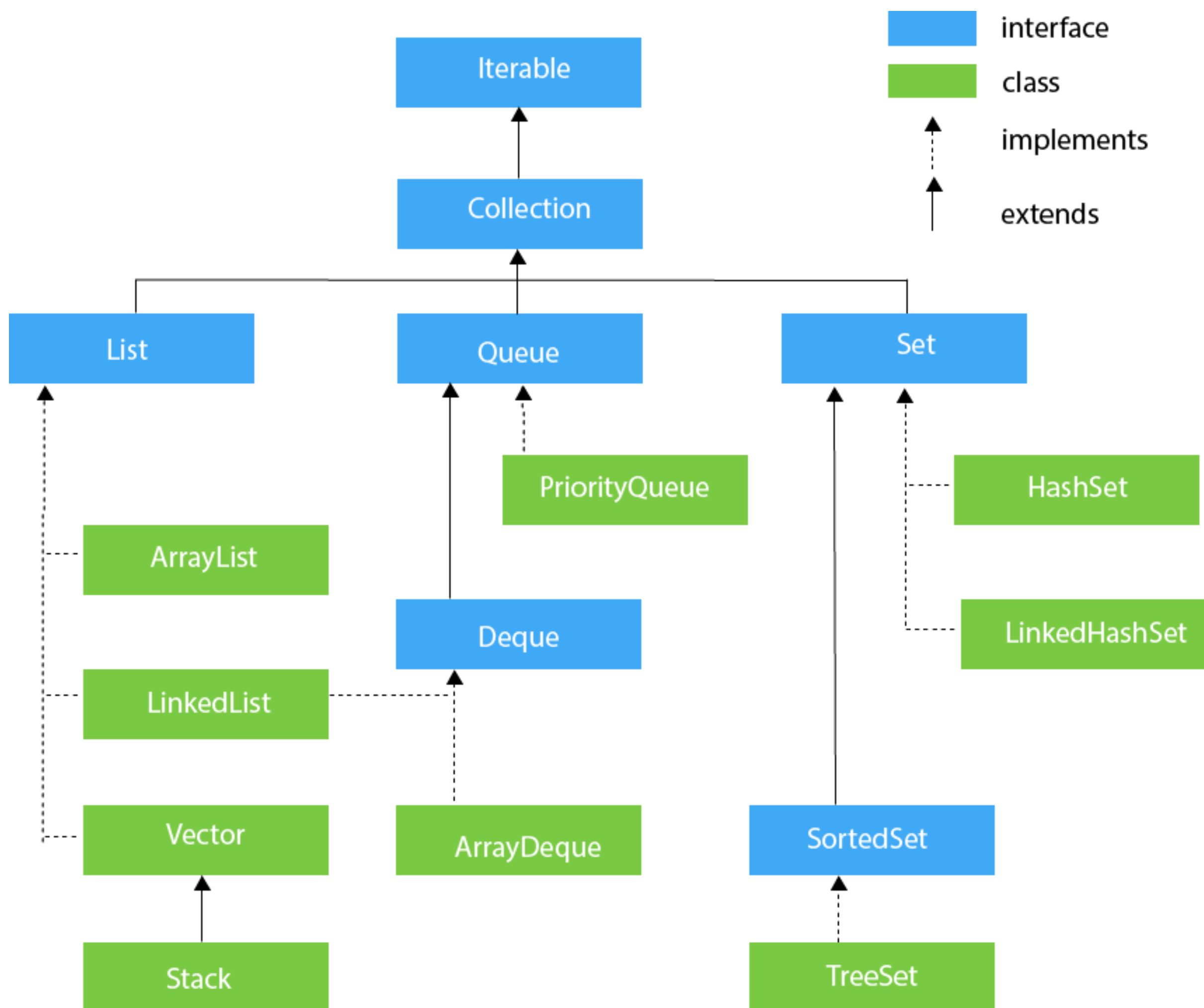


Fig.8.2: Java collections Hierarchy

8.11.2 Collection

Collection is an interface which is placed at the top or you can sayat the root of the hierarchy. Collection interface provides all besic methods which are supported by all collections classes (or throw UnsupportedOperationException). The Collection interface **extends Iterable** interface which enhances support for iterating over collection features using the "**for-each loop**" statement.

All other interfaces and classes otherthan Map in collection interface either extend or implement this interface. Let us consider with example, List *(indexed, ordered)* and Set *(sorted)* interfaces implement this collection.

8.11.3 List

Lists represent an **ordered** collection of elements. Lists, allows to access elements by their integer index which represent position in the list, and search for elements in the list. Index start with zero (0), just sililar to an array.

SoLet us see some different and useful classes which implement List interfaceare which are as - **ArrayList**, **CopyOnWriteArrayList**, **LinkedList**, **Stack** and **Vector**.

8.11.3 Set

Sets represent a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not guarantee to return the elements in any predictable order; though some of the Set implementations store elements in their natural ordering and guarantee this order.
SoLet us see some different and useful classes which implement Set interfaceare which are as - **ConcurrentSkipListSet**, **CopyOnWriteArraySet**, **EnumSet**, **HashSet**, **LinkedHashSet** and **T reeSet**.

8.11.4 Map

The **Map** interface stores the data in *key-value pairs*, where keys should be immutable. A duplicate key is not allowed in Map; each key can map to at most one value.

The Map interface provides three different views to map contents, which are a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Some of the map operations, like the TreeMap class, make particularassurances as to their order; others, like the HashMap class, do not.
Let us see some different and useful classes which implement Map interface which are as - **ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, Identity HashMap, LinkedHashMap**, **Properties, TreeMap** and **WeakHashMap.**

8.11.5 Stack

The Java **Stack** interface represents a classical stack data structure, where elements can be pushed to stack in last-in-first-out (LIFO) order. In Stack both push and pop operations are done from same end, that is push an element to the top of the stack, and pop element at top of the stack again later.

8.11.6 Queue:

A queue data structure is anticipated to contain the elements prior to processing by consumer thread(s). Moreover basic operations offered by collections, queues offersupplementary insertion, extraction, and inspection operations.

Queues characteristically, order elements in a FIFO (first-in-first-out) manner, but which is not necessarily in every situation. One of such situation or example is priority queue which order elements according to a supplied Comparator called as priority, or the elements' natural ordering.

In simple words we can say, queues do not support blocking insertion or retrieval operations. Blocking queue implementations classes implement **BlockingQueue** interface.
Some useful classes which implement Map interface are – ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueueand SynchronousQueue.

8.11.7 Dequeue:

A double ended queue is two ended queue queue that supports element insertion and removal at both ends. When a deque is implemented as a queue, FIFO (First-In-First-Out) behavior results.Deque can also be treated as a stack, LIFO (Last-In-First-Out) behavior results.

This interface should be used in preference to the legacy Stack class. When elements are inserted and removed from the beginning of the deque it can be treated as a stack.
Some of the commonly used classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.

8.11.8 ArrayList in Java

An ArrayList in Java represent a resizable list of objects, which means its size can be modified as per requirement. Arraylist allows us various operations like add, remove, find, sort and replace elements in this list. ArrayListimplements Java's List interfaceand it is a part of Java's collection framework and.

8.11.8.1 ArrayList Features: ArrayList has following features –

1. Ordered – Elements in arraylist preserve their ordering which is by default the order in which they were added to the list.
2. Index based – Elements can be randomly accessed using index positions. Index start with '0'.

3. Dynamic resizing – ArrayList grows dynamically when more elements needs to be added than its current size.
4. Non synchronized – ArrayList is not synchronized, by default. Programmer needs to use synchronized keyword appropriately or simply use Vector class.
5. Duplicates allowed – We can add duplicate elements in Arraylist. It is not possible in sets.

8.11.8.2 Constructors in ArrayList

Table 8.3 : ArrayList Constructors

Sr. No.	Constructors & Descriptions
1	ArrayList(): This constructor is used to build an empty array list
2	ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c
3	ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified

8.11.8.3 Methods in ArrayList

Table 8.4 : ArrayList Methods

Sr. No.	Methods & Descriptions
1	forEach(Consumer<? super E> action): Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
2	retainAll(Collection<?> c): Retains only the elements in this list that are contained in the specified collection.
3	removeIf(Predicate<? super E> filter): Removes all of the elements of this collection that satisfy the given predicate.
4	contains(Object o): if this list contains the specified element then this method returns true.
5	remove(int index): This method removes the element at the specified position in this list.
6	remove(Object o): This method Removes the first occurrence of the specified element from this list, if it is present.
7	get(int index): This method Returns the element at the specified position in this list.
8	subList(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
9	spliterator(): This method Creates a late-binding and fail-fast Spliterator over the elements in this list.
10	set(int index, E element): This method Replaces the element at the specified position in this list with the specified element.
11	size(): Returns the number of elements in this list.
12	removeAll(Collection<?> c): Removes from this list all of its elements that are contained in the specified collection.
13	listIterator(): Returns a list iterator over the elements in this list (in proper sequence).
14	listIterator(int index): Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

15	isEmpty(): Returns true if this list contains no elements.
16	void clear(): This method is used to remove all the elements from any list.
17	void add(int index, Object element): This method is used to insert a specific element at a specific position index in a list.
18	void trimToSize(): This method is used to trim the capacity of the instance of the ArrayLis to the list’s current size.
19	Object[] toArray(): This method is used to return an array containing all of the elements in the list in correct order.
20	Object clone(): This method is used to return a shallow copy of an ArrayList.
21	boolean addAll(int index, Collection C): Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list
22	boolean add(Object o): This method is used to append a specificd element to the end of a list.
23	boolean addAll(Collection C): This method is used to append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection’s iterator.
24	int indexOf(Object O): The index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.

Example: Java program to demonstrate working of ArrayList in Java

```
import java.io.*;

import java.util.*;

class arrayli

{

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

    {

        // size of ArrayList

        int n = 5;

          //declaring ArrayList with initial size n

        ArrayList<Integer> arrli = new ArrayList<Integer>(n);



        // Appending the new element at the end of the list

        for (int i=1; i<=n; i++)

            arrli.add(i);

          // Printing elements
```

```
        System.out.println(arrli);

          // Remove element at index 3

        arrli.remove(3);

          // Displaying ArrayList after deletion

        System.out.println(arrli);

          // Printing elements one by one

        for (int i=0; i<arrli.size(); i++)

            System.out.print(arrli.get(i)+" ");

    }

}
```

Output**:**

```
[1, 2, 3, 4, 5]

[1, 2, 3, 5]

1 2 3 5
```

8.11.9 Linked List in Java

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

8.11.9.1 LinkedList Features

1. **Doubly linked list** implementation which implements List and Deque interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
2. Permits all elements including duplicates and NULL.
3. LinkedList maintains the **insertion order** of the elements.
4. It is **not synchronized**. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.
5. Use Collections.synchronizedList(new LinkedList()) to get synchronized linkedlist.
6. The iterators returned by this class are fail-fast and may throw concurrentModificationException.
7. It does not implement RandomAccess interface. Hence it allows to access elements in sequential order only, and does not support accessing elements randomly.
8. We can use ListIterator to iterate LinkedList elements

8.11.9.2 Constructors in LinkedList class.

Table 8.5 : LinkedList Class Constructors

Sr. No.	Constructor & Description
1	LinkedList( ): This constructor builds an empty linked list.
2	LinkedList(Collection c): This constructor builds a linked list that is initialized with the elements of the collection c.

8.11.9.3 Methods in LinkedList class

Table 8.6 : LinkedList Class Methods

<table>
<tr><th>Sr. No.</th><th>Method & Description</th></tr>
<tr><td>1</td><td>void add(int index, Object element): Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).</td></tr>
<tr><td>2</td><td>boolean add(Object o): Appends the specified element to the end of this list.</td></tr>
<tr><td>3</td><td>boolean addAll(Collection c): Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.</td></tr>
<tr><td>4</td><td>boolean addAll(int index, Collection c): Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.</td></tr>
<tr><td>5</td><td>void addFirst(Object o): Inserts the given element at the beginning of this list.</td></tr>
<tr><td>6</td><td>void addLast(Object o): Appends the given element to the end of this list.</td></tr>
<tr><td>7</td><td>void clear(): Removes all of the elements from this list.</td></tr>
<tr><td>8</td><td>Object clone(): Returns a shallow copy of this LinkedList.</td></tr>
<tr><td>9</td><td>boolean contains(Object o): Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).</td></tr>
<tr><td>10</td><td>Object get(int index): Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size())</td></tr>
<tr><td>11</td><td>Object getFirst(): Returns the first element in this list. Throws NoSuchElementException if this list is empty.</td></tr>
<tr><td>12</td><td>Object getLast(): Returns the last element in this list. Throws NoSuchElementException if this list is empty.</td></tr>
<tr><td>13</td><td>int indexOf(Object o): Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.</td></tr>
<tr><td>14</td><td>int lastIndexOf(Object o): Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.</td></tr>
</table>

15	ListIterator listIterator(int index): Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
16	Object remove(int index): Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.
17	boolean remove(Object o): Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
18	Object removeFirst(): Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
19	Object removeLast(): Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.
20	Object set(int index, Object element): Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
21	int size(): Returns the number of elements in this list.
22	Object[] toArray(): Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
23	Object[] toArray(Object[] a): Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Example: The following program illustrates several of the methods supported by LinkedList

```
import java.util.*;

publicclassLinkedListDemo{

publicstaticvoid main(String args[]){

// create a linked list

LinkedList ll =newLinkedList();

// add elements to the linked list

    ll.add("F");

    ll.add("B");

    ll.add("D");

    ll.add("E");

    ll.add("C");

    ll.addLast("Z");
```

```
    ll.addFirst("A");

    ll.add(1,"A2");

System.out.println("Original contents of ll: "+ ll);

// remove elements from the linked list

    ll.remove("F");

    ll.remove(2);

System.out.println("Contents of ll after deletion: "+ ll);

// remove first and last elements

    ll.removeFirst();

    ll.removeLast();

System.out.println("ll after deleting first and last: "+ ll);

// get and set a value

Object val = ll.get(2);

    ll.set(2,(String) val +" Changed");

System.out.println("ll after change: "+ ll);

}

}
```

This will produce the following result −

```
Output

Original contents of ll: [A, A2, F, B, D, E, C, Z]

Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]

ll after change: [A2, D, E Changed, C]
```

8.11.10 Vector Class:

The java.util.Vector class implements a growable array of objects. Similar to an Array, it contains components that can be accessed using an integer index.

8.11.10.1 Features of Vector Class

- The Vectors is scalable that is size of a Vector can raise or shrink as desired to accommodate adding and removing items.
- Each vector tries maintaining a *capacity* and a capacityIncrement to maintain the memory optimization.
- As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface.
- Unlike the new collection implementations, *Vector* is synchronized.
- This class is a member of the Java Collections Framework.

8.11.10.2 Vector Class constructors

Table 8.7 : Vector Class Constructors

Sr. No.	Constructor & Description
1	Vector(): This constructor is used to create an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
2	Vector(Collection<? extends E> c): This constructor is used to create a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
3	Vector(int initialCapacity): This constructor is used to create an empty vector with the specified initial capacity and with its capacity increment equal to zero.
4	Vector(int initialCapacity, int capacityIncrement): This constructor is used to create an empty vector with the specified initial capacity and capacity increment.

8.11.10.3 Vector Class methods

Table 8.8 : Vector Class Methods

Sr. No.	Method & Description
1	boolean add(E e): This method appends the specified element to the end of this Vector.
2	void add(int index, E element): This method inserts the specified element at the specified position in this Vector.
3	boolean addAll(Collection<? extends E> c): This method appends all of the elements in the specified Collection to the end of this Vector.
4	boolean addAll(int index, Collection<? extends E> c): This method inserts all of the elements in the specified Collection into this Vector at the specified position.

5	void addElement(E obj): This method adds the specified component to the end of this vector, increasing its size by one.
6	int capacity(): This method returns the current capacity of this vector.
7	void clear(): This method removes all of the elements from this vector.
8	clone clone(): This method returns a clone of this vector.
9	boolean contains(Object o): This method returns true if this vector contains the specified element.
10	boolean containsAll(Collection<?> c): This method returns true if this Vector contains all of the elements in the specified Collection.
11	void copyInto(Object[ ] anArray): This method copies the components of this vector into the specified array.
12	E elementAt(int index): This method returns the component at the specified index.
13	Enumeration<E> elements(): This method returns an enumeration of the components of this vector.
14	void ensureCapacity(int minCapacity): This method increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	boolean equals(Object o): This method compares the specified Object with this Vector for equality.
16	E firstElement(): This method returns the first component (the item at index 0) of this vector.
17	E get(int index): This method returns the element at the specified position in this Vector.
18	int hashCode(): This method returns the hash code value for this Vector.
19	int indexOf(Object o): This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
20	int indexOf(Object o, int index): This method returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
21	void insertElementAt(E obj, int index): This method inserts the specified object as a component in this vector at the specified index.
22	boolean isEmpty(): This method tests if this vector has no components.
23	E lastElement(): This method returns the last component of the vector.
24	int lastIndexOf(Object o): This method returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
25	int lastIndexOf(Object o, int index): This method returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
26	E remove(int index): This method removes the element at the specified position in this Vector.
27	boolean remove(Object o): This method removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.

28	boolean removeAll(Collection<?> c): This method removes from this Vector all of its elements that are contained in the specified Collection.
29	void removeAllElements(): This method removes all components from this vector and sets its size to zero.
30	boolean removeElement(Object obj): This method removes the first occurrence of the argument from this vector.
31	void removeElementAt(int index): This method deletes the component at the specified index.
32	protected void removeRange(int fromIndex, int toIndex): This method removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	boolean retainAll(Collection<?> c): This method retains only the elements in this Vector that are contained in the specified Collection.
34	E set(int index, E element): This method replaces the element at the specified position in this Vector with the specified element.
35	void setElementAt(E obj, int index): This method sets the component at the specified index of this vector to be the specified object.
36	void setSize(int newSize): This method sets the size of this vector.
37	int size(): This method returns the number of components in this vector.
38	List <E> subList(int fromIndex, int toIndex): This method returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39	object[ ] toArray(): This method returns an array containing all of the elements in this Vector in the correct order.
40	<T> T[ ] toArray(T[ ] a): This method returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
41	String toString(): This method returns a string representation of this Vector, containing the String representation of each element.
42	void trimToSize(): This method trims the capacity of this vector to be the vector's current size.

Example: Following program will demonstrate that how we can store different types of data in a Vector.

```
// Java code illustrating add() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector

        Vector v = new Vector();

        v.add(1);
```

```
        v.add(2);

        v.add("Java");

        v.add("Vector");

        v.add(3);

        System.out.println("Vector is " + v);

    }

}
```

```
Output:    [1, 2, Java, Vector, 3]
```

8.11.11 Java PriorityQueue Class

Java PriorityQueue class is a queue data structure application in which objects are inserted based on their **priority**. It is not like standard queues where to insert and delete the element FIFO (First-In-First-Out) algorithm is followed.

In a priority queue, objects are arranged according to their priority. By default, the priority is determined by objects' natural ordering. Default priority can be modified by a value called as Comparator provided during the construction time of queue.

8.11.11.1 PriorityQueue Features: Let's note down few important points on the PriorityQueue.

1. PriorityQueue is an unbounded queue and grows dynamically. The default initial capacity is '11' which can be modified using initialCapacity parameter in appropriate constructor.
2. It does not allow NULL objects.
3. Objects added to PriorityQueue MUST be comparable.
4. The objects of the priority queue are ordered by default in natural order.
5. A Comparator can be used for custom ordering of objects in the queue.
6. The head of the priority queue is the least element based on the natural ordering or comparator based ordering. When we poll the queue, it returns the head object from the queue.
7. If multiple objects are present of same priority the it can poll any one of them randomly.
8. PriorityQueue is not thread safe. Use PriorityBlockingQueue in concurrent environment.
9. It provides O(log(n)) time for add and poll methods.

8.11.11.2 PriorityQueue Class constructors

Table 8.9 : PriorityQueue Class Constructors

Sr. No.	Constructor & Description
1	PriorityQueue(): This creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
2	PriorityQueue(Collection<? extends E> c): This creates a PriorityQueue containing the elements in the specified collection.
3	PriorityQueue(int initialCapacity): This creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
4	PriorityQueue(int initialCapacity, Comparator<? super E> comparator): This creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
5	PriorityQueue(PriorityQueue<? extends E> c): This creates a PriorityQueue containing the elements in the specified priority queue.
6	PriorityQueue(SortedSet<? extends E> c): This creates a PriorityQueue containing the elements in the specified sorted set.

8.11.11.2 PriorityQueue Class Methods

Table 8.10 : PriorityQueue Class Methods

Sr. No.	**Method & Description**
1	boolean add(E e): This method inserts the specified element into this priority queue.
2	void clear(): This method removes all of the elements from this priority queue.
3	Comparator<? super E> comparator(): This method returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
4	boolean contains(Object o): This method returns true if this queue contains the specified element.
5	Iterator<E> iterator(): This method returns an iterator over the elements in this queue.
6	boolean offer(E e): This method inserts the specified element into this priority queue.
7	E peek(): This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
8	E poll(): This method retrieves and removes the head of this queue, or returns null if this queue is empty.
9	boolean remove(Object o): This method removes a single instance of the specified element from this queue, if it is present.

10	int size(): This method returns the number of elements in this collection.
11	Object[] toArray(): This method returns an array containing all of the elements in this queue.
12	<T> T[] toArray(T[] a): This method returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Example: Following program will add the data in PriorityQueue.

```
import java.util.*;
class TestCollection12{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
```

```
}

}

}
```

```
Output:
 head:Amit
  head:Amit
  iterating the queue elements:
  Amit
  Jai
  Karan
  Vijay
  Rahul
  after removing two elements:
  Karan
  Rahul
  Vijay
```

8.11.12 SortedMap Interface in Java

SortedMap is an interface in collection framework. This interface extends Map interface and provides a total ordering of its elements (elements can be traversed in sorted order of keys). Exampled class that implements this interface is TreeMap.

8.11.12.1 Methods of SortedMap Interface

The methods of SortedMap are Listed in the following table –

Table 8.11 : SortedMap InterfaceMethods

Sr. No.	Method & Description
1	Comparator comparator( ): Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.
2	Object firstKey( ): Returns the first key in the invoking map.
3	SortedMap headMap(Object end): Returns a sorted map for those map entries with keys that are less than end.
4	Object lastKey( ): Returns the last key in the invoking map.

5	SortedMap subMap(Object start, Object end): Returns a map containing those entries with keys that are greater than or equal to start and less than end.
6	SortedMap tailMap(Object start): Returns a map containing those entries with keys that are greater than or equal to start.

Example: Following program will add the data in SortedMap in key value pair.

```
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
 public class SortedMapExample
{
   public static void main(String[] args)
   {
      SortedMap<Integer, String> sm =
                  new TreeMap<Integer, String>();

sm.put(new Integer(2), "practice");
      sm.put(new Integer(3), "quiz");
      sm.put(new Integer(5), "code");
      sm.put(new Integer(4), "contribute");
      sm.put(new Integer(1), "Java");
      Set s = sm.entrySet();
        // Using iterator in SortedMap
      Iterator i = s.iterator();
        // Traversing map. Note that the traversal  produced sorted (by keys) output .
      while (i.hasNext())
      {

Map.Entry m = (Map.Entry)i.next();
            int key = (Integer)m.getKey();
          String value = (String)m.getValue();
            System.out.println("Key : " + key + "  value : " + value);
      }
   }
}
```

Output:

```
Key : 1  value : Java
Key : 2  value : practice
Key : 3  value : quiz
Key : 4  value : contribute
Key : 5  value : code
```

Outcomes:

- Students will able to study threading, multithreading, threadcreation in java.
- Students will able to study lifecycle of thread and various states in that cycle of thread in java.
- Students will able to study thread class, constructors of thread class and its methods in java.
- Students will able to study synchronization in java, java genrics, its methods and class, java collections, its methods and classes, etc.

Questions:

1. Write a short note on Concept of threading in java?
2. What is multithreading? How threading can be achieved in java?
3. With neat diagram explain the life cycle of threading.
4. With suitable example demonstrate how to create thread using various methods .
5. List and explain various important methods in thread class.
6. Write a java program to assign and print the thread priority.
7. With a suitable example explain the concept of synchronisation in java.
8. Explain the concepts of Java Generics with its advantages.
9. Explain in short Generic Class, Write a java program to demonstrate the use of it.
10. Explain in short Generic methods, Write a java program to demonstrate the use of it.
11. What is Collection? With neat diagram explain Java Collections Hierarchy.
12. Write a short note on
    - List
    - Map
    - Queue
    - ArrayList
    - Stack
13. What is ArrayList? what are the various features of ArrayList. Demonstrate the use of it with suitable java program.

14. What is LinkedList? what are the various features of LinkedList. Demonstrate the use of it with suitable java program.
15. What is Vector Class? what are the various features of Vector class. Demonstrate the use of it with suitable java program.
16. What is Priority Queue? what are the various features of Priority Queue. Demonstrate the use of it with suitable java program.
17. List and explain some of important methods of SortedMap interface.