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

Course Objectives

- The fundamental point in learning programming is to develop the critical skills of formulating programmatic solutions for real problems.
- To learn the syntax and semantics to write Java programs.
- To understand the fundamentals of object-oriented programming in Java.
- Learn to develop object oriented software using class encapsulation and inheritance, packages and interfaces
- To impart the basic concepts of Java Programming and to develop understanding about Basic Object oriented Design using UML and Applet.
- Design and implement Applet and event handling mechanisms in programs

Learning Outcome:

Upon completion of this course, students will be able to:

- Understand the concept of OOP as well as the purpose and usage principles of inheritance, polymorphism, encapsulation and method overloading.
- Identify classes, objects, members of a class and the relationships among them needed for a specific problem.
- Create Java application programs using sound OOP practices (e.g., interfaces and APIs) and proper program structuring (e.g., by using access control identifies, automatic documentation through comments, error exception handling).
- Use testing and debugging tools to automatically discover errors of Java programs as well as use versioning tools for collaborative programming/editing.
- Develop programs using the Java Collection API as well as the Java standard class library.
- Apply object oriented programming concepts in problem solving through JAVA.

Unit No. and Name	Details	Counseling Sessions	Weightage
Unit 1 Evolution of Java; Variables and Naming Rules	Evolution • History of Java • Features of Java • Difference in the working of C++ and Java • What is JDK, JRE and JVM? • Introduction to Class and objects • Instantiation in java variables and naming rules • Variables in Java • Scopes of the variables • Datatypes • Operators • Primitive Variables • Garbage Collection of the variables. • Source File Declaration Rules • Class and Method Naming Rules • Camel Casing Rule	4	10

Unit 2 Decision Making and looping	• if statement • if-else statement • if – else if – else ladder • nesting of if • ? : operator • switch case • for loop • while loop • Do while loop • Jumps in Loops	4	10
Unit 3 Implementation of Methods	• Methods and Constructor • Method Overloading and Constructor Overloading • Method Overriding • Static members • Final keyword • Inheritance • Super keyword	4	10
Unit 4 Wrapper Classes, Arrays & String	Wrapper Classes • Data Types in Java • Wrapper Classes • Conversion and Utility methods of Wrapper Class • Type Casting • Boxing & autoboxing array & strings • Concept of Arrays • Array Declaration, Construction and Initialization • 1-D Array • Array of Objects • 2-D Arrays	4	10
Unit 5 String Handling and Exception Handling	String Handling • Understanding String class. • Methods of String • String buffer & string builder exception handeling • What is Exception? • Difference in Exception and Error • Using try….catch • Using throws for handling Exception • Making our own Exception • Difference in throw and throws	3	10
Unit 6 Package and Deferred Implementation	Package • How Java Library uses Packages • Import statements in Java • Creating our own package • Making Jar Files Deferred Implementation (Abstract Class and Interfaces) • Abstract Class • Working with abstract class and abstract	4	10

	methods • Interfaces • Abstract Class vs Interfaces • Multiple Interface Implementation • Generalization using Interface		
Unit 7 Java I/O	• Working with File Class • Reading and Writing with Disk Files • BufferedReader and BufferedWriter • Object Serialization • Scanner class	3	10
Unit 8 Thread, Generics and Collection	Thread • Defining Threads • java.lang.Thread and java.lang.Runnable • Thread States • Thread Priorities • Synchronization generics & collection • Defining Generics • Generics Methods • What is Collection API • Difference in Arrays and Collection • List(ArrayList,Vector and LinkedList) • Queue(PriorityQueue) • Map(SortedMap)	4	10
		30	80

Reference:

1. OCA/OCP Java SE 7 Programmer I and II Study Guide: Kathy Sierra and Bert Bates
2. Programming with Java, A Primer: E Balagurusamy
3. Head First Java, Second Edition: Kathy Sierra and Bert Bates

Note: This Study material is still under development and editing process. This draft is being made available for the sole purpose of reference. Final edited copies will be made available once ready.

CHAPTER 1

EVOLUTION OF JAVA, VARIABLES AND NAMING RULES

1.1 BASIC INTRODUCTION

Java is a High level object oriented programming language which is designed at Sun Microsystems (Sun) in 1991 by James Gosling. The intention behind java language was to develop a language which allows 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. **Java** does not have any **full form**, but a programming language originally developed by James Gosling at Sun Microsystems in 1995. It derives much of its syntax from the most popular programming languages of all time: C and C++. Today **Java** of Sun Microsystems is a subsidiary of Oracle Corporation.

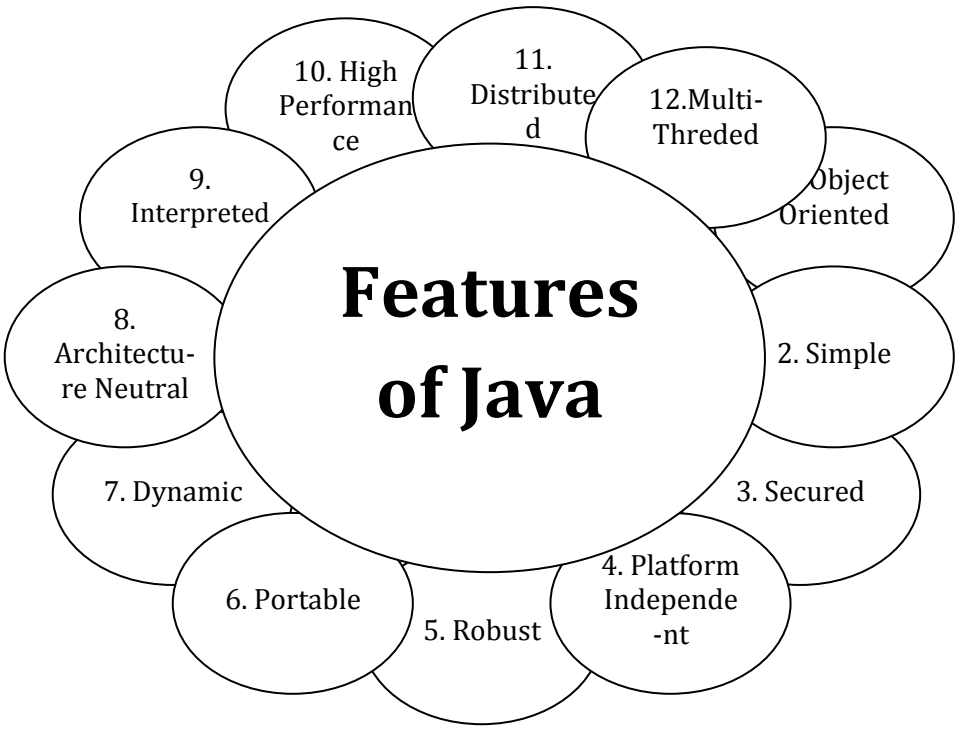
1.3 FEATURES OF JAVA

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

1.3.1. Object oriented: Java is an OOPL that 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.

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 secure in a networked environment. The Java run-time environment uses a byte code verification process 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. It 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: Java applications that are compiled to byte codes can be interpreted by any system that implements the Java Virtual Machine. Since the Java Virtual

Machine is supported across most 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 is compiled to bytecodes, which are interpreted by a Java run-time environment.

1.3.10. High Performance: Although 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 is designed to support various levels of network connectivity. Java applications are network aware: TCP/IP support is built into Java class libraries. They can open and access remote objects on the Internet.

1.4 DIFFERENCE IN THE WORKING OF C++ AND JAVA

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 platform, while java compiler produces a byte code that can be run on any platform. Following two diagrams shows the difference between working of C++ and Java

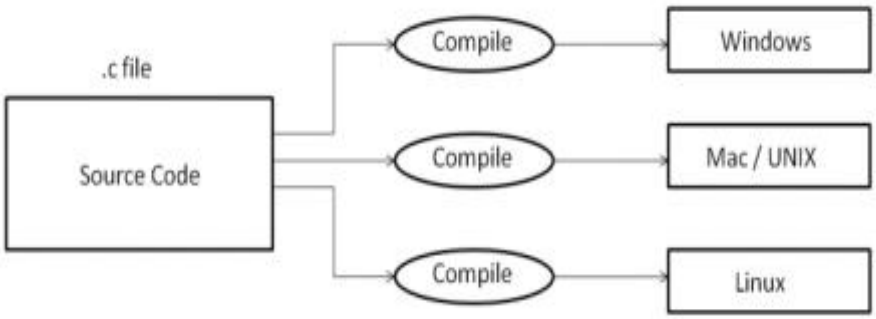


Figure: C Program Execution

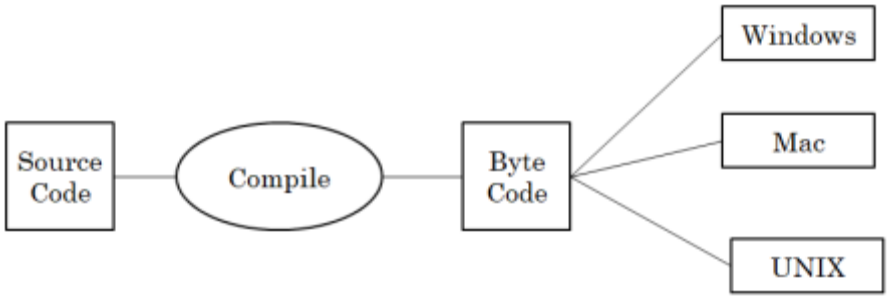
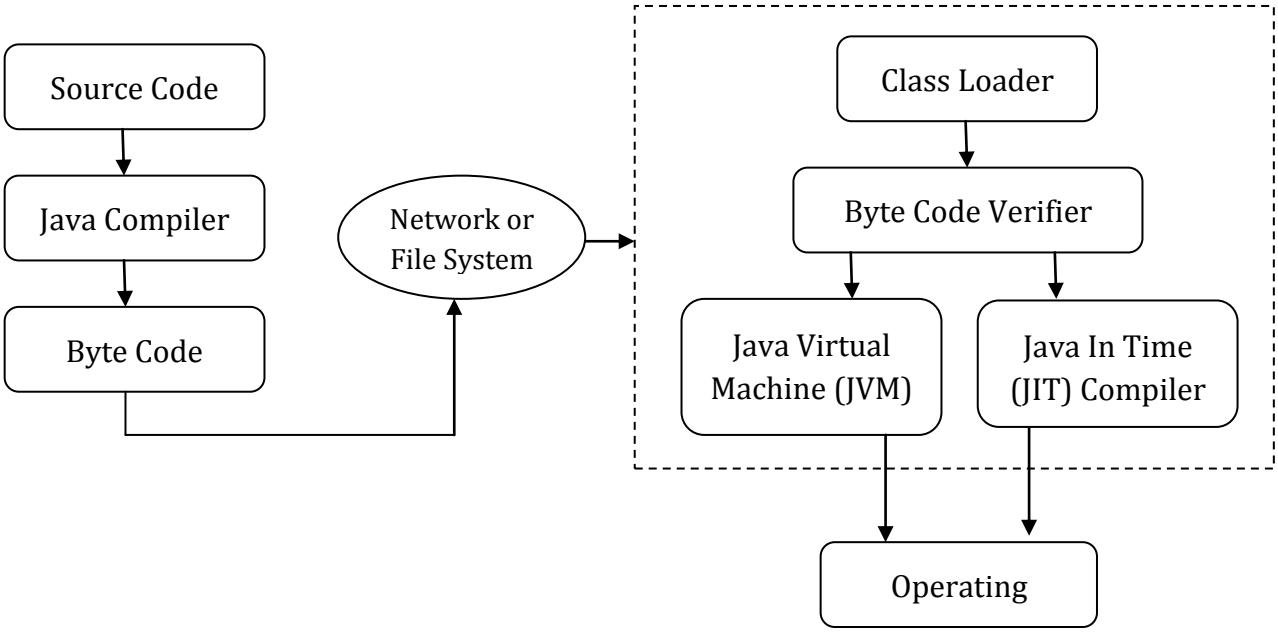


Figure: Java Program Execution

1.4.1Java Environment Architecture

Java combines 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 byte code and generates machine code which will be directly executed by the machine in which **java** program runs. So **java** is both compiled and interpreted language.



Byte code: As discussed above, javac compiler of JDK compiles the java source code into byte code so that it can be executed by JVM. The byte code is saved in a .class file by compiler.
Class Loader: The Java Class loader is a part of the Java Run-time Environment that is responsible for dynamically loading of Java classes into the Java Virtual Machine. The Java run time system does not need to know about files and file systems because of class loaders.
Byte Code Verifier: Is again a part of Java Run-time Environment, after loading the byte code in JVM byte code are first inspected by a verifier. The verifier checks that the instructions cannot perform actions that are obviously damaging.
Java Virtual Machine (JVM): This is generally referred as JVM. Before, we discuss about JVM lets see the phases of program execution. Phases are as follows: we write the program, then we compile the program and at last we run the program.
1) Writing of the program is of course done by java programmer like you and me.

2) Compilation of program is done by javac compiler, javac is the primary java compiler included in java development kit (JDK). It takes java program as input and generates java byte code as output.

3) In third phase, JVM executes the byte code generated by compiler. This is called program run phase.
The Just-In-Time (JIT) compiler: is a component of the java runtime environment that improves the performance of Java applications by compiling byte codes at run time to native machine code. Java programs consists of classes, which contain platform independentbytecodes that can be interpreted by a JVM. At run time, the JVM loads the class files, determines the semantics of each individual byte code, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling byte codes into native machine code at run time.

1.5 What is JDK, JRE and JVM

1.5.1 Java Development Kit(JDK): As the name suggests this is complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc. In order to create, compile and run Java program you would need JDK installed on your computer.The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the 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 libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. In addition, 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. The JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications. JRE includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.

What does JRE consists of?
JRE consists of the following components:

- **Deployment technologies**, including deployment, Java Web Start and Java Plug-in.
- **User interface toolkits**, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD) and input methods.
- **Integration libraries**, including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method

Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.

- **Other base libraries**, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
- **Lang and util base libraries**, including 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 Java Hotspot Client and Server Virtual Machines.

1.5.3Java Virtual Machine (JVM): The **Java Virtual Machine** (JVM) is the virtual machine that runs the Java bytecodes. The JVM doesn't understand Java source code; that's why you need compile your *.java files to obtain *.class files that contain the byte codes understood by the JVM. It's also the entity that allows 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 sufficient to solve the problems of its field but was not able to solve the problems of other fields. For example, *FORTRAN* could have been used to write efficient programs for scientific problems, but it was not good for system code. Similarly, *Basic* 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.

Programming languages such as Cobol, FORTRAN do not have structural principles. They use the Goto statement to control the flow of the program. Therefore, programs using this type of code are made up of many jumps and conditional statements that make it difficult to understand.

Therefore, *"C"* was invented in 1970, to replace the assembly language and to create a structured, effective and high-level language. The development of C was the result of the development process started with *BCPL* by Dennis Ritchie. BCPL is an old language developed

by Martin Richard. Ken Thompson developed a language called *B,* which was influenced by BCPL.

C is a processor-oriented programming language; it is easy to execute and understand. C became quite famous at that time because it was reliable, simple and easy to use.

Though C was a quite efficient and successful programming language, 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 demand at that time. So a new concept came.

C++ came with object-oriented programming features. C++ is the extension of C language which has been used extensively. It is a powerful modern language that includes 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 arose, 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 solution was needed, and this requirement became the reason for the creation of Java, which is a portable and platform-independent language.

History of various 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" was the codename which was given to this form and was released on 8th December 1998. Its real expansion included: strictfp keyword • the Swing 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 for reading and writing images in 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 including 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	Released Date- 20th March 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	Released Date- 25th September, 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	Released Date- 19th March 2019 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 that 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
}
}
```

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

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. If we apply static keyword with any method, it is known as static method. A static method belongs to the class rather than object of a class. A static method invoked without the need for creating an instance of a class. Static method can access static data member and can change the value of it.

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
{
```

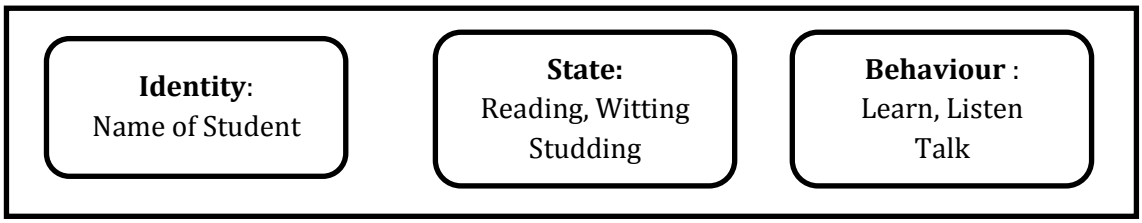
```
public static void main(String [] args)   //Main Function Declaration
{
int j=10;
System.out.println(" J==>"+j);
}
}
Output: J==>10
```

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 dog has states - colour, name, breed as well as behaviours – wagging the tail, barking, eating. An **object** is an instance of a class. Class − A class can be defined as a template/blueprint that describes the behaviour/state that the **object** of its type support.

An object consists of:

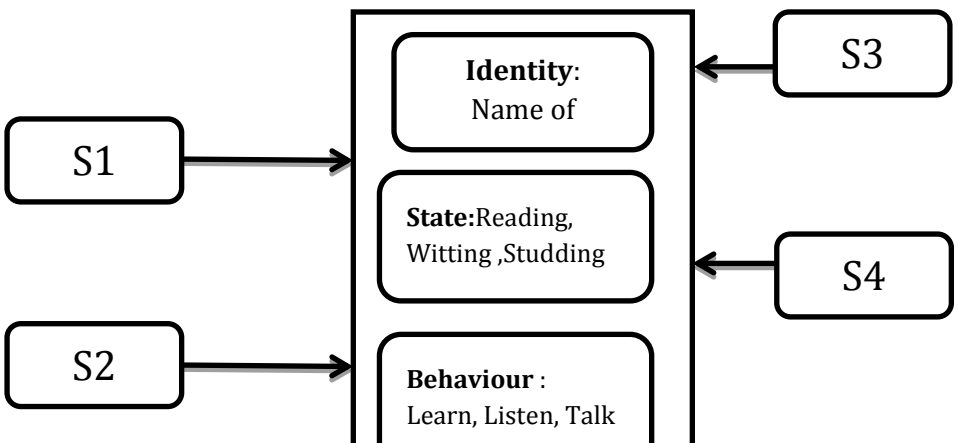
1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behaviour:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: Student



When an object of a class is created, the class is said to be **instantiated**. 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 any number of instances.

```
Student S1 =new Student();

Student S2 =new Student();

Student S3 =new Student();

Student S3 =new Student();
```



1.8 Instantiation in Java

1.9 Variables in Java

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

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

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable 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

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.

It is called instance variable because its value is instance specific and is not shared among instances.

3) Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory. 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 sign** is used to assign values to the variable.

To create a variable that should store text, look at 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 short names (like x and y) or more descriptive names (age, sum, totalVolume).

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

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

1.10 Scope of Variables in Java

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e. scope of a variable can determined at compile time and independent of function call stack.
Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

1.10.1 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**:

```
publicclassClassScopeExample {
   privateInteger amount = 0;
   publicvoidexampleMethod() {
      amount++;
   }
   publicvoidanotherExampleMethod() {
      Integer anotherAmount = amount + 4;
   }
}
```

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

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

```
publicclassMethodScopeExample {
   publicvoidmethodA() {
      Integer area = 2;
   }
   publicvoidmethodB() {
      // compiler error, area cannot be resolved to a variable
      area = area + 2;
   }
}
```

In *methodA*, we created a method variable called *area*. For that reason, we can use *area* inside *methodA*, but we can't use it anywhere else.

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

```
publicclassLoopScopeExample {
   List<String> listOfNames = Arrays.asList("Joe", "Susan", "Pattrick");
   publicvoiditerationOfNames() {
      String allNames = "";
      for(String name : listOfNames) {
         allNames = allNames + " "+ name;
      }
      // compiler error, name cannot be resolved to a variable
      String lastNameUsed = name;
   }
}
```

We can see that method *iterationOfNames* has a method variable called *name*. This variable can be used only inside the loop and is not valid outside of it.

1.10.4 Bracket Scope

We can define additional scopes anywhere using brackets (*{}*):

```
publicclassBracketScopeExample {
   publicvoidmathOperationExample() {
      Integer sum = 0;
      {
         Integer number = 2;
         sum = sum + number;
      }
      // compiler error, number cannot be solved as a variable
      number++;
   }
}
```

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

1.11 Java Data Types

1.11.1 Primitive Data Types

Primitive data types are those data types which are provided by the language in build such as int, char, float etc.A primitive data type specifies the size and type of variable values, and it has no additional methods.There are eight primitive data types in Java:

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

1.11.2 Non-Primitive Data Types

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

The main difference between **primitive** and **non-primitive** data types is Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).

1.13 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

1.13.1 Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

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

1.13.2 Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x =10;
```

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

A list of all 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

1.13.4 Java Comparison Operators

Comparison operators are used to compare two values:

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

1.13.5 Java Logical Operators

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

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 statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

1.14 Java Garbage Collection

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine or JVM for short. When Java programs run 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 finds these unused objects and deletes them to free up memory.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

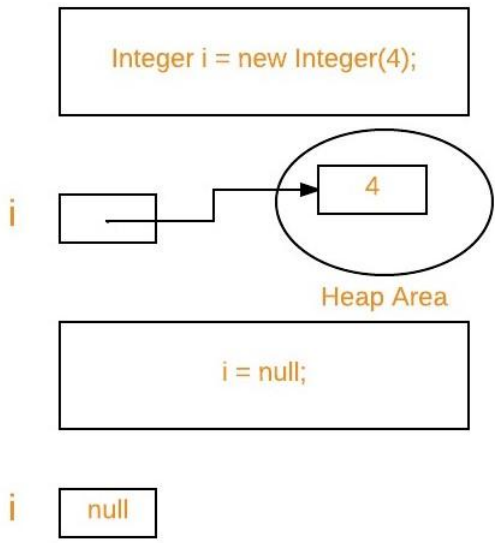
1.14. qAdvantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

1.14.2 How can an object be unreferenced?

There are many ways:

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



1.14.3 finalize() method

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

1.14.4 gc() method

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

1.15 What are Java source file declaration rules?

A Java source file is a plain text file containing Java source code and having .java extension. The .java extension means that the file is the Java source file. Java source code file contains source code for a class, interface, enumeration, or annotation type. There are some rules associated to Java source file. We should adhere to following rules while writing Java source code.

- There can be only one public class per source code file.
- Comments 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 inserted anywhere in a program code where a white space can be
- If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Dog { } must be in a source code file named Dog.java.
- If the class is part of a package, 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. If there are no package or import statements , the class declaration must be the first line in the source code file.
- import and package statements apply to all classes within 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 class.
- Files with non~public classes can have a name that does not match any of the classes in the file

1.16 Java Naming Conventions

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

Java heavily uses **Camel Case** notations for naming the methods, variables etc. and **TitleCase** notations for classes and interfaces.

Let’s understand these naming conventions in detail with examples.

1.16.1 Packages naming conventions

Package names must be a group of words starting with all lowercase domain name (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;
```

1.16.2. Classes naming conventions

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

```
publicclassArrayList {}

publicclassEmployee {}

publicclassRecord {}

publicclassIdentity {}
```

1.16.3. Interfaces naming conventions

In Java, interfaces names, generally, should be **adjectives**. Interfaces should be in titlecase with the first letter of each separate word capitalized. In same cases, interfaces can be **nouns** as well when they present a family of classes e.g. List and Map.

```
publicinterfaceSerializable {}

publicinterfaceClonable {}

publicinterfaceIterable {}

publicinterfaceList {}
```

1.16.4. 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 be a single or 2-3 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) {}
```

1.16.5. Variables naming conventions

All instance, 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;

publicEmployeeDao employeeDao;

privateProperties properties;

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

}
```

1.16.6. 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;
```

1.16.7. Enumeration naming conventions

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

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

CHAPTER 2

DICISION MAKING AND LOOPS

2.1 Java If-else Statement

The Java*if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

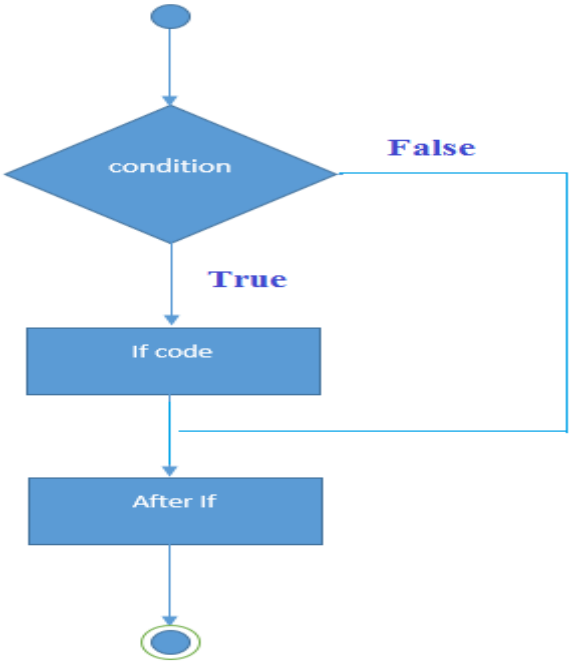
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

2.1.1 Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

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



Example:

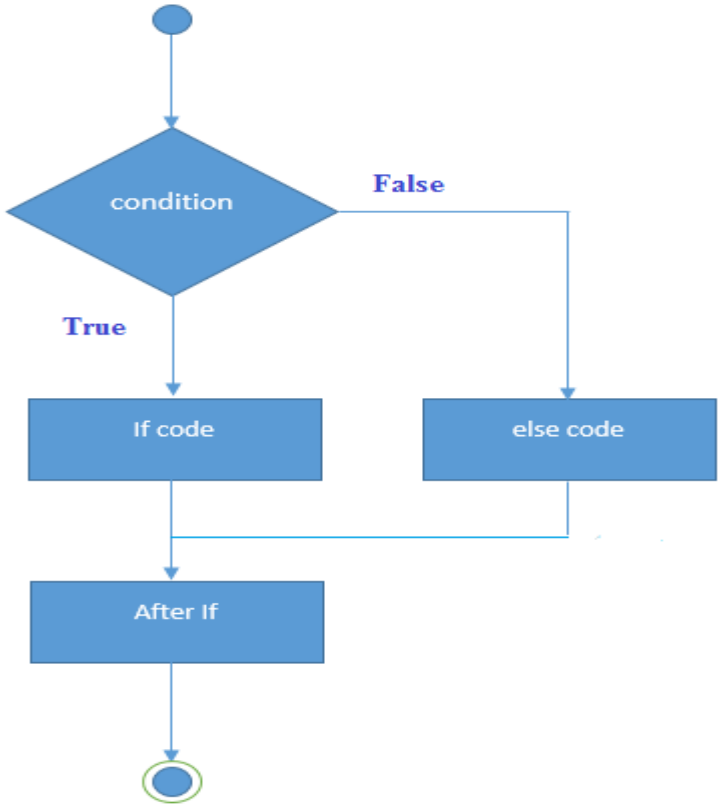
```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
   }
}
}
```

2.2 Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

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



Example:

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
```

```
}

}
```

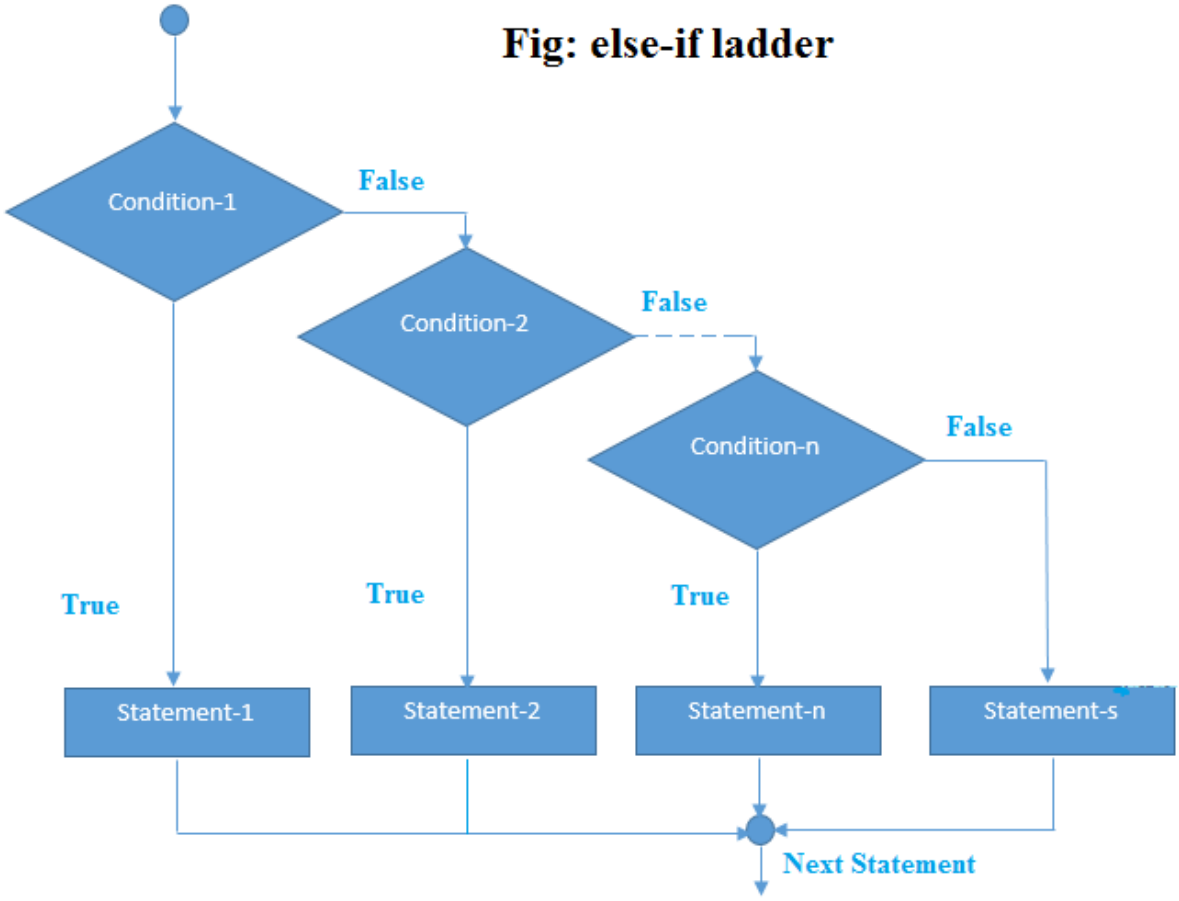
2.3 Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Fig: else-if ladder



Example:

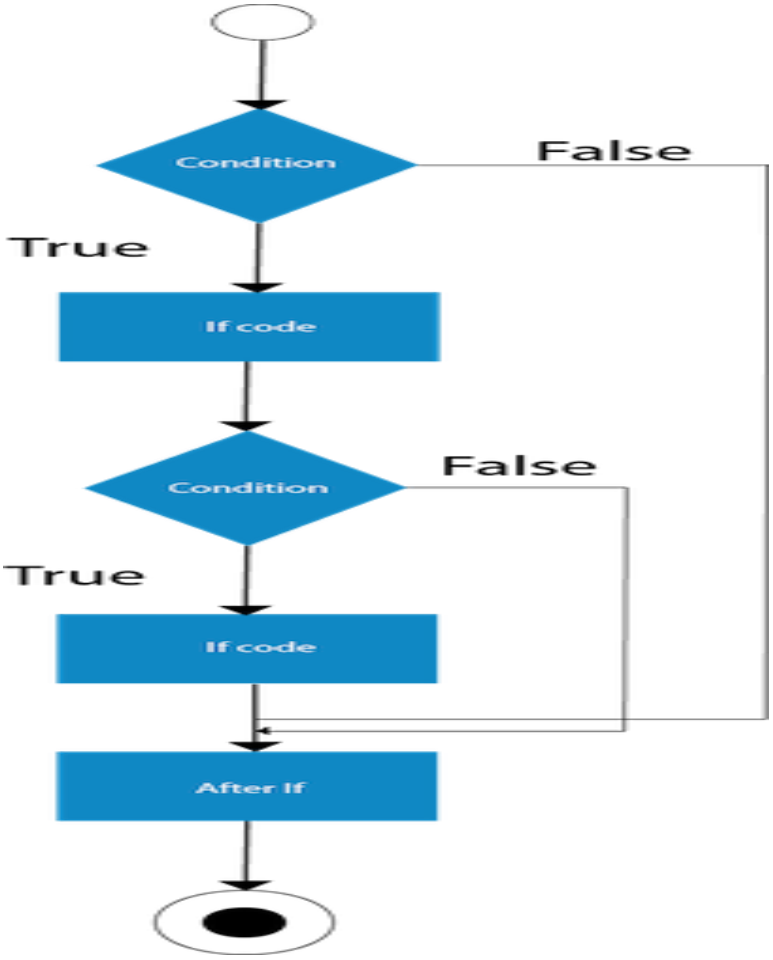
```
//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;

    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!");
    }
}
}
```

2.4 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
    }
}
```



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");
        }
    }
}}
```

2.5 Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, 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);
}
}
```

2.6 Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

- for loop
- while loop
- do-while loop

2.7 Java for Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

2.7.1 Java Simple for Loop

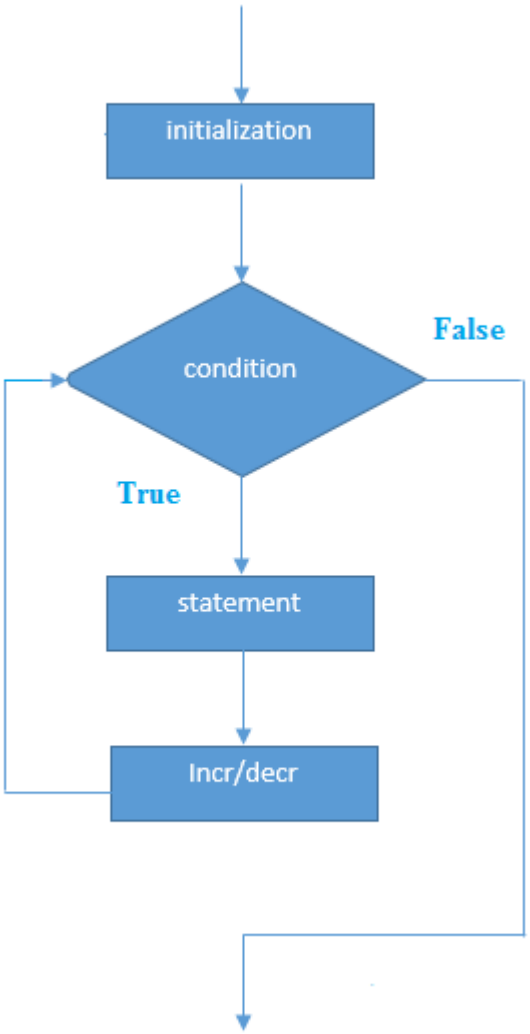
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must 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 increments or decrements the variable value. It is an optional condition.

Syntax:

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

Flowchart:



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);
    }
}
}
```

2.7.2 Java Nested For Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

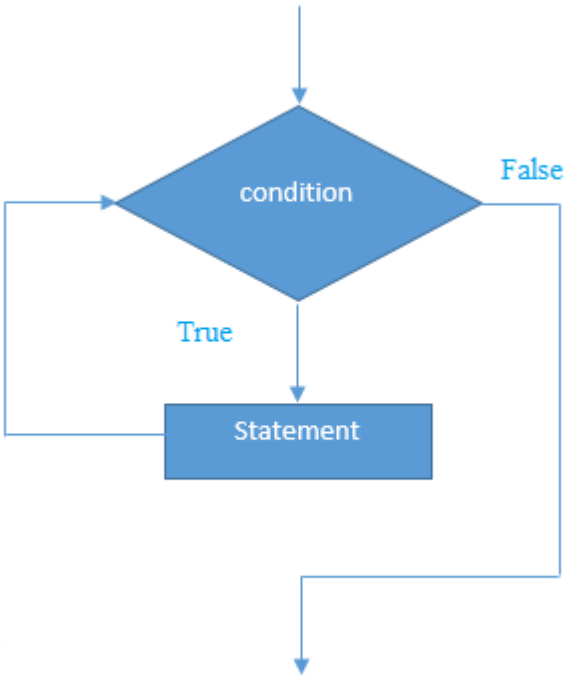
```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
	System.out.println(i+" "+j);
}//end of i
}//end of j
}
}
```

2.8 Java While Loop

The Java*while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

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




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

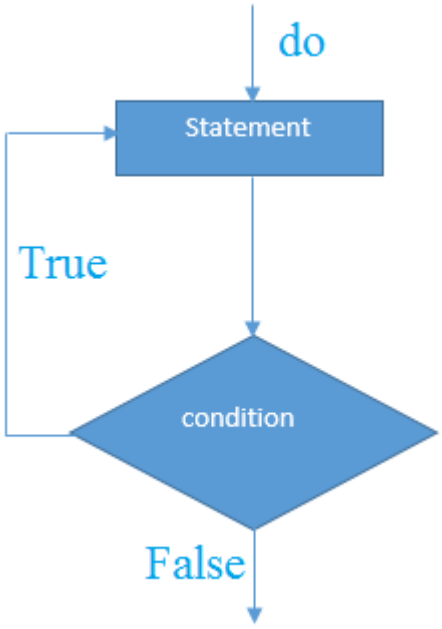
2.9 Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

- do{
- //code to be executed
- }while(condition);



Example:

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

2.10 Java for Loop vs. While Loop vs. Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given Boolean condition.	The Java do while loop 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.
When to	If the number of iteration is fixed, it	If the number of iteration	If the number of iteration

use	is recommended to use for loop.	is not fixed, it is recommended to use while loop.	is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

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);`

2.11 Java Switch Statement

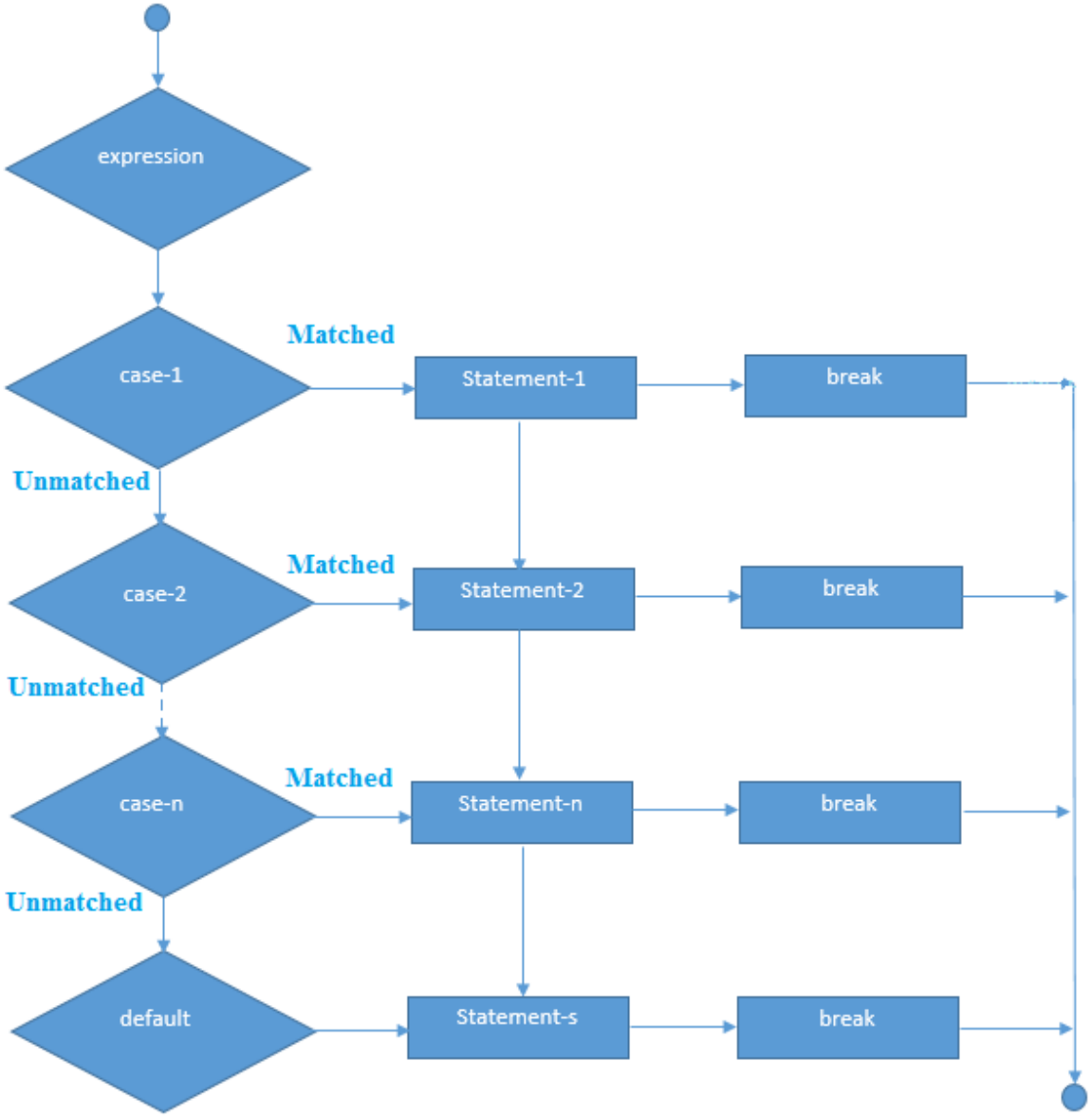
The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. 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 must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which 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;
}
```



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");

   }

}

}
```

2.12 Java Break Statement

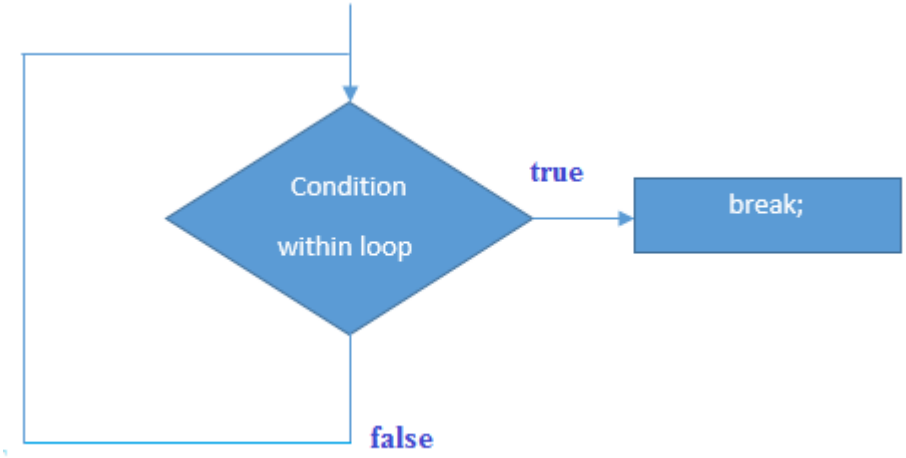
When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of 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;
```



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);
   }
}
}
```

2.13 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 *continue 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:

1. jump-statement;
2. 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);

   }

}

}
```

2.14 Java Comments

The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code.

2.14.1 Types of Java Comments

There are three types of comments in Kava.

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

2.14.2 Java Single Line Comment

4. The single line comment is used to comment only one line.

Syntax:

```
1. //This is single line comment
```

Example:

```
1. public class CommentExample1 {
2. public static void main(String[] args) {
3.     int i=10;//Here, i is a variable
4.     System.out.println(i);
5. }
6. }
```

2.14.3 Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
1. /*
2. This
3. is
4. multi line
5. comment
6. */
```

Example:

```
1. public class CommentExample2 {
2. public static void main(String[] args) {
3. /* Let's declare and
4.  print variable in java. */
5.     int i=10;
6.     System.out.println(i);
7. }
8. }
```

2.14.3 Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
1. /**
2. This
3. is
4. documentation
5. comment
6. */
```

Example:

```
1. /** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*
   /
2. public class Calculator {
3. /** The add() method returns addition of given numbers.*/
4. public static int add(int a, int b){return a+b;}
5. /** The sub() method returns subtraction of given numbers.*/
6. public static int sub(int a, int b){return a-b;}
7. }
```

CHAPTER 3

IMPLEMENTATION OF METHODS

3.1 Introduction:

3.1.1 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 that looks and feels like other instances of the same class.

A class is a user defined blueprint or prototype from which objects are created. It represents 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 name should begin 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, preceded 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

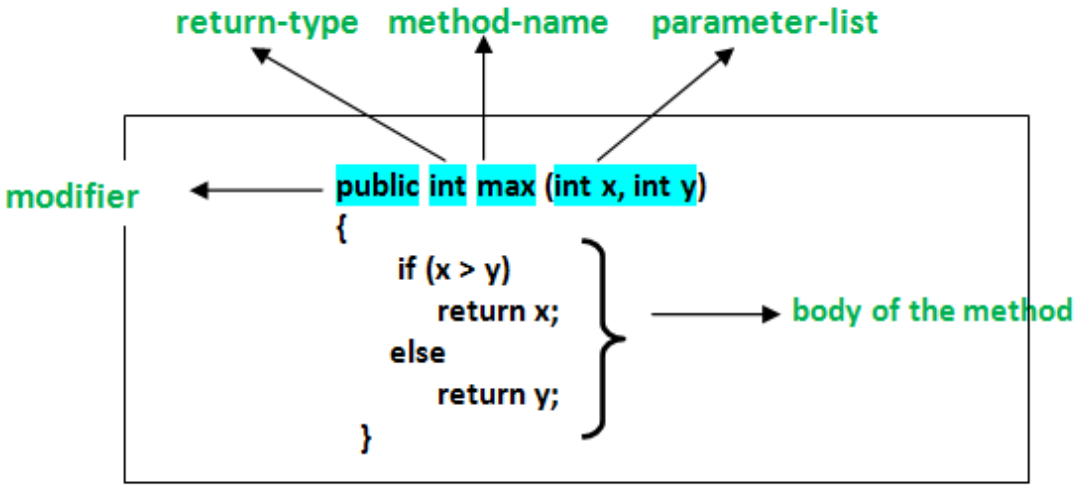
3.2 Methods and Constructors in Java

3.2.1 Java Methods:

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python. Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration

- **Modifier**-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
    - public: accessible in all class in your application.
    - protected: accessible within the class in which it is defined and in its **subclass(es)**
    - private: accessible only within the class in which it is defined.
    - default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- **The return type**: The data type of the value returned by the method or void if does not return a value.
- **Method Name**: the rules for field names apply to method names as well, but the convention is a little different.
- **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 ().
- **Exception list:** The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body:** it is enclosed between braces. The code you need to be executed to perform your intended operations.



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 and exceptions are not considered as part of it.
Method Signature of above function:

```
max(int x, int y)
```

Calling a method

The method needs to be called for using its functionality. There can be three situations when a method is called:

A method returns to the code that invoked it when:

- It completes all the 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
```

3.2.2 What is Constructor

In Java, a *constructor* is a block of codes similar to the method. It is called when an instance of the 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. It is not necessary to write a constructor for a class. 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.

When Constructor is called

Every time an object is created using new() keyword, at least one constructor is called. It calls a default constructor.

3.2.2.1 Rules for Writing Java constructor

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized
4. Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

3.2.2.2 Types of constructor

There are two type of constructor in Java:

a. 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. Default constructor provides the default values to the object like 0, null, etc. depending on the type.

A constructor is called "Default Constructor" when it doesn't have any parameter.

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 constructor

```
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
}

}
```

purpose of a default constructor

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

b. Java Parameterized Constructor

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

Purpose of parameterized constructor?

The parameterized constructor is used to provide 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 have two parameters. We can have any number of parameters in the constructor.

```
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

c.Ccopy constructor:

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

```
classComplex {
    privatedoublere, im;
        // copy constructor
  Complex(Complex c) {
    System.out.println("Copy constructor called");
    re = c.re;
    im = c.im;
  }
```

```
   // Overriding the toString of Object class
   @Override
   publicString toString() {
      return"("+ re + " + "+ im + "i)";
   }
     publicstaticvoidmain(String[] args) {
       Complex c1 = newComplex(10, 15);

     // Following involves a copy constructor call
       Complex c2 = newComplex(c1);
         System.out.println(c2); // toString() of c2 is called here
   }
}
```

3.3 Method Overloading :

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

3.3.1 Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class AddOperation{
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(AddOperation.add(14,10));
System.out.println(AddOperation.add(21,11,31));
}
```

2) Method Overloading: changing data type of arguments

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

```
class AddOperation {
int add(int a, int b)
{return a+b;}
double add(float a, float b)
{return a+b;}
}
public static void main(String[] args){
System.out.println(AddOperation.add(21,11));
System.out.println(AddOperation.add(12.3,12.1));
}
```

Like methods, constructors can also be overloaded. It allows having *more than one constructor inside one Class but with different signature*.Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task. The difference in constructors parameters can be number of parameters, data types of parameters, order of parameters

Important points related to Constructor overloading:

1. Constructor overloading is similar to method overloading in Java.

2. You can call overloaded constructor by using this() keyword in Java.

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

4. make sure you add no argument default constructor because once compiler will not add if you have added any constructor in Java.

5. if an overloaded constructor called, it must be the first statement of constructor in java.

6. Its best practice to have one primary constructor and let overloaded constructor calls that. this way your initialization code will be centralized and easier to test and maintain.

3.4 Method Overriding

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide 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 same name, same parameters or signature and same 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 way by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object 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* (not the type of the reference variable) 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 amethod

3.4.1 Rules for method overriding:

1. **Overriding and Access-Modifiers:** The access modifier for an overriding method can allow more, but not less, access than the overridden method. For 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:** If we don't want a method to be overridden, we declare it as final.
3. **Static methods cannot be overridden (Method Overriding vs Method Hiding) :** When you defines a static method with same signature as a static method in base class, it is known as method hiding.
4. Private methods cannot be overridden: Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.(See this for details).

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 does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

9 **.Overriding and abstract method:** Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

10. **Overriding and synchronized/strictfp method:** The presence of synchronized/strictfp modifier with method have 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.

3.5 Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

- variable
- method
- class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

3.5.1 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

3.5.2 Java final method

If you make any method as final, you cannot override it.

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

3.5.3 Java final class

If you make any class as final, you cannot extend it.

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

3.6 Static keyword in java

The **static keyword in Java** is used mainly for memory management. It is used with variables, methods, blocks and nested classes. It is a keyword that is used to share 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. The main method of a class is generally labeled static.

Static variable

If any variable we declared as static is known as static variable.

- A Static variable is used to fulfill the common requirement. For 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 only once in the class area at the time of class loading.

Advantage of static variable

Using a static variable we make our program memory efficient (i.e it saves memory).

When and why we use static variable

Suppose we want to a store record of all employees of any company, in this case, employee id is unique for every employee but company name is common for all. When we create a static variable as a company name then only once memory is allocated otherwise it allocates a memory space each time for every employee.

Syntax: public static datatype VariableName;
Example: public static int a;

Difference between static and final keyword

static keyword always fixed the memory that means that it will be located only once in the program whereas **final** keyword always fixed the value that means it makes variable values constant.

Static Method in Java

If you apply static keyword with any method, it is known as static method. A static method belongs to the class rather than the object of a class. It can be invoked without the need for creating an instance of a class. It can access static data member and can change the value of it.

Example: public static void mymethod() {}

```
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

3.7 This keyword in java

this is a reference variable that refers to the current object. It is a keyword in java language represents current class object

Usage of this keyword

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

Why use this keyword in java?

The main purpose of **using this keyword** is to differentiate the formal parameter and data members of class, whenever the formal parameter and data members of the class are similar then jvm get ambiguity (no clarity between formal parameter and member of the class)

To differentiate between formal parameter and data member of the class, the data member of the class must be preceded by "this".

Example:

3.8 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.

3.8.1 Important points

- In the inheritance the class which is give data members and methods is known as base or super or parent class.

- The class which is taking the data members and methods is known as sub or derived or child class.
- The data members and methods of a class are known as features.
- The concept of inheritance is also known as re-usability or extendable classes or sub classing or derivation.

3.8.2 Why use Inheritance?

- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

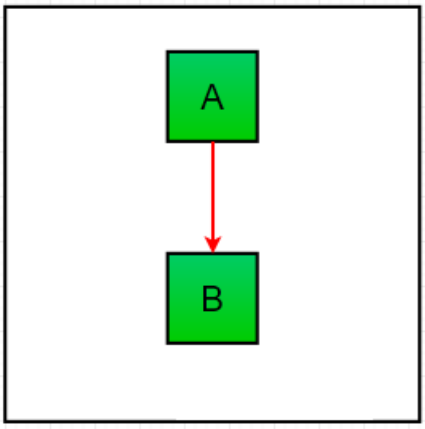
3.8.3 Types of Inheritance

Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance; they are:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

3.8.3.1 Single inheritance

Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



Example:
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);
	}
	}
```

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);
        }
}
```

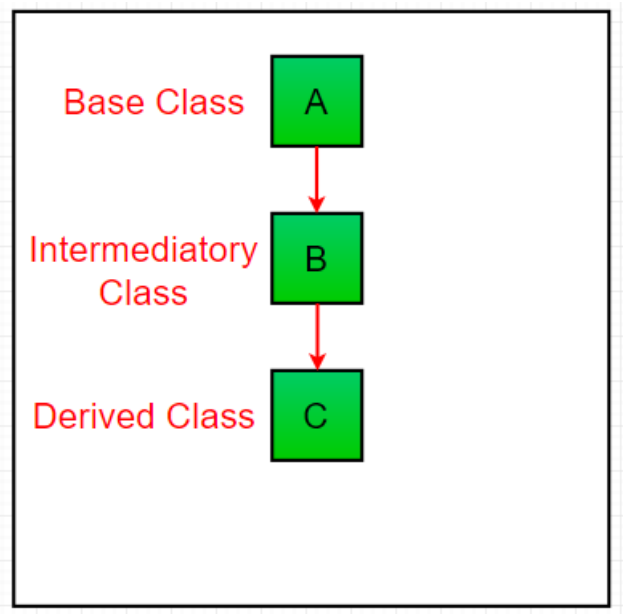
Class Containing Main function which will Create object of Sub class access the Super class members.

```
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
```

3.8.3.2 **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. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.




```
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;
	}

}
```

```
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;
        }


}
```

```
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);
        }
}
```

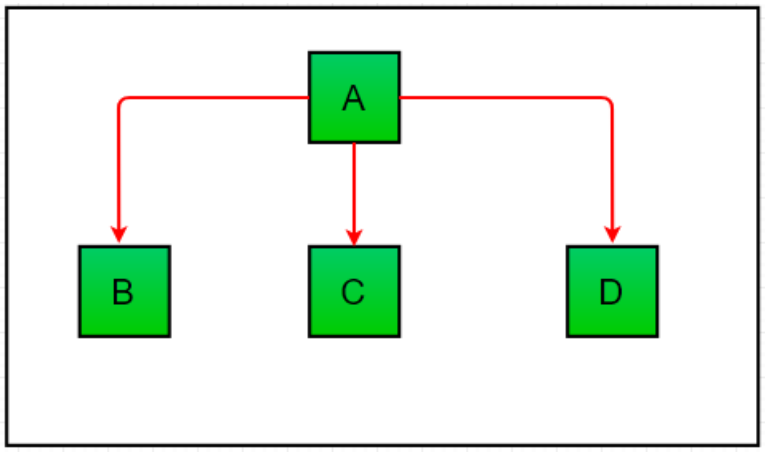
```
class StudentDemo
{
```

```
        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
```

3.8.3.3. **Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.




```
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;
                        }

}
```

```
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);

        }
}
```

```
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);
```

```
        }
}
```

```
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

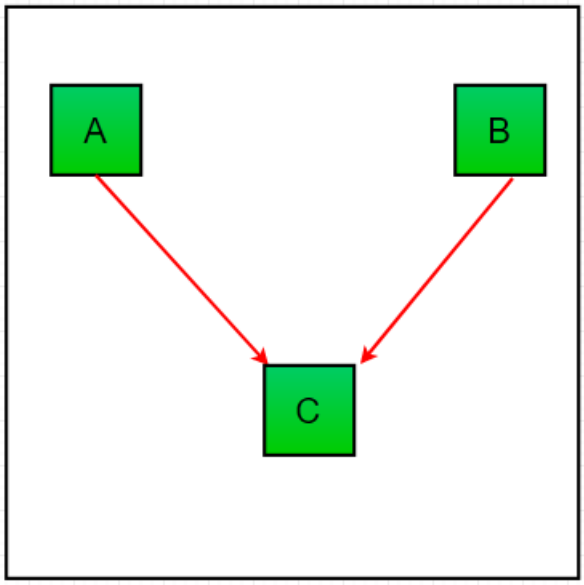
Student Address==>Jalgaon

Student Specialization==>Brain

Student College==>IIM

3.8.3.4 **Multiple Inheritance (Through Interfaces):** In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.

Multiple inheritance in jave is not directly supported that is one class cannot extend more than one class at a time, so to achieve multiple inheritance we need to use interface. Using which we can obtain multiple inheritance for which class have to extend one class and can implement one or more interfaces.




```
class Employee
{
	int emp_no;
	String e_name;
	Employee()
	{
		System.out.println("U r in Default Constructor of Employee Class");
		emp_no=10;
		e_name="Abcd";
	}
	Employee(int i,String s)
	{
		System.out.println("U r in Parametric Constructor of Employee Class");
		emp_no=i;
		e_name=s;
	}

}
```

```
interface CalSalary
{
	void calSalary();

}
```

```
class WegEmployee extends Employee implements CalSalary
{
	int hrs,hrs_rt;
	String dept;
	int sal;
	WegEmployee()
	{
		System.out.println("U r in Default Constructor of WegEmployee Class");
		hrs=8;
		hrs_rt=200;
		dept="Testing";
	}
	WegEmployee(int i,String s,int j,int r,String s1)
	{
		super(i,s);
		System.out.println("U r in Parametric Constructor of WegEmployee Class");
		hrs=j;
		hrs_rt=r;
		dept=s1;
	}
	public void display()
	{
		 System.out.println("\n\t The Emp_Id==>"+emp_no);
		 System.out.println("\n\t The Emp_Name is==>"+e_name);
		 System.out.println("\n\t The Dept. of Emp is==>"+dept);
		 System.out.println("\n\t The Working Hours Are==>"+hrs);
		 System.out.println("\n\t The Hrs per Rate is==>"+hrs_rt);

		  calSalary();
	}
   public void calSalary()
	{
		sal=hrs*hrs_rt*25;
		System.out.println("\n\t The Salary Of Employee is==>"+sal);
	}
}
```

```
class EmployeeDemo
{
	public static void main(String[] args)
	{
		System.out.println("Hello World!");
		WegEmployee s= new WegEmployee();
		s.display();
		WegEmployee s1= new WegEmployee(35,"Vijay",8,100,"Account");
		s1.display();
```

```
        }
}
```

Output:

Hello World
U r in Default Constructor of Employee Class
U r in Default Constructor of WegEmployee Class

The Emp_Id==>10

The Emp_Name is==>Abcd

The Dept. of Emp is==>Testing

The Working Hours Are==>8

The Hrs per Rate is==>200

The Salary Of Employee is==>40000
U r in Parametric Constructor of Employee Class
U r in Parametric Constructor of WegEmployee Class

The Emp_Id==>35

The Emp_Name is==>Vijay

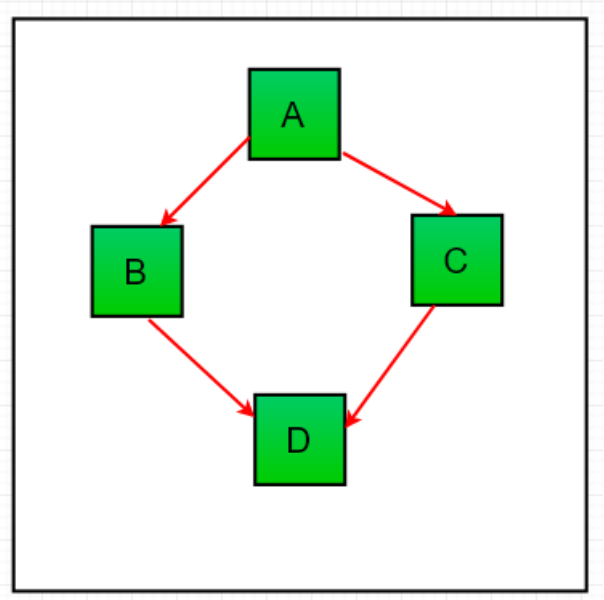
The Dept. of Emp is==>Account

The Working Hours Are==>8

The Hrs per Rate is==>200

The Salary Of Employee is==>40000

3.8.3.5 **Hybrid Inheritance(Through Interfaces):** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.
Same as Multiple inheritance hybrid inheritance in jave is not directly supported that is one class cannot extend more than one class at a time, so to achieve hybrid inheritance we need to use interface. Using which we can obtain multiple inheritance for which class have to extend one class and can implement one or more interfaces.



3.8.4 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 create 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.
- Bottom most derived class contains logical appearance for the data members of all 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 final base classes are not reusable or not inheritable.
- If we are do not want to give some of the features of base class to derived class than such features of base class must be as private hence private features of base class are not inheritable or accessible in derived class.
- Data members and methods of a base class can be inherited into the derived class but constructors of base class can not be inherited because every constructor of a class is made for

initializing its own data members but not made 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 concept is known as scope of base class object).
- For each and every class in java there exists an implicit predefined super class called java.lang.Object. because it providers garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.

3.9 Interfaces in Java

Like a class, an interface 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:

```
interfacePlayer
{
```

```
    finalintid = 10;
    intmove();
}
```

A real-world example:

Let's consider the example of vehicles like bicycle, car, bike, etc. they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc. implement all these functionalities in their own class in their own way.

```
importjava.io.*;

interfaceVehicle {

    // all are the abstract methods.
    voidchangeGear(inta);
    voidspeedUp(inta);
    voidapplyBrakes(inta);
}

classBicycle implementsVehicle{

    intspeed;
    intgear;

     // to change gear
    @Override
    publicvoidchangeGear(intnewGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    publicvoidspeedUp(intincrement){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    publicvoidapplyBrakes(intdecrement){

        speed = speed - decrement;
    }

    publicvoidprintStates() {
         System.out.println("speed: "+ speed
             + " gear: "+ gear);
    }
}

classBike implementsVehicle {

    intspeed;
    intgear;
```

```
    // to change gear
    @Override
    publicvoidchangeGear(intnewGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    publicvoidspeedUp(intincrement){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    publicvoidapplyBrakes(intdecrement){

        speed = speed - decrement;
    }

    publicvoidprintStates() {
         System.out.println("speed: "+ speed  + " gear: "+ gear);
    }

}
classGFG {

    publicstaticvoidmain (String[] args) {

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

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

        // creating instance of the bike.
        Bike bike = newBike();
        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
```

3.10 Super Keyword in Java

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

3.10.1. Use of super with variables: This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* 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
```

3.10.2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
classPerson
```

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

/* Subclass Student */
classStudent extendsPerson
{
   voidmessage()
   {
      System.out.println("This is student class");
   }

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

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

/* Driver program to test */
classTest
{
   publicstaticvoidmain(String args[])
   {
      Student s = newStudent();

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

Output:

This is Student Class

This is Person Class

3.10.3. Use of super with constructors: super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non-parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

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

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

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

/* Driver program to test*/
classTest
{
   publicstaticvoidmain(String[] args)
   {
      Student s = newStudent();
   }
}
```

Output:

```
Person class Constructor

Student class Constructor
```

3.10.4 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 superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a 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*.

CHAPTER 4

WRAPPER CLASSES, ARRAY AND STRING

4.1 Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value 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 handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is 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.

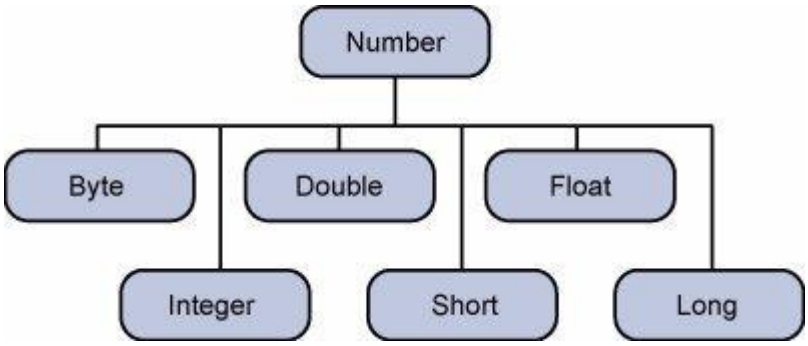
Sr. No.	Wrapper Class	Primitive Data Types
1	Character	Char
2	Short	short
3	Byte	byte
4	Integer	int
5	Float	float
6	Double	double
7	Long	long
8	Boolean	Boolean

4.2 Java.lang.Number Class in Java

Most of the time, while working with numbers in java, we use primitive data types. But, Java also provides various numeric wrapper sub classes under the abstract class Number present in *java.lang* package. There are mainly **six** sub-classes under Number class.These sub-classes define some useful methods which are used frequently while dealing with numbers.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

These classes “wrap” the primitive data type in a corresponding object. Often, the wrapping is done by the compiler. If you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class for you. Similarly, if you use a Number object when a primitive is expected, the compiler unboxes the object for you. This is also called Autoboxing and Unboxing.



Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements −

Sr. No.	Method & Description
1	xxxValue(): Converts the value of *this* Number object to the xxx data type and returns it.
2	compareTo(): Compares *this* Number object to the argument.
3	equals(): Determines whether *this* number object is equal to the argument.
4	valueOf(): 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(): Returns the absolute value of the argument.
8	ceil(): Returns 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(): Returns the smaller of the two arguments.
13	max(): Returns the larger of the two arguments.
14	exp(): Returns the base of the natural logarithms, e, to the power of the argument.
15	log(): Returns the natural logarithm of the argument.
16	pow(): Returns the value of the first argument raised to the power of the second argument.
17	sqrt(): Returns the square root of the argument.
18	sin(): Returns the sine of the specified double value.
19	cos(): Returns the cosine of the specified double value.
20	tan(): Returns the tangent of the specified double value.
21	asin(): Returns the arcsine of the specified double value.
22	acos(): Returns 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.

4.3 Java Integer Class

The Java Integer class comes under the **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.

Java Integer 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.

4.4 Java.Lang.Float class in Java

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.

Some of the Methods of Float Class

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.

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.

Some of the Methods of Byte Class

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.

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.

Some of the Methods of Short Class

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.

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.

Some of the Methods of Long Class

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.

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.

Some of the Methods of Double Class

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.

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.

Some of the Methods of Charecter Class

Sr.No.	Method & Description
1	**toString(char ch)** : It 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)** : It 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)** : It 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.

4.10 Type conversion / Type Casting

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

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. 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.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or Boolean. Also, char and Boolean are not compatible with each other.

Byte → Short → Int → Long → Flaot → Double

Example:

```
classTest
{
   publicstaticvoidmain(String[] args)
   {
      inti = 100;

      // automatic type conversion
      longl = i;

      // automatic type conversion
      floatf = l;
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
      System.out.println("Float value "+f);
   }
}
```

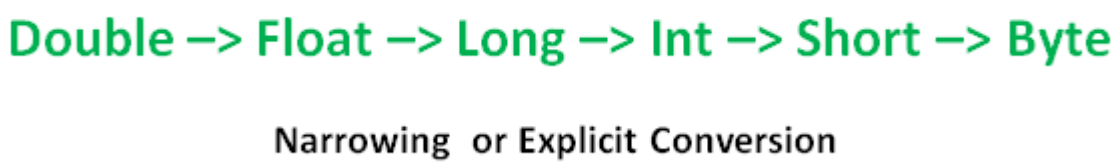
Output:

Int value 100
Long value 100
Float value 100.0

Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we 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.



Narrowing or Explicit Conversion

Example:

```
/Java program to illustrate explicit type conversion
classTest
{
   publicstaticvoidmain(String[] args)
   {
      doubled = 100.04;

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

      //explicit type casting
      inti = (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

4.11 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
//Autoboxing 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: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For 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

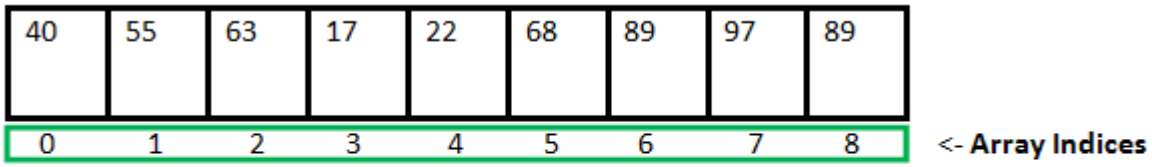
4.12 Arrays in Java

an array is a collection of similar type of elements which have a contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

- 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 direct superclass of an array type is Object.

Array can contains primitives data types as well as objects of a class depending on the definition of array. In case of primitives 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

4.12.1 Advantages

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

4.12.2 Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

4.12.3 Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

4.12.3.1 Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
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
```

4.12.3.2 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

1. **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
```

Arrays of Objects

4.13 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.

4.13.1 Declaring An Array Of Objects In Java

We use the class name Object, followed by square brackets to declare an Array of Objects.

Object[] JavaObjectArray;
Another declaration can be as follows:

4.13.2 Declaring An Array Objects With Initial Values

Declaration of an array of object can be done by adding initial values. Here, we create an array consisting of a string with the value “Women Empowerment”, 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
```

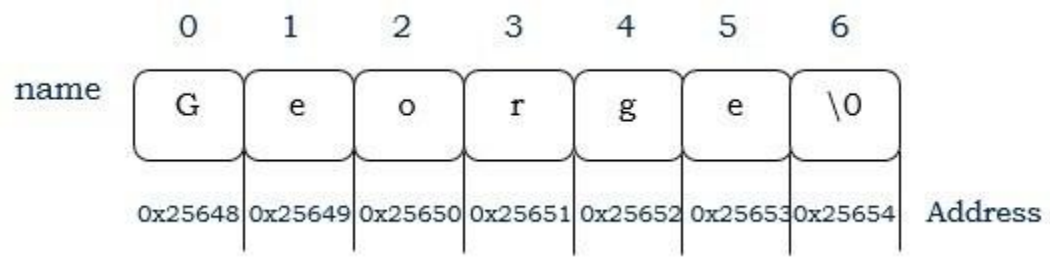
CHAPTER 5

STRING AND EXCEPTIONS HANDLING

5.1 Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are imm In **Java** programming language, **strings** are treated as objects. The **Java** platform provides the **String** class to create and manipulate **strings**. Whenever a change to a String is made, an entirely new String is created. 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.



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

For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);

is same as:

    String s="javatpoint";
```

```
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:

Hello
strings
Demo
```

5.2 String Methods

Here is the list of methods supported by String class −

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	int compareTo(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.

5.3 String Buffer & String Builder Class in Java

5. 3.1 StringBuffer: StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

Syntax:

```
StringBuffer s = new StringBuffer("GeeksforGeeks");
```

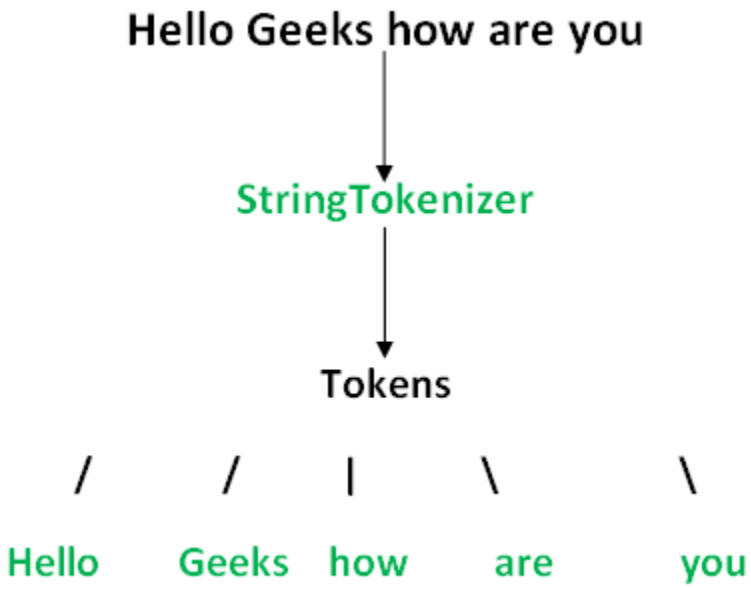
5.3.2 StringBuilder: The StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates and immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

Syntax:

```
StringBuilder str = new StringBuilder();

str.append("GFG");
```

5.3.3 StringTokenizer: StringTokenizer class in Java is used to break a string into tokens.

Example:



A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.

A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

5.3.4 StringJoiner: StringJoiner is a class in java.util package which is used to construct a sequence of characters(strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix. Though this can also be with the help of StringBuilder class to append delimiter after each string, StringJoiner provides an easy way to do that without much code to write.

Syntax:

public StringJoiner(CharSequence delimiter)

5.3.5 Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

5.3.6 Difference between StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

A list of differences between StringBuffer and StringBuilder are given below:

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer 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.

5.3.7 Immutable String in Java

In java, **string objects are immutable**. Immutable simply means un-modifiable or unchangeable.

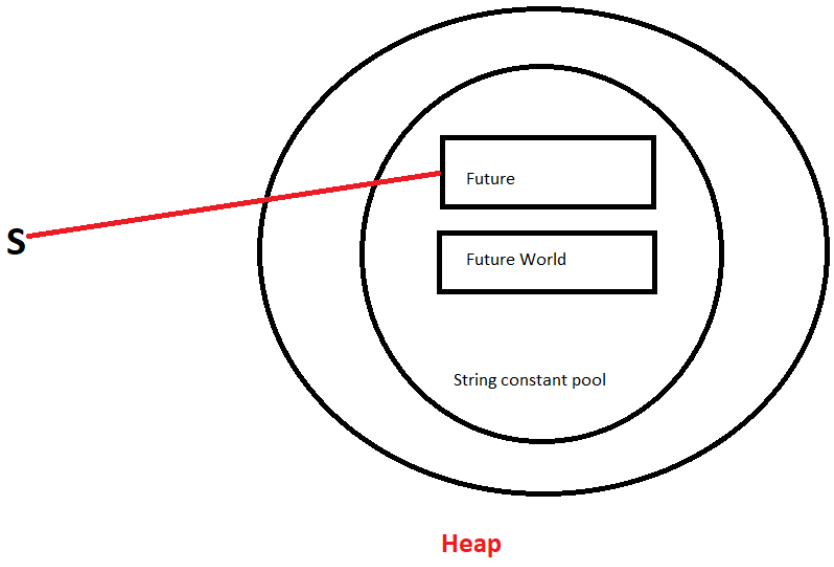
Once string object is created its data or state can't be changed but a new string object is created.

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 it can be understood by the diagram given below. Here Future is not changed but a new object is created with Future World. That is why 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".

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

5.3.8 Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all referes to one object "Future". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

5.4 Exception Handling

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

5.4.1 Why an exception occurs and How to Handle it?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc. in simple word it occurs because of the logical mistakes done by the programmer.

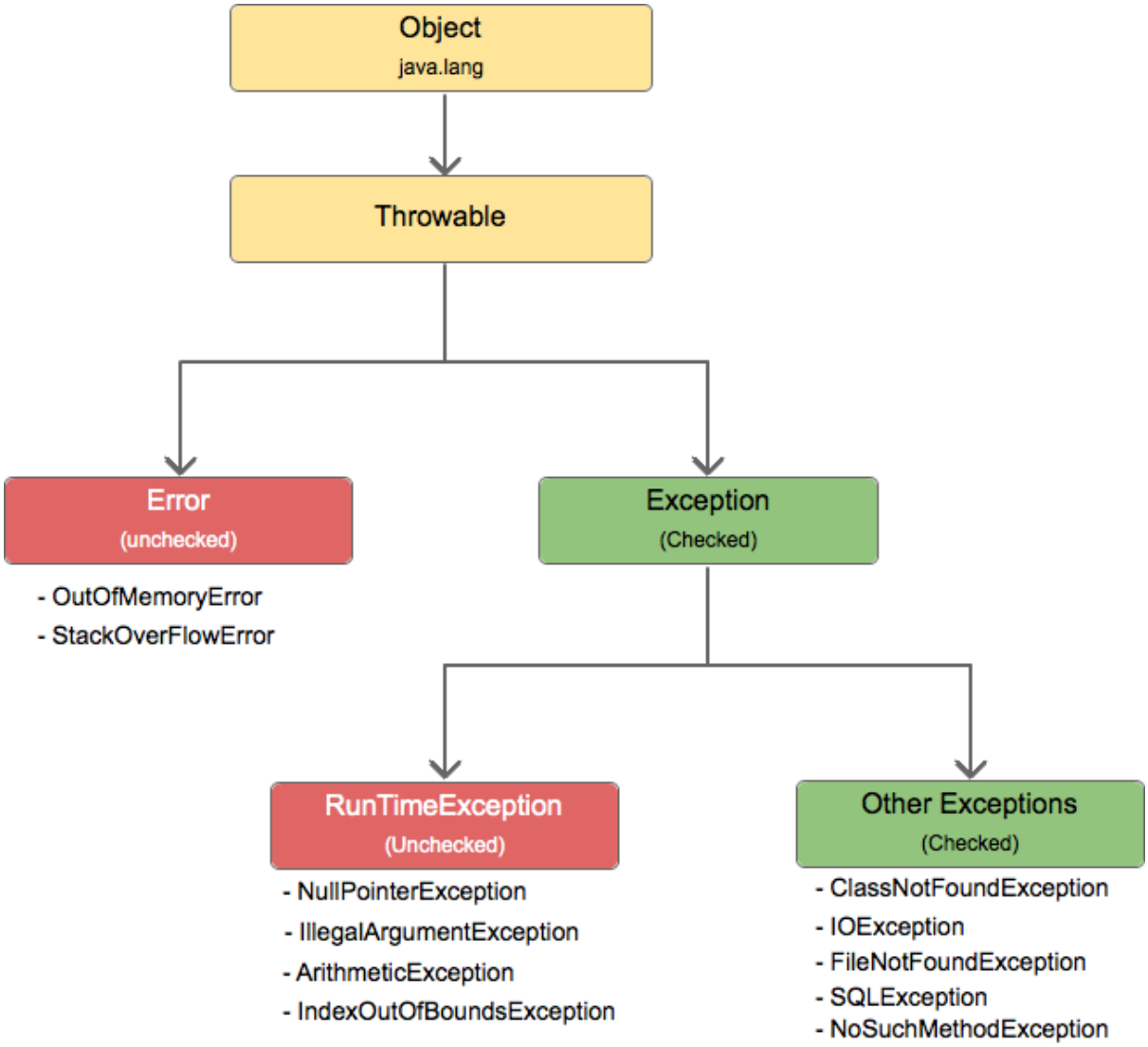
If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

5.4.2Advantage of exception handling

Exception handling ensures that the flow of the program doesn’t break when 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.By handling we make sure that all the statements execute and the flow of program doesn’t break.

5.4.3 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.



As shown in the exception hierarchy, a *Throwable* class is a superclass of all errors and exceptions in Java. Objects those are instances of *Throwable* or one of its subclasses are thrown by JVM or by Java throw statement.

5.5 Diffrence Between Exception and errors.

5.5.1 Error:

An Error is a subclass of *Throwable* that represents serious errors that can't be handled. A method is not required to declare throws clause for Error or any of its subclasses for the errors thrown during execution of the method but not caught.

Error and its subclasses are unchecked exceptions. Eroors may be occurred because of syntactical and symbolic mistakes done by the programmer .

Examples of errors: OutOfMemoryError, StackOverflowError, AssertionError, IOError, NoClassDefFoundError etc.

5.5.2 Exception:
word it occurs because of the logical mistakes done by the programmer. Exception is a subclass of *Throwable*. Exception and its subclasses represent the problems from which a program can recover and should be handled by the application.Exception has two sub type:

5.5.2.1 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 occurs at run-time, it can't be checked at compile time, an application should handle these exceptions.

For example: NullPointerException, ArithmeticException, IllegalArgumentException, IndexOutOfBoundException etc.

5.5.2.2 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. The classes that extends Throwable class except RunTimeExpception and Error are called Checked Exception.

For example: IOException, SQLException, FileNotFoundException, ClassNotFoundException, NoSuchMethodException etc.

5.5.2.3 User-defined custom exception:

We can also create our own exception. Here are few rules:

1. All exception must be a child of Throwable.
2. To create a RuntimeException, we need to extend the RunTimeException class.
3. To create the checked exception, we need to extend the Exception class.

5.5.3 Exceptions Methods

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

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.

5.6 Terminology Related to Exceptions

5.6.1 Try and Catch Blocks

Try Block: The "try" keyword is used to specify a block where we should place exception code. It means we write the The code which may cause exception (risky code) in try block. If exception occurs 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 followed by either catch or finally. It means, we can't use try block alone.

Catch: The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. 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 can be followed by finally block later.

Syntax:

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

5.6.2 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 previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second 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.

5.6.3 Finally: The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. It guarantees that the code written in finally block will surely execute even if exception occur or doesn't occur.

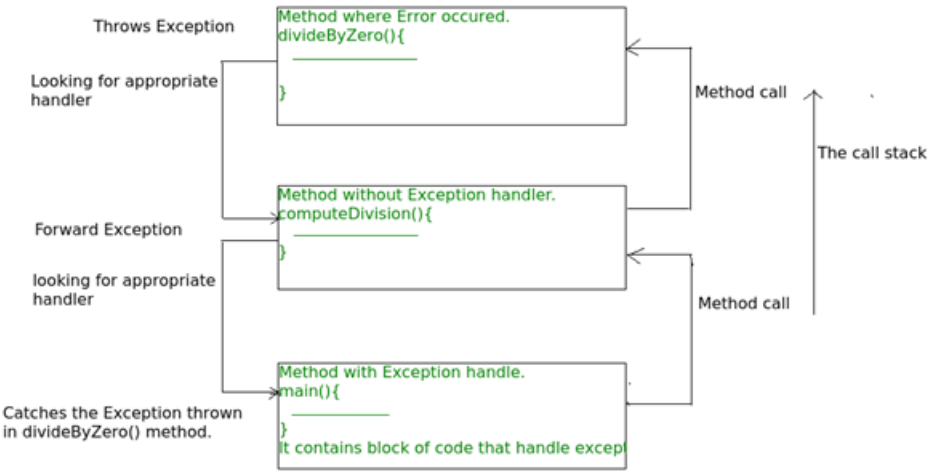
Syntax:

```
try {

  // Risky code

} catch (ExceptionType e1) {

  // Catch block

}

finally {

  // The finally block always executes.

}
```

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

5.6.5 Throws: The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies 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 doesn't have to handle the exception that will be handled by the compiler. It is always used with method signature.

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



Java Exception Handling Example

Let's see an example of Java Exception Handling by using a try-catch statement to handle the exception.

```
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.
```

5.7 Important Points to remember:

- 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 within own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is an exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** finally block.
- 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 if

exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

CHAPTER 6

PACKAGE AND DEFERRED IMPLIMENTATIONS

6.1 What is Package in Java?

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

ackages in Java are a way to encapsulate a group of classes, interfaces, enumerations, annotations, and sub-packages. Conceptually, you can think of java packages as being similar to different folders on your computer. In this tutorial, we will cover the basics of packages in Java.

When software is written in the Java programming language, it can be composed of hundreds or even thousands of individual classes. It makes sense to keep things organized by placing related classes and interfaces into packages.

Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Although interfaces and classes with the same name cannot appear in the same package, they can appear in different packages. This is possible by assigning a separate namespace to each package.

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

We can reuse existing classes from the packages as many time as we need it in our program.

6.1.1 How packages work?

Package names and directory structure are closely related. For example if a package name is *college.staff.cse,* then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example:

```
import java.util.*;
```

util is a subpackage created inside java package.

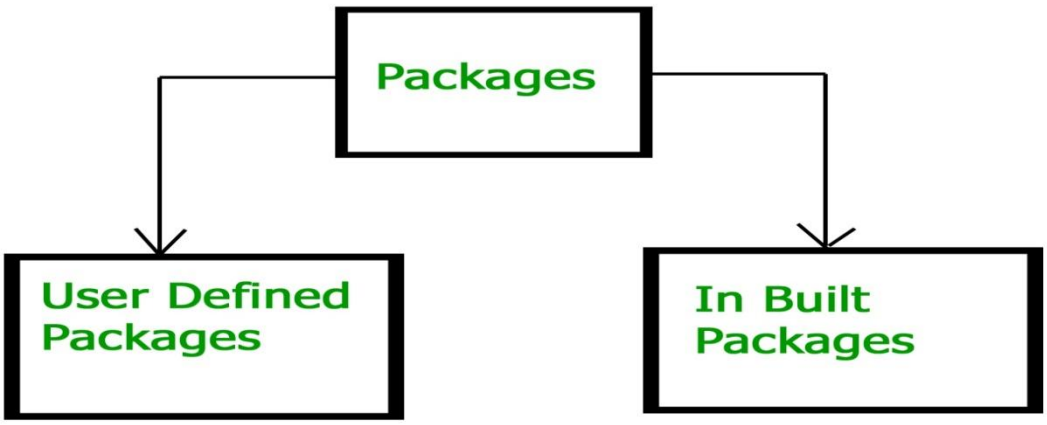
Accessing classes inside a package

Consider following two statements:

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

import java.util.vector;

// import all the classes from util package

import java.util.*;
```

6.1.2 Types of packages:

Packages



6.1.2.1 Built-in Packages

These packages 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: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

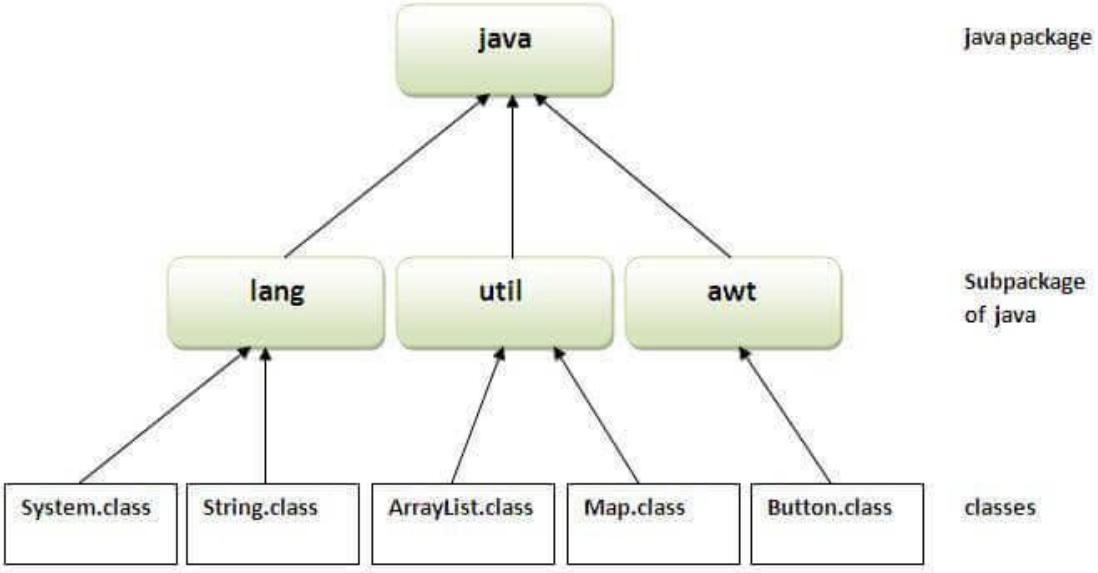
2) java.io: Contains classed for supporting input / output operations.

3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) java.applet: Contains classes for creating Applets.

5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) java.net: Contain classes for supporting networking operations.



6.1.2.2 User-defined packages

These are the packages that are defined by the user. First we create a directory myPackage (name should be same as the name of the package). Then create the MyClass inside the directory with the first statement being the package names.

6.2 Creating User Defined Packages:

we will see the the following program in which we will create the user defined package we will see how to compile to compile java package so that it can be used or imported whenever required.

```
// 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.

```
/* import '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

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
```

6.2.1 How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

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

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

6.2.2 How to run java package program

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 tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

6.3How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

6.3.1 Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make 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
```

6.3.2 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
```

6.3.3 Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import 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.

Import a Package

There are many packages to choose from. In the previous example, we used the 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, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the java.util package:

Example

import java.util.*;

6.3.3 Setting CLASSPATH:

CLASSPATH can be set by any of the following ways:

CLASSPATH can be set permanently in the environment: In Windows, 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 setting of the CLASSPATH, issue the following command:

```
> SET CLASSPATH
```

CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:

```
> SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
```

Instead of using the CLASSPATH environment variable, you can also use 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
```

6.4 Access Protection in Java Packages

You might be aware of various aspects of Java's access control mechanism and its access specifiers. Packages in Java add another dimension to access control. Both classes and packages are a means of data encapsulation. While packages act as containers for classes and other subordinate packages, classes act as containers for data and code. Because of this interplay between packages and classes, Java packages addresses 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 below gives a real picture of which type access is possible and which is not when using packages in Java:

	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 Important Points to Remember

- Every class is part of some package. If you omit the package statement, the class names are put into the default package
- A class can have only one package statement but it can have more than one import package statements
- The name of the package must be the same as the directory under which the file is saved
- When importing another package, package declaration must be the first statement, followed by package import

6.6 Abstract Classes and Interfaces

A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods(methods without body) 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. It needs to be extended and its method implemented. 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.

Abstraction is a process of hiding the implementation details and showing only functionality to the user. it shows only essential things to the user and hides the internal details, 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 lets you focus on what the object does instead of how it does it.

types to achieve Abstraction:

There are two ways to achieve abstraction in java

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:

```
//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

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about 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. We will learn about the factory method later.

In this example, 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
```

6.6.2Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- The methods which are marked as abstract need to be implemented in a class who is extending the abstract class.
-

6.6.3 Example of 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 methods
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

Like a class, an interface can have methods and variables, 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.

- 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. 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();
}
```

A real-world example:

Let's consider the example of vehicles like bicycle, car, bik etc. they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
importjava.io.*;

interfaceVehicle {

   // all are the abstract methods.
   voidchangeGear(inta);
   voidspeedUp(inta);
   voidapplyBrakes(inta);
}
```

```
classBicycle implementsVehicle{

   intspeed;
   intgear;

    // to change gear
   @Override
   publicvoidchangeGear(intnewGear){

      gear = newGear;
   }

   // to increase speed
   @Override
   publicvoidspeedUp(intincrement){

      speed = speed + increment;
   }

   // to decrease speed
   @Override
   publicvoidapplyBrakes(intdecrement){

      speed = speed - decrement;
   }

   publicvoidprintStates() {
       System.out.println("speed: "+ speed
           + " gear: "+ gear);
   }
}

classBike implementsVehicle {

   intspeed;
   intgear;

   // to change gear
   @Override
   publicvoidchangeGear(intnewGear){

      gear = newGear;
   }

   // to increase speed
   @Override
   publicvoidspeedUp(intincrement){

      speed = speed + increment;
   }

   // to decrease speed
   @Override
   publicvoidapplyBrakes(intdecrement){

      speed = speed - decrement;
```

```
    }

    publicvoidprintStates() {
        System.out.println("speed: "+ speed + " gear: "+ gear);
    }

}
classGFG {

    publicstaticvoidmain (String[] args) {

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

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

        // creating instance of the bike.
        Bike bike = newBike();
        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 does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfaces. In interfaces, a class can implement more than one interface which can't be done through extends keyword.Let's say we have two interfaces with same method name (demo) and different return types(int and String)

```
publicinterfaceInterfaceX
{
    publicint demo ();
}
publicinterfaceInterfaceY
```

```
{
   publicString demo ();
}
```

Now, Suppose we have a class that implements both those interfaces:

```
publicclassTesting implementsInterfaceX, InterfaceY
{
publicString demo ()
   {
      return"hello";
   }
}
```

Example of multiple interfaces Implementation in Java

* Implementation of multiple interfaces java example */

```
interface Flyable {
        void fly();
}
interface Eatable {
        void eat();
}
```

// Bird class will implement both interfaces

```
class Bird implements Flyable, Eatable {


        public void fly() {
                System.out.println("Bird flying");
        }
        public void eat() {
                System.out.println("Bird eats");
        }
        // It can have more own methods.
```

```
}
```

* Test multiple interfaces example */

```
public class Sample {
        public static void main(String[] args) {
                Bird b = new Bird();
                b.eat();
                b.fly();
        }
}
```

Output:
Bird eats
Bird flying

6.8 Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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**.	erface has **only static and final variables**.
Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
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 like private, protected, etc.	Members of a Java interface are public by default.
Example: public abstract class Shape{	**Example:** public interface Drawable{

public abstract void draw(); }	void draw(); }

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

6.9 Generalization, Specialization, and Inheritance

6.9.1 Generalization

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods. Generalization is the bottom-up process of abstraction, where we club the differences among entities according to the common feature and generalize them into a single superclass. The original entities are thus subclasses of it.

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 Car is a Vehicle, there will be no objection

6.9.2 Specialization

Converting a super class type into a sub class type is called '**Specialization**'. Here, 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.
Narrowing is **not safe** because the classes will become more and more specific thus giving rise to more and more doubts. For example if we say Vehicle is a Car 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.

In contrast to generalization, *specialization* means creating new subclasses from an existing class. 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.

CHAPTER 7

JAVA INPUT-OUTPUT

7.1 Introduction:

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files. 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. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. 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.

7.1.1 Constructors of Class

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.

7.1.2 Methods in a file class

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.

Example 1:

```
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 check if a file or directory physically exist or not.

In this program, we accepts a file or directory name from, command line arguments. Then the program will check if that file or directory physically exist or not and it displays the property of that file or directory.

```
*import java.io.File;

 // Displaying file property

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

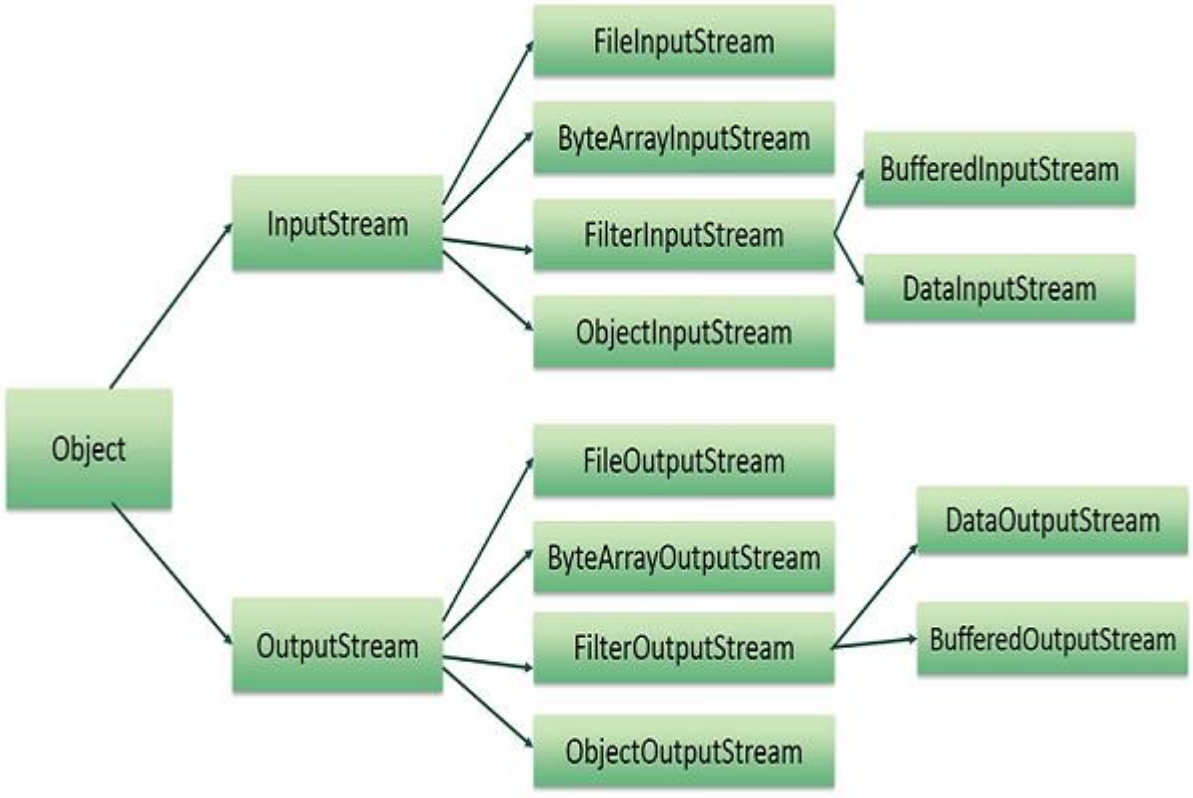
Is a directory:false

File Size in bytes 20

7.2 Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** 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.



7.2.1 Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

7.2.2 FileOutputStream

FileOutputStream is used to 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.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
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. Ensures that the close method of this file output stream is called when there are no more references to this stream. 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 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.
```

7.2.3 FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
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. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}:** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}:** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}:** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- Java FileInputStream example 2: read all characters
- package com.javatpoint;

```
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

7.3 BufferedWriter and BufferedWriter Class

7.3.1 Java BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

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.

Class methods

Method	Description
void newLine()	It is used to add a new line by writing a line separator.
void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

```
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.

7.3.2 Java BufferedReader Class

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

Java BufferedReader class declaration

Let's see the declaration for Java.io.BufferedReader class:

1. **public class** BufferedReader **extends** Reader

Java BufferedReader class constructors

Constructor	Description
BufferedReader(Reader rd)	It is used to create a buffered character input stream that uses the default size for an input buffer.
BufferedReader(Reader rd, int size)	It is used to create a buffered character input stream that uses the specified size for an input buffer.

Java BufferedReader class methods

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array.
boolean markSupported()	It is used to test the input stream support for the mark and reset method.
String readLine()	It is used for reading a line of text.
boolean ready()	It is used to test whether the input stream is ready to be read.
long skip(long n)	It is used for skipping the characters.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	It is used for marking the present position in a stream.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

Java BufferedReader 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.
```

7.3.3 Reading data from console by InputStreamReader and BufferedReader

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading 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
```

7.4 Serialization and Deserialization in Java

Serialization in Java is a mechanism of writing the state of an object into a byte-stream.

The reverse operation of serialization is called deserialization where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

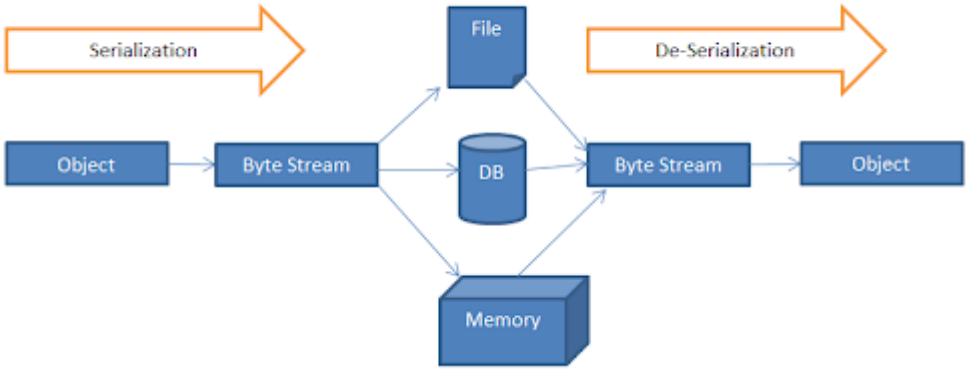
For serializing the object, we call the **writeObject()** method ObjectOutputStream, and for deserialization we call the **readObject()** method of ObjectInputStream class.

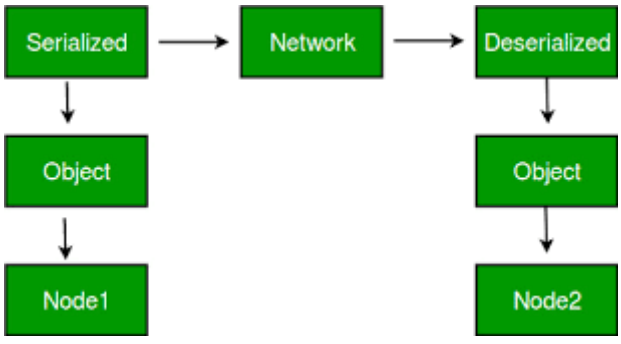
7.4.1 Advantages of Serialization

1. To save/persist state of an object.

2. To travel an object across a network.

3. Serialized Object Can be stored in File, database and can be desterilized when ever required.





Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface.

7.4.2 java.io.Serializable interface

Serializable is a marker interface (has no 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.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

7.4.3 ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor:

Constructor	Description
public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
public void flush() throws IOException {}	flushes the current output stream.
public void close() throws IOException {}	closes the current output stream.

7.4.4 ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

Constructor	Description
public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	Description
public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
public void close() throws IOException {}	closes ObjectInputStream.

Let's see the example given below:

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

Example of Java Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality 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
  Student s1 =new Student(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
```

7.5 Example of Java Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an 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"));
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(“Student ID:”+s.id);
System.out.println("Student Name:  "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
Output:
Student ID: 11

Student name: Rama
```

7.6 Java Scanner

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Java Scanner Class Constructors

SN	Constructor	Description
1)	Scanner(File source)	It constructs a new Scanner that produces values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner that produces values scanned from the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified input stream.
5)	Scanner(Readable source)	It constructs a new Scanner that produces values scanned from the specified source.
6)	Scanner(String source)	It constructs a new Scanner that produces values scanned from the specified string.
7)	Scanner(ReadableByteChannel source)	It constructs a new Scanner that produces values scanned from the specified channel.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified channel.
9)	Scanner(Path source)	It constructs a new Scanner that produces values scanned from the specified file.
10)	Scanner(Path source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.

Java Scanner Class Methods:

The following are the list of Scanner methods:

SN	Method	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()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

6)	boolean hasNext()	It returns true if this scanner has another token in its input.
7)	boolean hasNextBigDecimal()	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.
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()	It 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()	It is used 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()	It 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()	It is used 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()	It is used to check if there is another line in the input of this scanner or not.
15)	boolean hasNextLong()	It 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()	It is used 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 get 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()	It scans the next token of the input as a BigDecimal.
22)	BigInteger nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	byte nextByte()	It scans the next token of the input as a byte.

25)	double nextDouble()	It scans the next token of the input as a double.
26)	float nextFloat()	It scans the next token of the input as a float.
27)	int nextInt()	It scans the 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()	It scans the next token of the input as a long.
30)	short nextShort()	It scans the next token of the input as a short.
31)	int radix()	It is used to get the default radix of the Scanner use.
32)	void remove()	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner reset()	It 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 which is in use to the specified radix.

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
```

CHAPTER-8

THREAD, GENERICS AND COLLECTIONS

8.1 Introduction:

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and 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 concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

8.3 Types to create the Thread:

1. Extending the Thread class
2. Implementing the Runnable Interface

8.3.1 Thread creation by extending the Thread class:

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
/ Java code for thread creation by extending
// the Thread class
classMultithreadingDemo extendsThread
{
   publicvoidrun()
   {
      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
publicclassMultithread
{
    publicstaticvoidmain(String[] args)
    {
        intn = 8; // Number of threads
        for(inti=0; i<8; i++)
        {
            MultithreadingDemo object = newMultithreadingDemo();
            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

8.3.2 Thread creation by implementing the Runnable Interface:

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing
```

```
// the Runnable Interface
classMultithreadingDemo implementsRunnable
{
   publicvoidrun()
   {
      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
classMultithread
{
   publicstaticvoidmain(String[] args)
   {
      intn = 8; // Number of threads
      for(inti=0; i<n; i++)
      {
         Thread object = newThread(newMultithreadingDemo());
         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
```

8.3.3 Lifecycle and States of a Thread in Java

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.

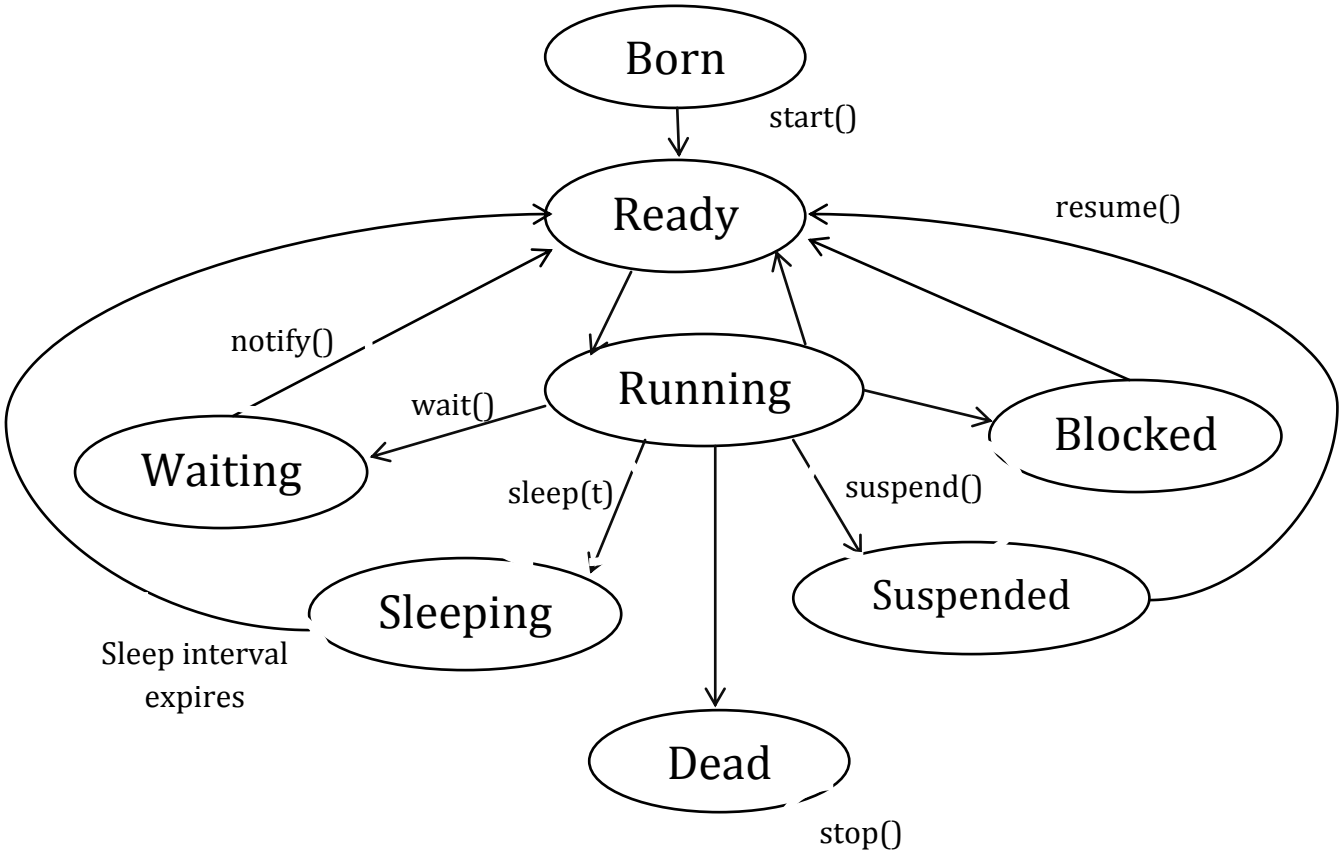


Figure: Life Cycle of a thread

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.

Runnable State (Ready): A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. Thread enter in to runnable state after calling start() function from thread class.

Running: When run() function is called from thread function is enter in to running state. A multi-threaded program allocates a fixed amount of time to each individual thread. 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. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

Waiting state: When a thread is temporarily inactive, then it's in one of the following states:

Waiting

Blocked: When a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked thread. A thread in this state cannot continue its execution any further until 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.

A thread is in the blocked state 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 one of the thread which is blocked for that section and moves it to the runnable state.

Waiting State: a thread is in the waiting state when it waits for another thread on a condition. Another thread may call wait() function. When this condition is fulfilled, the scheduler is notified by notify() function and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

Timed waiting (sleeping): A thread lies in timed waiting state when it calls a method sleep(t) with a time out parameter in nanoseconds. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

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.

Terminated (Dead) State: A thread terminates because of either of the following reasons:

- Because it exists normally. This happens when the code of thread has entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.
- A thread that lies in a terminated state does no longer consume any cycles of CPU.

Constructors in Thread Class:

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)**: Allocates a new Thread object
4	**Thread(String name)**: Allocates a new Thread object
5	**Thread(ThreadGroup group, Runnable target)**: Allocates a new Thread object
6	**Thread(ThreadGroup group, Runnable target, String name)**: 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)**: 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

Methods in Thread Class:

Sr. No.	**Method & Descriptions**
1	**activeCount():** Returns an estimate of the number of active threads in the current thread's thread group and its subgroups

2	**checkAccess():** 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():** Returns a reference to the currently executing thread object
5	**dumpStack():** Prints a stack trace of the current thread to the standard error stream
6	**getAllStackTraces():** 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():** Returns an array of stack trace elements representing the stack dump of this thread
11	**getState():** Returns the state of this thread
12	**getThreadGroup():** 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():** Returns a string representation of this thread, including the thread's name, priority, and thread group
20	**start():** 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():**

8.4 Java Thread Priority in Multithreading

In a Multi-threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.

public static int MIN_PRIORITY: This is minimum priority that a thread can have. Value for this is 1.
public static int NORM_PRIORITY: This is default priority of a thread if do not explicitly define it. Value for this is 5.
public static int MAX_PRIORITY: This is maximum 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 of given 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);

        // t3.setPriority(21); will throw IllegalArgumentException
        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
```

8.5 Synchronization in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. 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 attempting to enter the synchronized block are blocked 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

}
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi-threading with synchronized.

```
// 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
```

8.6 Generic Methods & Classes in java

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Generics in Java are similar to templates in C++. The idea 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 for definition and use of **generic** types and methods." **Generic** types are instantiated to form parameterized types by providing actual type arguments that replace the formal type parameters. A class like LinkedList<E> is a **generic** type that has a type parameter E.

8.6.1 Advantage of Java Generics

1) Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

4) Code Reusability: it allows code reusability as you can reuse the same code for different Types of data.

8.6.2 Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note 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
```

8.6.3 Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

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

```
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

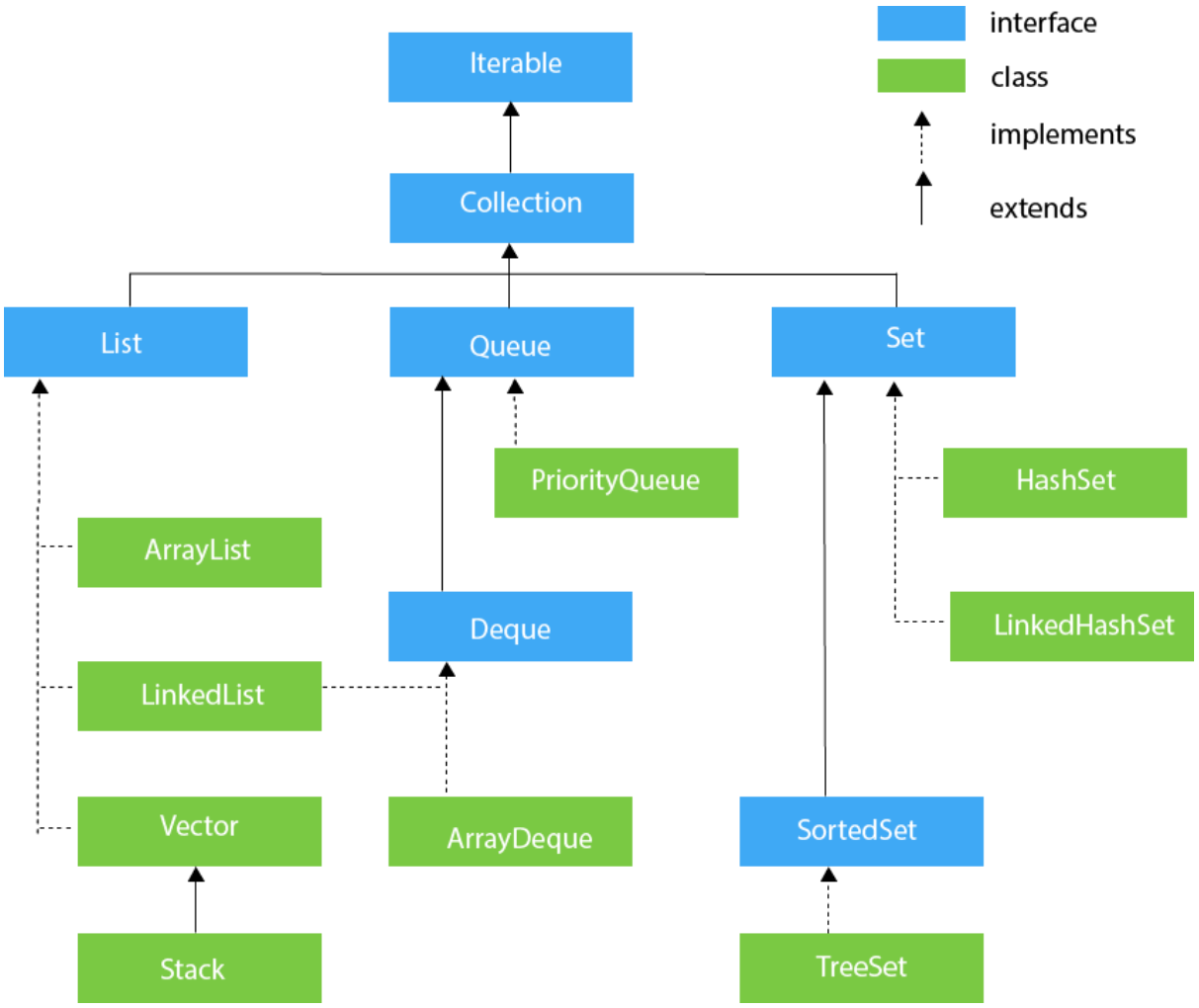
8.7 Collections in Java:

A collection, as name implies, is group of objects. **Java Collections framework** is consist of the interfaces and classes which helps in working with different types of collections such as **lists, sets, maps, stacks and queues** etc.

These ready-to-use collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects. The common operations in involve **add, remove, update, sort, search** and more complex algorithms. These collection classes provide very transparent support for all such operations using **Collections APIs**.

8.8 Java Collections Hierarchy

The Collections framework is better understood with the help of **core interfaces**. The collections classes implement these interfaces and provide concrete functionalities.



1.1. Collection

Collection interface is at the root of the hierarchy. Collection interface provides all general purpose methods which all collections classes must support (or throw `UnsupportedOperationException`). It **extends Iterable** interface which adds support for iterating over collection elements using the "**for-each loop**" statement.

All other collection interfaces and classes (except Map) either extend or implement this interface. For example, List *(indexed, ordered)* and Set *(sorted)* interfaces implement this collection.

1.2. List

Lists represents an **ordered** collection of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with `0`, just like an array.

Some useful classes which implement `List` interface are – **ArrayList**, **CopyOnWriteArrayList**, **LinkedList**, **Stack** and **Vector**.

1.3. Set

Sets represents a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not provides no guarantee to return the elements in any predictable order; though some Set implementations store elements in their natural ordering and guarantee this order.

Some useful classes which implement `Set` interface are – **ConcurrentSkipListSet**, **CopyOnWriteArraySet**, **EnumSet**, **HashSet**, **LinkedHashSet** and **TreeSet**.

1.4. Map

The **Map** interface enable us to store data in *key-value pairs* (keys should be immutable). A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Some useful classes which implement `Map` interface are – **ConcurrentHashMap**, **ConcurrentSkipListMap**, **EnumMap**, **HashMap**, **Hashtable**, **IdentityHashMap**, **LinkedHashMap**, **Properties**, **TreeMap** and **WeakHashMap**.

1.5. Stack

The Java **Stack** interface represents a classical stack data structure, where elements can be pushed to last-in-first-out (LIFO) stack of objects. In Stack we push an element to the top of the stack, and popped off from the top of the stack again later.

1.6. Queue

A queue data structure is intended to hold the elements (put by producer threads) prior to processing by consumer thread(s). Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. One such exception is priority queue which order elements according to a supplied Comparator, or the elements' natural ordering.

In general, 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, PriorityQueue and SynchronousQueue.

1.7. Deque

A double ended queue (pronounced "***deck***") that supports element insertion and removal at both ends. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. When a deque is used as a stack, LIFO (Last-In-First-Out) behavior results.

This interface should be used in preference to the legacy Stack class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

Some common known classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.

8.9 ArrayList, Vector and linked List

8.9.1ArrayList in Java:

An **ArrayList in Java** represent a resizable list of objects. We can add, remove, find, sort and replace elements in this list. ArrayList is part of Java's collection framework and implements Java's **List** interface.

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 it's current size.
4. Non synchronized – ArrayList is not synchronized, by default. Programmer needs to use synchronized keyword appropiately or simply use Vector class.
5. Duplicates allowed – We can add duplicate elements in arraylist. It is not possible in sets.

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

Methods in ArrayList

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): Returns true if this list contains the specified element.

5	remove(int index): Removes the element at the specified position in this list.
6	remove(Object o): Removes the first occurrence of the specified element from this list, if it is present.
7	get(int index): 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(): Creates a late-binding and fail-fast Spliterator over the elements in this list.
10	set(int index, E element): 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.

```
// 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
```

8.9.2 Linked List in Java

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

LinkedList Features:

- **Doubly linked list** implementation which implements List and Deque interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
- Permits all elements including duplicates and NULL.
- LinkedList maintains the **insertion order** of the elements.
- 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.
- Use Collections.synchronizedList(new LinkedList()) to get synchronized linkedlist.
- The iterators returned by this class are fail-fast and may throw `ConcurrentModificationException`.
- It does not implement RandomAccess interface. So we can access elements in sequential order only. It does not support accessing elements randomly.
- We can use ListIterator to iterate LinkedList elements

Following are the constructors supported by the LinkedList class.

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**.

Apart from the methods inherited from its parent classes, LinkedList defines following methods –

Sr.No.	Method & Description
1	**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()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**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.
4	**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.
5	**void addFirst(Object o)** Inserts the given element at the beginning of this list.

6	**void addLast(Object o)** Appends the given element to the end of this list.
7	**void clear()** Removes all of the elements from this list.
8	**Object clone()** Returns a shallow copy of this LinkedList.
9	**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)).
10	**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()).
11	**Object getFirst()** Returns the first element in this list. Throws NoSuchElementException if this list is empty.
12	**Object getLast()** Returns the last element in this list. Throws NoSuchElementException if this list is empty.
13	**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.
14	**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.
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]
```

8.9.3 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.

Features of Vector Class

- The size of a Vector can grow or shrink as needed to accommodate adding and removing items.
- Each vector tries to optimize storage management by maintaining a *capacity* and a *capacityIncrement*.
- 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.

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.

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:

```
/ Java code illustrating add() method
importjava.util.*;
classVector_demo {
   publicstaticvoidmain(String[] arg)
   {

      // create default vector
      Vector v = newVector();
```

```
        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]

8.10 Java PriorityQueue class:

Java PriorityQueue class is a queue data structure implementation in which objects are processed based on their **priority**. It is different from standard queues where FIFO (First-In-First-Out) algorithm is followed.

In a priority queue, added objects are according to their priority. By default, the priority is determined by objects' natural ordering. Default priority can be overridden by a Comparator provided at queue construction time.

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 overridden 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.

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.

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.

```
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 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.

The methods declared by SortedMap are summarized in the following table −

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.

```
// Java code to demonstrate SortedMap

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
```