COREJAVA MATERIAL

Name: BV Narayana M Tech
E-mail: bvnarayana28@gmail.com
Mobile: 99899 31232
Date: 15th Aug 2015

Introduction to software:

Software:

- A technical definition of software is, it is collection of different programs which are designed to perform a particular task.
- A general definition of software is a conversion of a requirement into an application.
- A Program is a set of different instructions.
- Software will reduce the human task and make the business automated.
- By using software, we can remove disadvantage of book keeping system.
- Disadvantages of book keeping system: Manual Process
  Time Consuming
  No Exact results
  No Security
  No Sharing

We have following 2 types of software's:

1. System software

- System software: If any software is developed for the purpose of making any hardware devices work, such kind of software's are called as system software.
- System software's are generally developed in languages like C, C++, ...

  Eg: OS, printer drivers, web cam drivers, Compilers and etc.

2. Applications software

- If any software is developed for the purpose any user to complete his requirements like maintaining his business, for entertainment, for storage called as application software's.
- application software's can be developed by using languages like java,.net , …

  Eg: notepad, wm player, calc, bank s/w, hosp s/w, SM s/w, ...

Java is released by sun micro systems into the market in following 3 editions.

JSE: JSE stands for Java standard edition which can be used for developing stand alone applications.

JEE: JEE stands for java enterprise edition which can be used for developing web based applications.

JME: JME stands for java mobile edition or micro edition which can be used for developing software's for mobile devices or embedded controllers.

In Java we can develop Following 2 types of application software's.

1. Stand alone applications:

- Stand alone applications are available in client system and runs in the same client system which are also called as client side applications
- Stand alone applications are specific to single system which are also called as off-line applications .

Eg:

MS-office, ….

2. web based applications:

- web based applications are available in server system and runs in the same client system which are also called as server side applications
- web based applications are not specific to single system which are also called as on-line applications .
- web based application always runs in the context of browser.

Ex: Facebook, Gmail, online games and etc.

Topic1. Introduction to Java, History of Java and Features of Java:

1. A **programming language** is a formal constructed language designed to communicate instructions to a machine, particularly a computer. (or)
   A programming language is a set of commands, instructions, and other syntax use to create a software program.

Languages that programmers use to write code are called "high-level languages." This code can be compiled into a "low-level language," which is recognized directly by the computer hardware.

2. Programming languages can be used to create programs to control the behavior of a machine.
3. A programming language is a notation for writing programs, which are specifications of a computation or algorithm.

We have two languages:

1) Procedural Languages: The languages which hold the following three features can be known as procedural languages.

Procedural Language: It specifying order of procedure i.e we have to specify everything Ex. C, C++, Java

2) Non Procedural Languages: The languages which do not hold the features which are mentioned in procedural language, are known as non-procedural languages. The most common example of non-procedural language is SQL (Structured Query Language). In SQL, only commands are used to implement ant task.

Non-Procedural Language (Structured Query Language): It ordering without specifications. Ex. Oracle, SQL

Earlier days we are using HTML for Front End design.

It can create only static and plain pages so if we need dynamic pages (displaying another page or another value for the given value) then HTML is not useful.

Dynamic pages: including another page into current page.

It cannot produce dynamic output alone, since it is a static language

Introduction to Java:

Java introduced applet instead of HTML for front end.

a) A Java applet is an applet delivered in the form of Java byte code.

b) Java applets can run in a Web browser using a Java Virtual Machine (JVM), or in Sun's AppletViewer, a stand-alone tool for testing applets.

c) Java applets were introduced in the first version of the Java language in 1995. Java applets are usually written in the Java programming language but they can also be written in other languages that compile to Java bytecode such as Jython.

d) Applets are used to provide interactive features to web applications that cannot be provided by HTML. Since Java's bytecode is platform independent, Java applets can be executed by browsers for many platforms, including Windows, Unix, Mac OS and Linux. There are open source tools like applet2app which can be used to convert an applet to a standalone Java application/windows executable/linux executable.

e)

Drawbacks of applet:

a) To View applet we must install java software in browsers.

b) It requires the Java plug-in.

c) Some browsers, notably mobile browsers running Apple iOS or Android do not run Java applets at all.

d) Some organizations only allow software installed by the administrators. As a result, some users can only view applets that are important enough to justify contacting the administrator to request installation of the Java plug-in.

e) So as to run a Java applet you need Java Plug-in which is not available by default on all browsers.

f) Applets cannot start until java Virtual Machine is running. If used for first time it takes significant startup time.

g) **The first browser introduced in the world is Netscape with java plug-in.**

Netscape: This browser will be available with default java software plug-in.

Designers are also facing the problems

Java Script: Dynamic pages with animation and without database.

a) JavaScript started life as LiveScript, but Netscape changed the name, possibly because of the excitement being generated by Java.to JavaScript.

b) JavaScript made its first appearance in Netscape 2.0 in 1995 with a name *LiveScript*.

c) JavaScript is a lightweight, interpreted programming language with object-oriented capabilities that allows you to build interactivity into otherwise static HTML pages.

d) The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers

JavaScript is:

- JavaScript is a lightweight, interpreted programming language
- Designed for creating network-centric applications
- Complementary to and integrated with Java
- Complementary to and integrated with HTML
- Open and cross-platform

Client-side JavaScript:

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML document for the code to be interpreted by the browser.

It means that a web page need no longer be static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

Perl: Same as HTML and with database. And it is a process based.

a) Perl 5 is used for graphics programming, system administration, network programming, finance, bioinformatics, and other applications.

b) Process contains three areas stack, code and Data areas.

c) For every user it creates process.

Drawback: If no. of users is increasing then no. of database connections is also increases on server.

Java with Threads: Java introduced Threading concept

Now java starts working with threading concept and it reduces the drawback of Perl programming language.

Advantage of threading concept: number of database connections are reduced in java Because All threads communicate only one data area and this data communicates with server.
So number of connections is reduced with server.

Due to thread concept we can able to process more than one request at a time i.e. Java uses multithreading concept to process more than one than request at a time.

History of Java:

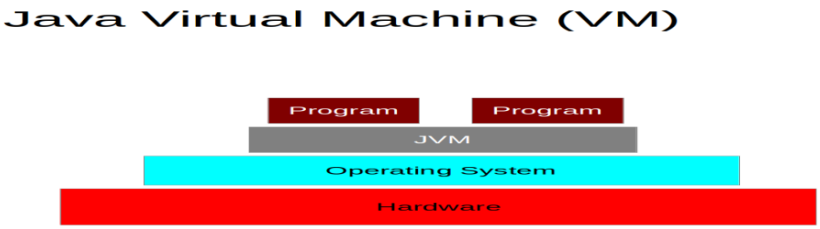
a) James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside

Gosling's office, also went by the name Green and ended up later being renamed as Java, from a list of random words.

b) Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.
c) On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
d) On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Author	:	**James Gosling**
Vendor	:	Oracle (founder -Sun Micro System)
SUN	:	Stanford Universally Network
Project name	:	Green Project
Type	:	open source(free software)
Initial Name	:	OAK language
Present Name	:	java
Extensions	:	.java, .class, .jar, .war & .ear
Initial version	:	java1.0 or JDK1.0 (Java Development Kit)
Present version	:	java1. 8
Operating System	:	Multi Operating System
Implementation Lang	:	c, cpp……
Symbol	:	coffee cup with saucer
Objective	:	To develop web applications
Slogan/Motto	:	WORA (write once run anywhere)

a) **Java** is a general-purpose computer programming language that is concurrent, class-based and object-oriented .
b) It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another.
c) Java applications are typically compiled to byte code that can run on any Java Virtual Machine (JVM) regardless of computer architecture.
d) The Java Virtual Machine (JVM) is a software implementation of a computer that executes programs like a real machine.
e) The Java virtual machine is written specifically for a specific operating system, e.g., for Linux a special implementation is required as well as for Windows.



f) Java programs are compiled by the Java compiler into *bytecode*. The Java virtual machine interprets this *bytecode* and executes the Java program.

Example program:

```
Class Test
    {
        public static void main (String args[])
            {
                System.out.println ("BV Narayana");
             }
    }
```

Compilation: javac filename.java/classname.java i.e. javac Test.java
Execution: java classname/filename i.e. java Test
Output: BV Narayana

Importance of core java:

According to the SUN 3 billion devices run on the java language only.

1) Java is used to develop Desktop Applications such as MediaPlayer, Antivirus etc.

2) Java is Used to Develop Web Applications such as naukri.com, irctc.co.in etc.

3) Java is Used to Develop Enterprise Application such as Banking applications.

4) Java is Used to Develop Mobile Applications.

5) Java is Used to Develop Embedded System.

6) Java is Used to Develop SmartCards.

7) Java is Used to Develop Robotics.

8) Java is used to Develop Games etc.

Technologies Depends on Core java:

a) Advance java
b) Hadoop
c) Android
d) Sales force(Cloud Computing)
e) Selenium Testing tool
f) Embedded Systems

Java Versions with released years:

Java Alpha & beta: 1995

JDK 1.0: 1996

JDK1.1: 1997

J2SE 1.2: 1998

J2SE 1.3: 2000

J2SE 1.4: 2002

J2SE 1.5: 2004

JAVA SE 6: 2006

JAVA SE 7: 2011

JAVA SE8: 2014

Features of Java language:

- Every programming language provided some facility's or services which are called as features of that programming language.
- java language provides following various features

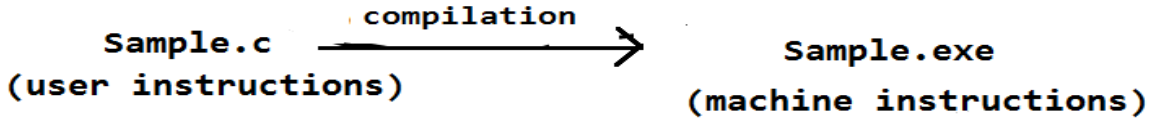
1. **Simple**:
   The java programming language is called as a simple programming language because of the following reasons.
   1. The complex topics like pointers, templates, virtual functions, friend functions,.... are eliminated in java language making it simple.
   2. The syntax of java language are simple to read , understand and we can easily implement in the application which are similar to syntaxes of C, C++

2. **Java is a Platform Independent or Architectural Neutral (due to Portable):**
   - When an application is developed in java language if it can be executed in any machine without considering the operating system, without considering the Architecture, without considering the software and hardware etc.

translators

- a translator is a program which is used to translate or convert user instructions of the program into machine understandable instructions so that we can execute our program.
- we have following 2types of translators

1 .compiler: compiler will translate the program all lines the program at a time.
2 .Interpreter: Interpreter which will translate the program line by line of program.

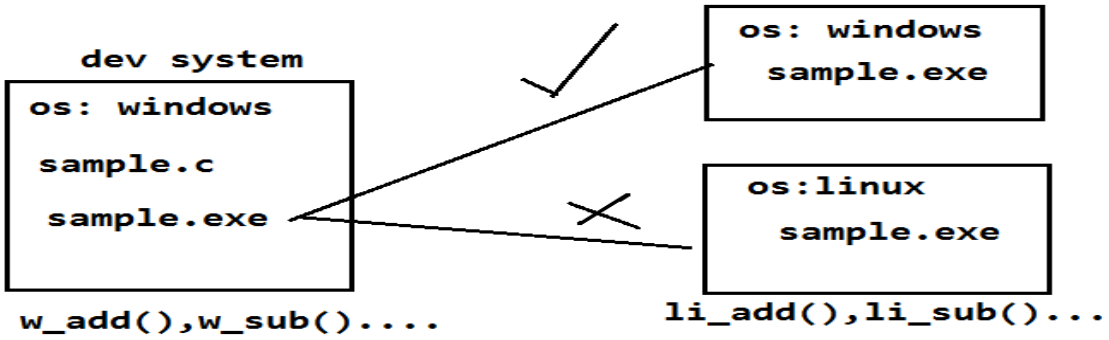


C language does not support this Platform Independent

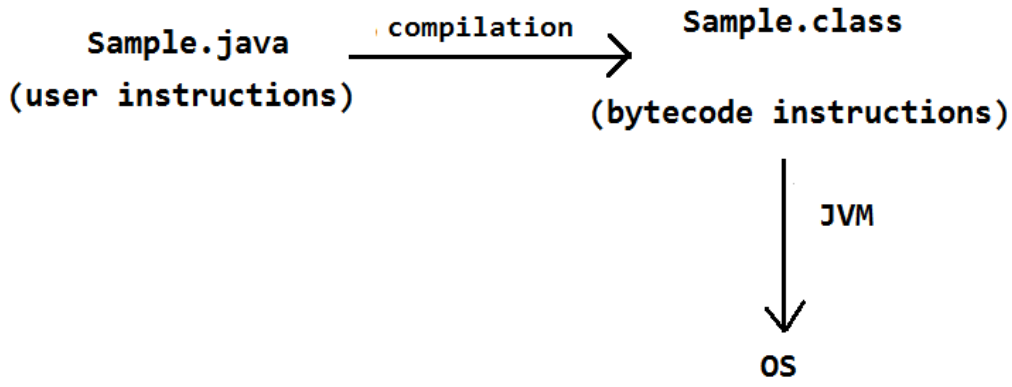
- When we compile C or C++ programs then compiler will generate .exe files

  which contains machine instructions
- This compiled code or .exe file can be executed only in compatible machines.

But if we execute the same .exe file in another incompatible machine it can not be executed hence we can call c or c++ applications are platform or machine dependent.

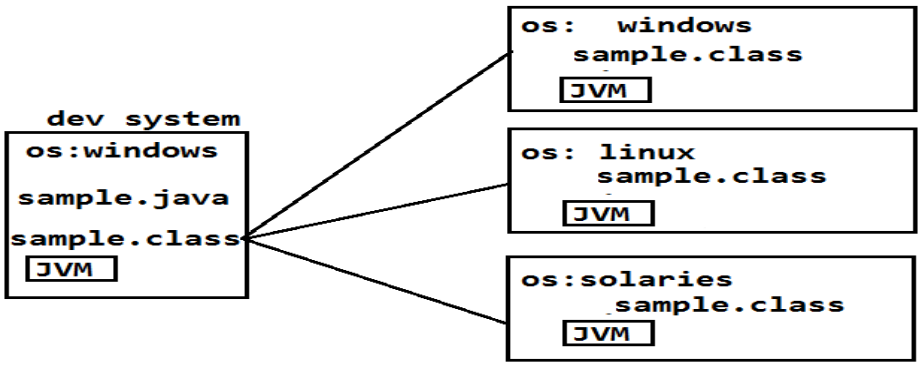


But Java language support this Platform Independent



- When we compile java program then java compiler will generate byte code instructions.
- Byte code instructions are not machine instructions.
- Byte code instructions are special java instructions which are ready to convert into machine instructions

- we can execute this byte code instructions in both compatible and incompatible machines by taking the help of JVM(java virtual machine)hence java applications are called platform independent application.

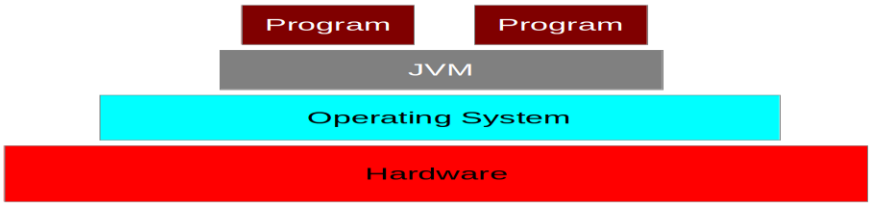


Java programs use the Java virtual machine as abstraction and do not access the operating system directly. This makes Java programs highly portable. A Java program (which is standard-compliant and follows certain rules) can run unmodified on all supported platforms, e.g., Windows or Linux. Generally we say java is a platform independent.

The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine.

The Java virtual machine is written specifically for a specific operating system, e.g., for Linux a special implementation is required as well as for Windows.

Java Virtual Machine (VM)



Java programs are compiled by the Java compiler into *bytecode*. The Java virtual machine interprets this *bytecode* and executes the Java program.

3. Components:

1) JDK (Java Development Kit): It is required software to get javac, java, javap and other tools.

2) JRE (Java Runtime Environment): It is used to for executing the application. It consists JVM and library files (eg. Jar files).

Note: The classes often contained in library files (like Java's JAR file format) and the instantiated objects residing only in memory.

JAR (**J**ava **AR**chive) is a package file format typically used to aggregate many Java class files and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.

3) Java API: It is used for developing the application. Java API consisting of set of packages, a package internally contains set of classes & internally a class contains set of methods used for developing java application.

4. Compiled & interpreted(using javac & java tools):

Interpreted and compiled language: Java source code is transferred into the bytecode format which does not depend on the target platform. These bytecode instructions will be interpreted by the Java Virtual machine (JVM). The JVM contains a so called Hotspot-Compiler which translates performance critical bytecode instructions into native code instructions.

Java is both compiled and interpreted; it means java compiler is used for converting java code into byte code and java interpreter is used for converting byte code into machine code.

Note: Name of java compiler is javac
Name of java interpreter is JVM (we will use java tool to execute program)

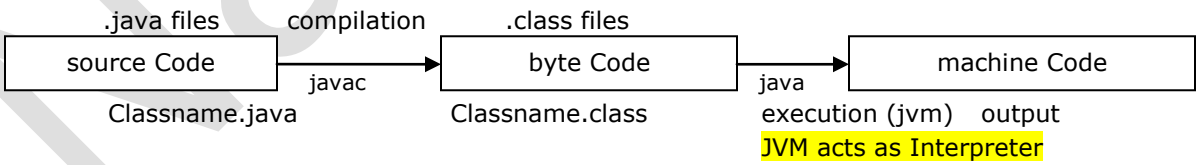
How to compile and execute any java program:

1. Open notepad or editplus or IDE to write program.
2. Save the program with .java extension.
3. Compile the program with javac tool with followed by classname.java
4. Execute the program with java tool followed by classname.

Whenever java program was compiled then it immediately generates a mediator code called byte code. Later this byte code will be converted into the format of machine code that can be understood by operating system.

When we compile any .java file then we get .class file(byte code) which is system independent.
Java source code ---------compile----->Byte code ---------execute--->machine code

.java file-------compile using **javac** tool -->.class file---execute using **java** tool -->.output



Note:

1. Byte code is independent of Operating System.
2. Byte code is platform independent.
3. Java slogan is 'write once run anywhere'(we **write** only **once** and we can **run** the same application **anywhere**)
4. To convert byte code to machine code it is using interpreter i.e JVM (Java Virtual Machine)
5. JVM is platform dependent.

Compilation: converting .java file to .class file

.java file-----compile using javac tool--------.class file

Execution: converting .class file to output

.class file--execute using java tool----------output

5. **Robust(Memory Management):**

Java is a portable programming language, it means we can develop a java application in one operating system and we can execute it on any other operating system without making any changes.
Java is a robust programming language, it means the MM(Memory Management) will be taken care by java itself.
Note: Java contains garbage collector which frees the memory (using system.gc()) once the program execution is completed.
Automatic memory management: Java manages the memory allocation and de-allocation for creating new objects. The program does not have direct access to the memory. The so-called garbage collector automatically deletes objects to which no active pointer exists.

6. **Multithreaded programming:**

Except DOS, every operating system is multitasking. i.e. it is possible to execute more than one process.
Thread: A part of the process which is under execution it is called thread.
In multi-threading (one process with different sub processes) 2 or more sub process (thread) are running simultaneously.
Every application (java program) requires minimum one thread i.e. main thread.

7. **Security(by using JAAS)**

Java is a secured programming language; it means java provides many types of security according to the application.
For a standalone java application, the security is defined as JAAS.
JASS: Java Authorization & Authentication Service

8. **Strongly-typed programming(Case-sensitive programming language):**

The Java syntax is similar to C++. Java is case-sensitive, e.g., variables called `myValue` and `myvalue` are treated as different variables.
Java is a Strongly-typed programming language. Java is strongly-typed, e.g., the types of the used variables must be pre-defined and conversion to other objects is relatively strict, e.g., must be done in most cases by the programmer.

Java is a case-sensitive programming language. It means there is difference between upper case and lower case letters.
In java, an application is written according to java naming rules.
Ex: Class names must start with capital letters only. ex. `Online`
Method names first word small and remaining words are all capitals. ex. onlineCoreJava()

9. **Packages(java.lang, java.util, com.java.example):**

It contains a set of classes.
In set of classes we will have set of methods.
The program who is writing main method is called end user.
We should create an object and call method.

Every package is started with java or javax.

2. interface looks like class with static final variables and abstract methods.
3. `import com.java.onlinesessions.*; //user defined package`
4. import java.lang.*;// system defined package
5. import java.util.concurrent.*; // system defined sub-package

Ex: java.lang.Object ------here Object is class.
Java.util.*;
Java.io.*;
Javax.servlet.Servlet ---here Servlet is an interface.

10. **Native methods:**

A method which is developed using other programming languages like C, C++ and if it is invoked(executed) from java programming, then this method is called native method for java.

11. **Remote methods:**

The present technology is distributed technology; in this we are using remote methods. It is a method which is written in a separate application within a single system or in a networked system and executing it from another system.

12. **Interfaces:**

An interface is like a class; in this we write methods without any method definition.

13. **Java Archive(jar):**

Jar is the facility, is used to compress n no. of files into single file.

Type of archives:

.jar (java archive): it consists of java classes i.e all .class files.

.war (web archive): it consists of html files with java classes. i.e it consists of images, jsp, html files along with .class files(java classes).

.ear (enterprise archive): it consists .class files, images, jsp, html files along with Middle Ware components like ejb components. So we can say ear file is a combination of both jar & war files.

14.

Java Language Keywords:

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try

char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

Structure and rules to write, save, compile and execute any java program:

Structure of Java Program:

Syntax:

```
package  packagename;

import   packagename;

class  ClassName{
                variables;
                methods;

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

Rules:

package statement
-
-
import statement
-
-

class declaration
-
-
-

main() method

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

-

procedure to write ,save, compile and execute java program

step1: to write the java program we need to use any editor like notepad, wordpad, VI Editor, IDE,...

step2: write the java program according to the requirement

```
class   Welcome{

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

```
        System.out.println("My First Java Program");
        }

       }
```

step3: save the program with any name but provide extension name as .java (FirstProgram.java)

step4: (compiling the java program)

- for compiling and executing the java program we have to take the help of command prompt

- to compile the java program we have to use a java command called " javac "

syntax:
javac programname (with extension)
Eg:
javac FirstProgram.java

- when we compile the java program first it will check whether the code written is valid or not if code is valid then it will generate .class file based on class name

- in the above program when we compile, compiler will generate Welcome.class file

step5: (executing the java program)

- to execute the java program we have to use a java command called " java "

syntax:
java classname (without extension)
Eg:
java Welcome
o/p:
My First Java Program

Note:

- before we compile or execute the java program first we have to install a java software.

- java software is called as open source because it is available free of cost and we can freely download from internet

- before we compile or execute the java program first we have to set path to java installation folder like follows

set path="c:\Program Files\Java\jdk1.6.0_12\bin"

```
//write a program to display user address
class  UserAddress{
public  static void main(String  args[]){

System.out.println("bvn");
System.out.println("hno:4-55");
System.out.println("ameerpet");
```

System.out.println("delhi");// we can write anything instead of hyd, bz just we are displaying address so we can write anything between double codes.
}
}

Rules:
---
1. we can write any number of statements in a java program which will execute one by one in sequence order.

2. every statement in java should be terminated by " ; "

3. java is called a case sensitive language it means the lower case letters and upper case letters are different.

4. If we dont write main() method then code is valid and compiled successfully but at runtime it will throw a runtime error or exception saying " **NoSuchMethodError**: main "

TopicName2. OOPs, Packages, Class and Objects:

Object-Oriented Programming (OOP) is a programming paradigm using "objects" – usually instances of a class – consisting of data fields or instance variables and methods together with their interactions – to design applications and computer programs.

Object oriented programming language supports all the features of an object.
Important features of an object:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Why Java is object oriented programming language.

1. The basic aim of an object is it is a reusable software component.
2. To perform anything in java program we need object.
3. To call any instance variable in the class we need object.
4. To call any method, we need object.
5. To call any class1 variables from class2 then we need class1 object.
6. We are using constructor to create an object.
7. Constructor will execute automatically while we are creating an object.
8. If we initialize any object once then by using object we can retrieve initialized values n number of times.
9.

Ex: When we take Address(properties are H No, street, city, pin) as an object then reuse **n** no. of times i.e. we can use the same address object for Employee add, Student add, customer add etc.

Encapsulation:

1. It is a class(Combining the data with its associated functions/methods and making a unit and this unit is called class).
2. We will use encapsulation to hide the data.
3. If a data member is private it means it can only be accessed within the same class.
4. No outside class can access private data member (variable) of other class. However if we setup public getter and setter methods to update (for e.g. void setSSN(int ssn))and read (for e.g. int getSSN()) the private data fields then the outside class can access those private data

fields via public methods. This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That's why encapsulation is known as data hiding.

Benefits of Encapsulation:

1. Combining the data with its associated functions/methods and making a unit and this unit is called class.

Encapsulation means hiding of data

2. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Inheritance:

Inheritance is a mechanism in which a new class will get the properties from already existing class. The old class is called as base class and the newly created class is called derived class. (or)

Inheritance is the process by which one object acquires the properties of another object.

Advantages:

1. Inheritance is the process of inheriting all the features from base class to derived class.
2. The advantages of inheritance are reusability of code and accessibility of variables and methods of the super class by subclasses.
3.

Polymorphism:

It means "one word having different meaning at different instances".

Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Polymorphism is the ability to use an operator or function in different ways.

Single function name can be use to perform different functionality.

Example of polymorphism:

The plus sign "+" operator can be used to add 6 + 10(=16) it refers to integer addition.

The same + operator can be used with different meanings with strings: "Online" + "Training" as OnlineTraining.

The same + operator can also be used for floating point addition: 7.15 + 3.78

Types of Polymorphism:

Polymorphism refers to the ability of an object to behave differently to the same message.

static polymorphism:	dynamic polymorphism:
Method overloading would be an example of static polymorphism.	Method overriding would be an example of dynamic polymorphism,
In static polymorphism it is decided on compile-time.	In dynamic polymorphism the response to message is decided on run-time
If the assignments of data types in compile time it is known as early or static binding.	The assignment of data types in dynamic polymorphism is known as late or dynamic binding.
In static binding method call occur based on the reference type at compile time. Eg: method overloading	In dynamic binding method call occur based on the object (instance) type at Run time. Ex: method overriding

Abstraction:

It shows essential features **and** hiding implementation **part.**

Abstraction in Java allows the user to hide non-essential details relevant to user.

It allows only showing the essential features of the object to the end user.

1. Use abstraction if you know something needs to be in class but implementation of that varies.
2. In Java you **cannot create instance of abstract class** , its compiler error.
3. abstract is a keyword in java.
4. A class automatically becomes abstract class when any of its method declared as abstract.
5. Abstract method doesn't have method body.
6. **Variable cannot be made abstract**, its only behavior or methods which would be abstract.
7. If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively this class can also be abstract.
8.

Package, Class and Object:

It is important to understand the base terminology of Java in terms of *packages*, *classes* and *objects*. This section gives an overview of these terms.

Package

Java groups classes into functional *packages*.

Packages are typically used to group classes into logical units. For example, all graphical views of an application might be placed in the same package called `com.java.bvn.webapplication.views`.

It is common practice to use the reverse domain name of the company as top level package. For example, the company might own the domain, abc.com and in this example the Java packages of this company starts with `com.bvn`.

Other main reason for the usage of packages is to avoid name collisions of classes. A name collision occurs if two programmers give the same *fully qualified name* to a class. The *fully qualified name* of a class in Java consists of the package name followed by a dot (.) and the class name.

Without packages, a programmer may create a Java class called **Test.** Another programmer may create a class with the same name. With the usage of packages you can tell the system which class to call. For example, if the first programmer puts the **Test** class into package **venkata** and the second programmer puts his class into package **narayana** you can distinguish between these classes by using the *fully qualified name of class*.

Ex: narayana.Test or
venkata.Test.

1. A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

2. The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface" or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming.
   For example:
   a `String` object contains state and behaviour for character strings;
   a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the file system;
   a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.
3. The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Built-in packages or predefined packages:

Ex: 1. java.lang.*;
2. java.io.*;
3. java.util.*;
4. java.lang.reflect
5. java.util.concurrent
6. javax.servlet.HttpServlet.

User-defined packages:

Ex. com.bvn.corejava.classexample (or) com.java.bvn.onlinesessions (or) javasessions

Advantage of Java Package:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Class:

In Java source code a class is defined by the *class* keyword and must start with a capital letter. The body of a class is surrounded by {}.

```
package com.java.bvn.example;

class MyClass {

}
```

The data associated with a class is stored in *variables*; the behaviour associated to a class or object is implemented with *methods*.

A class is contained in a Java source file with the same name as the class plus the .java extension.

1) A class is a way to combine variables and methods, making as a single unit. This unit is called class and name of this unit is called class name. (or)

2) A class is a definition or prototype or collection of instance variables and methods. (or)

3) A class describes all the attributes of objects, as well as the methods that implement the behaviour of member objects. It's a comprehensive data type, which represents a blue print of objects. It's a template of object.

Note: No memory allocates for class until object is created because it acts as a template.

Syntax:

```
class Classname
{
    variables/ instance variables/ data members;
    methods(){};
}
```

Sample code for class:

```
class Example {
  public static void main (String args[])
    {
       System.out.println("example program for class");
    }
}
```

In above code

1. **class:** it is a keyword.
   Why above code is the example for class means, the above code looks like a template or combination of methods and instance variables or data members.

   Example:

```
class Bank{
        String name;
  public String balance;
  private String location;

public void deposit()
  {
    }
 public void withdraw()
   {
     }
}
```

2. **Example**: it is an identifier name given to class.
   We know that always class name should start with capital letter.
   If we use small letter instead of capital letter then we won't get any compilation error but when we follow java naming convention we will use class name which is have first letter as capital letter.
3. **public:** It is a Access Modifier which makes the main() to visible to JVM.
   If we are not use public access modifier in main method then we will get error as "main method not public" at runtime there was no compilation error at compile time.
   We know that always JVM searches for public methods.
4. **static:** main method is static means, that method will be executed by JVM installed into any drive of the system.

If any method is static then we must call that method with class name only.
Generally we will call instance methods with objects.
i.e. method can access only by the members of the class.

Note: why main method is static?
Because object is not required to call static method if it were non-static method, JVM create object first then call main() method that will lead the problem of extra memory allocation.

5. **void:** void is a return type of a method.
   JVM looking for void. JVM performs operation but doesn't return anything.
   This method declared as void as JVM won't give or return any value after implementing the main method.
   If we give any int/float/any data type in place of void then we get runtime error.
6. **main():**
   a. In java every program execution starts from main method.
   The arguments written inside the main() method are called command line arguments.

Ex1:

```
class MainMethodExample {

public static void main(String args[]){

        System.out.println(args[0]);// we are printing value of index zero
        System.out.println(args[1]);// we are displaying value of index one
        System.out.println(args[2]);
        System.out.println(args[3]);
        System.out.println(args[4]);
        System.out.println(args[5]);
        System.out.println(args[6]);

  }
}
```

How to pass parameters to main method:

```
We can pass all Data Type values as Command Line Arguments to main()method at
the time of execution of the program.
        eg:10 "hyd" "nani" 10.10f 123456 123.45 B
```

For Compilation: javac MainMethodExample.java
For Execution: java MainMethodExample 10 "hyd" "nani" 10.10f 123456 123.45 B

Output:

```
10
hyd
nani
10.10f
123456
123.45
B
```

Ex2:

```
public class AddNumbers
{
public static void main(String args[])
  {
        System.out.println ("Addition of two numbers!");
        int a = Integer.parseInt(args[0]);// type casting into int type
        int b = Integer.parseInt(args[1]);
        int sum = a + b;
```

```
            System.out.println("Sum: " + sum);
        }
    }
```

b. The command line arguments are always String form bz every input into a java application is taken in the form of String.
c. main() is a procedure in java, it means main() method doesn't return any value so it's return type is void.
d. Always execution starts from main method.
e. We can use more than one main() method bz always JVM looks for main() which starts with p s v m() only and JVM treats all remaining main()'s as ordinary methods.
f. We can overload main method but we can't override main method.
g. **Note:** without main() method we can compile any program but at run time we will get exception. i.e. w/o main method we can't execute any class but we can compile.
h.

7. String: datatype of the array.
8. args[]: name of the array.
9. command line arguments(`String args[]`):command line arguments means passing values from keyboard or command prompt at the time of execution of the program.
10. System: It is a class defined in java.lang package.
11. Out: out is an object of PrintStream class.
12. Println(): It is a method defined in Print Stream class and it is available in java.io package.

Object:

An object is an instance of a class.
The object is the real element which has data and can perform actions.
Each object is created based on the class definition.

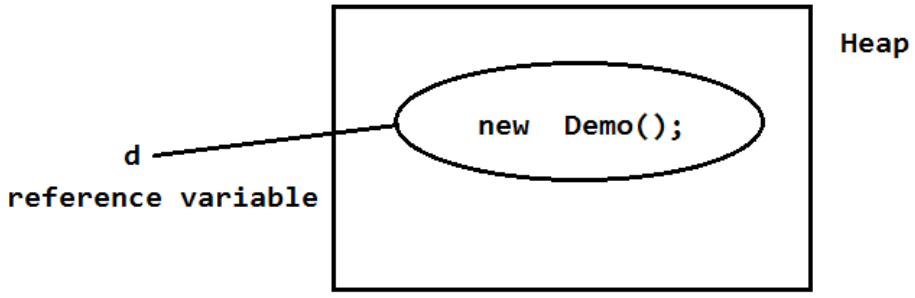
1. An **object** is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behaviour.
   Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation.*
2. An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.
3. Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines the characteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the **new operator**. This process is called instantiating an object or creating an object instance.
4. When you create a new object, you use the new operator to instantiate the object. The new operator returns the location of the object which you assign to a reference type.
5. **Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance (result) of a class.

Imp points:

b) Anything that exists physically in the real world can be considered as an object.
c) Every object will contains some properties and some actions.
d) Properties can be used for describing the object by priviing some data.
e) Actions represent the task performed by an object.

Syntax1: classname objectname = new classname();
Ex: Demo ob = new Demo();
Where **ob** is the object and it contains dynamic memory.
New is the dynamic memory operator.
Syntax2: classname objectname = new classname(parameter);
objectname: It is used to refer created memory for data members/instance variables that's why it uses address of created memory(memory creates in heap).
new: it is a keyword. Which allocates the memory based on data members of the class. And also only for non-static data members of the class.
classname(parameter): constructor. This is used to initialize the data members of the class at the time of allocating memory/creating of an object and assigns the address of this memory location to object. Based on this address location object gets the values of variables or initialized values.

Note: Object reference doesn't contain memory where as
Object contains dynamic memory
Case1: Demo ob = new Demo ();
Case2: Demo ob1; // here ob1 is reference only (in below figure **d** also reference only)
ob1 = new Demo (); //here obj1 is object
case3: Demo ob1;
ob1 = ob;



. (Dot) operator

- This operator used to access the members of the class using reference or column like follows

Eg:

reference.variable;
reference.method();
ClassName.variable; // if variable is static

ClassName.method(); // if method is static

- this operator can also be used to refer or identify the class from a package

Eg:

java.lang.NoSuchMethodError

java.lang.String

We can convert an object reference into object by assigning another object into it.

In case2: ob and ob1 memory locations are different

In case3: ob & ob1 memory locations are same

Ex:
```
class Test {
    int a;
    static String name ;
    float c = 1.22f;

    void method1()
      {}
    Void method2()
     {}
    Test obj = new Test();
}
```

Note:

1. For non-static members(a, c) of the class, allocates memory dynamically (at runtime) using new operator in heap area
2. At the time of compilation itself, it creates memory for static variables (static variable is b).
3. For methods, allocates the memory in stack area.

Topic Name3: Operators and Operands, Java Naming Conventions, Variables:

Java Operators and Operands:

An operation is an action performed on one or more values either to modify the value held by one or both of the values, or to produce a new value by combining existing values. Therefore, an operation is performed using at least one symbol and at least one value.

The symbol used in an operation is called an operator.

A value involved in an operation is called an operand.

A unary operator is an operator that performs its operation on only one operand. An operator is referred to as binary if it operates on two operands.

Curly Brackets { }

Curly brackets are used to create a section of code.

Semi-Colon;

The semi-colon is used to indicate the end of an expression, a declaration, or a statement.

Parentheses ( )

Like most computer languages, Java uses parentheses to isolate a group of items that must be considered as belonging to one entity. For example, parentheses are used to differentiate a method such as **main** from a regular variable. Here is an example:

```
main()
```

Square Brackets [ ]

Square brackets are mostly used to control the dimension or index of an array. We will learn how to use them when we study arrays.

The Comma ,

The comma is used to separate variables used in a group.

The Assignment =

When you declare a variable, a memory space is reserved for it. To "put" a value in the memory space allocated to a variable, you can use the assignment operator represented as =. Based on this, the assignment operation gives a value to a variable.

```
VariableName = Value;
```

Single-Quote ‘’

The single quote is used to include one character to initialize, or assign a symbol to, a variable declared as **char**.

Double Quotes””

The double quote " is used to delimit a string. As mentioned for the single-quote, the double-quote is usually combined with another double quote. Between the combinations of double-quotes, you can include an empty space, a character, a word, or a group of words, making it a string.

The Positive Operator +

Algebra uses a type of ruler to classify numbers. This ruler has a middle position of zero. The numbers on the left side of the 0 are referred to as negative while the numbers on the right side of the rulers are considered positive:

A value on the right side of 0 is considered positive. To express that a number is positive, you can write a + sign on its left. Examples are +4, +228, +90335. In this case the + symbol is called a unary operator because it acts on only one operand. The positive unary operator, when used, must be positioned on the left side of its operand, never on the right side.

As a mathematical convention, when a value is positive, you don't need to express it with the + operator. Just writing the number without any symbol signifies that the number is positive. Therefore, the numbers +4, +228, and +90335 can be, and are better, expressed as 4, 228, 90335. Because the value doesn't display a sign, it is referred as **unsigned**.

To express a variable as positive or unsigned, you can just type it.

The Negative Operator –

As you can see on the above ruler, in order to express any number on the left side of 0, it must be appended with a sign, namely the - symbol. Examples are -12, -448, -32706. A value accompanied by

- is referred to as negative. The - sign must be typed on the left side of the number it is used to negate. Remember that if a number does not have a sign, it is considered positive. Therefore, whenever a number is negative, it MUST have a - sign. In the same way, if you want to change a value from positive to negative, you can just add a - sign to its left.

The Addition Operations:

The addition is an operation used to add things of the same nature one to another, as many as necessary. Sometimes, the items are added one group to another. The concept is still the same, except that this last example is faster. The addition is performed in mathematics using the + sign. The same sign is used in Java.

To get the addition of two values, you add the first one to the other. After the addition of two values has been performed, you get a new value. This means that if you add Value1 to Value2, you would write Value1 + Value2. The result is another value we could call Value3.

Incrementing a Variable:

Instead of writing Value = Value + 1, you can write Value++ and you would get the same result.

The ++ is a unary operator because it operates on only one variable.

It is used to modify the value of the variable by adding 1 to it. Every time the Value++ is executed, the compiler takes the previous value of the variable, adds 1 to it, and the variable holds the incremented value:

Pre and Post-Increment:

When using the ++ operator, the position of the operator with regard to the variable it is modifying can be significant. **To increment the value of the variable before re-using it, you should position the operator on the left of the variable:** When writing ++Value, the value of the variable is incremented before being called.

On the other hand, if you want to first use a variable, then increment it, in other words, **if you want to increment the variable after calling it, position the increment operator on the right side of the variable:**

The Multiplication Operations:

The multiplication allows adding one value to itself a certain number of times, set by a second value. As an example, instead of adding a value to itself in this manner: A + A + A + A, since the variable a is repeated over and over again, you could simply find out how many times A is added to itself, then multiply a by that number which, is this case, is 4. This would mean adding a to itself 4 times, and you would get the same result.

Just like the addition, the multiplication is associative: a * b * c = c * b * a. When it comes to programming syntax, the rules we learned with the addition operation also apply to the multiplication.

The Subtraction Operations:

The subtraction operation is used to take out or subtract a value from another value. It is essentially the opposite of the addition. The subtraction is performed with the - sign.

Note: Unlike the addition, the subtraction is not associative. In other words, a - b - c is not the same as c - b - a.

Decrementing a Variable:
When counting numbers backward, such as 8, 7, 6, 5, etc, we are in fact subtracting 1 from a value in order to get the lesser value. This operation is referred to as decrementing a value. This operation works as if a value is decremented by 1, as in Value = Value – 1:

As done to increment we have a quicker way of subtracting 1 from a value. This is done using the decrement operator, that is --. To use the decrement operator, type –- on the left or the right side of the variable when this operation is desired.

Pre Decrementing a Value:
Once again, the position of the operator can be important. If you want to decrement the variable before calling it, position the decrement operator on the left side of the operand.

If you plan to decrement a variable only after it has been accessed, position the operator on the right side of the variable.

The Division Operations:

Dividing an item means cutting it in pieces or fractions of a set value. For example, when you cut an apple in the middle, you are dividing it in 2 pieces. If you cut each one of the resulting pieces, you will get 4 pieces or fractions. This is considered that you have divided the apple in 4 parts. Therefore, the division is used to get the fraction of one number in terms of another. The division is performed with the forward slash /.

When performing the division, be aware of its many rules. Never divide by zero (0). Make sure that you know the relationship(s) between the numbers involved in the operation.

The Remainder:
The division program above will give you a result of a number with double values if you type an odd number (like 147), which is fine in some circumstances. Sometimes you will want to get the value remaining after a division renders a natural result. Imagine you have 26 kids at a football (soccer) stadium and they are about to start. You know that you need 11 kids for each team to start. If the game starts with the right amount of players, how many will seat and wait?

The remainder operation is performed with the percent sign (%) which is gotten from pressing Shift + 5.

Java naming conventions:

1. Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc. But, it is not forced to follow. So, it is known as convention not rule.
2. All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

1. By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers.

2. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Variables:

1. A variable is the basic storage unit for the Java program.
2. It is a name given to a temporary memory.
3. The Java programming language uses both "fields" and "variables" as part of its terminology.
4. When naming your fields or variables, there are rules and conventions that you should (or must) follow.
5. Variables must be declared before using them.
6. A variable is defined as the combination of type, identifier and an optional initializer.
7. A variable provides us with named storage that our programs can manipulate.
8. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
9. Variables in a class are called instance variables.
10. Instance variables and instance methods are members of the class.

Syntax: access specifier datatype variable/instance variable/data member;
access specifier datatype variable/instance variable/data member =value;

The basic form of a variable declaration is shown here:

```
Access Modifier Data type variable; // variable declaration
Access Modifier Data type variable = value; //variable initialization
```

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;          // Declares three int type variables a, b, and c.
int a = 10, b = 10;   // Example of initialization
byte B = 22;          // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';         // the char variable a is initialized with value 'a'
```

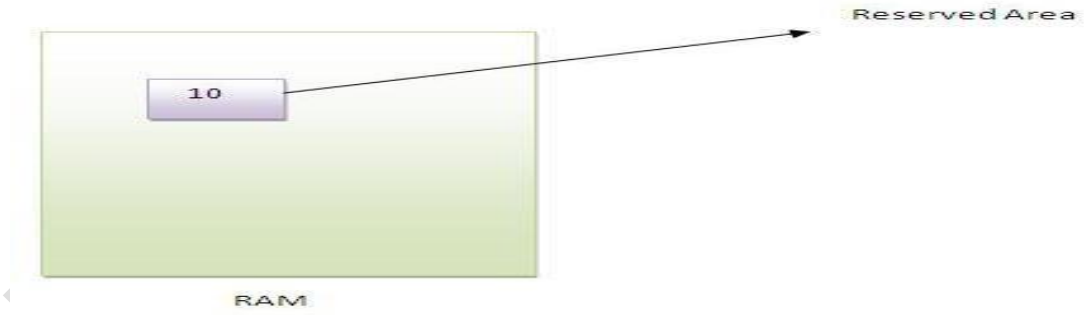
Variable and Data Type in Java:

1. Variable
2. Types of Variable
3. Data Types in Java

Variable is a name of memory location.
There are three types of variables: local, instance and static.
There are two types of data types in java, primitive and non-primitive.

Variable

Variable is name of reserved area allocated in memory.

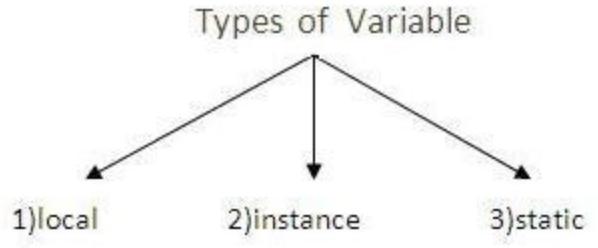



```
int data=50;//Here data is variable
```

Types of Variable:

There are three types of variables in java

- local variable
- instance variable
- static variable



Local Variable:

A variable that is declared inside the method is called local variable.

Instance Variable:

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

Static variable:

A variable that is declared as static is called static variable. It cannot be local.
We will have detailed learning of these variables in next chapters.

Example to understand the types of variables:

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
public static void main(String args[]){
}
}//end of class
```

More info about Type of Variables:

A class can contain any of the following variable types.

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.
- Variables defined inside methods, constructors or blocks are called local variables.

- The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields").
-

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values.
    - For numbers the default value is 0,
    - For Booleans it is false and
    - For object references it is null.
    - Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name.

ObjectReference.VariableName

a) Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded.
b) Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
c) Instance variables (non-static fields) are unique to each instance of a class.

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.

- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name. *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.
- Class variables are variables declared with in a class, outside any method, with the static keyword.
- Class variables (static fields) are fields declared with the static modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated.

Constants:

What is a Constant in Java?
A constant in Java is used to map an exact and unchanging value to a variable name.
Constants are used in programming to make code a bit more robust and human readable.

Syntax:

```
public static final String EMPLOYEE_NAME = "BV Narayana";//EMPLOYEE_NAME is CONSTANT VARIABLE
```

Need of constant: Refer program `WithOutConstant.java`

Look how many times we use the `double` value `3.14159`, this value represents pi (π). We should create a constant that will assign the value of π to a variable name.

We have a variable named `PI` declared in the instance variable declaration space. We've done this because we need to use this new constant value throughout our WithOutConstant class. This constant will function just like any other instance variable with one main exception… we've made the value `final`.

The *public* Access Modifier:

The fact that we declared it **public** allows us to access it from other classes (Static1Test.java in this case i.e. means we can access name variable of Static1Example.java from Static1Test.java). Although

actually in this example the **public** keyword is superfluous as it happens, due to us not having mentioned any package names.

The *static* keyword:

From above syntax we can happily access the '*EMPLOYEE_NAME'* instance variable in the WithEmployeeConstant1 class without actually creating an object of type WithEmployeeConstant1. We can just use the WithEmployeeConstant1 class directly. That's because the variable is **static**, and hence belongs to the class, not any particular object of that class.

Using the static keyword to create constants:

One common use of **static** is to create a constant value that's attached to a class. The only change we need to make to the above example is to add the keyword **final** in there, to make '*EMPLOYEE_NAME*' a constant (in other words, to prevent it ever being changed).

We should also make '*EMPLOYEE_NAME*' uppercase by convention, to indicate that it's a constant. This isn't strictly necessary; it's literally just a convention.

The *final* Keyword:

1) When you declare a variable to be `final` we are telling Java that we will NOT allow the variable's "pointer" to the value to be changed.
2) The `final` keyword means is that once the value has been assigned, it cannot be re-assigned. So if you tried to put in some code later that tries to do this:

```
PI = 3.14159265359
```

   You would get a compilation error, and your IDE would tell you that a new value cannot be assigned because the variable has been declared `final`.
3) A constant is a constant for a reason, it shouldn't ever change! Now, here's a quick note on conventions in Java. When you name a constant, you should use UPPER_CASE_LETTERS_WITH_UNDERSCORES_INDICATING_SPACES. Again, it's not mandatory to use upper case letters with underscores to indicate spaces, but it's a convention that programmers use and are familiar with in Java.

Topic Name4: Data Types (primitive & non-primitive), Data type conversions.

Data Types in Java:

1. A **data type** informs the compiler what type of data a variable is permitted to store.

2. Data type declaration should be done when the variable is declared. It is required in Java as Java is a strongly typed language.

3. The data type indicates what possible values that can be assigned to variable and possible operations that can be performed. The language system allocates **memory** to the variable on the type of data it can hold like an integer or double etc.

4. String in Java is not a data type. String is a class from java.lang package. String is not a keyword; for that matter no class is a keyword. Strings are used very often in coding like data types.
5. All data types are keywords. As a rule, all keywords should be written in lowercase. Java does not support unsigned data types; every data type is implicitly signed (except char). "**unsigned**" is not a keyword of Java. Note that **null**, **true** and **false** are also not keywords.
6. **Primitive Data types** form the back bone to represent the properties/attributes of the **objects**. Any form of the data can be stored by using the primitive data types in Java.
7. **Data types in Java** are actually derived from the programming language C. This is the reason why, the **primitive data types** representation in **Java** looks like the structured programming. This is also one of the reasons why some people quote that Java is only 99% **Object oriented programming** and the rest one percent – they are mentioning about the primitive data types representation.

In java, there are two types of data types:

1. **primitive data types** -------byte, short, int, long, float, double, char, Boolean.

   Java supports 8 data types that can be divided into 4 broad categories.

   integer type: byte(1byte), short(2bytes), int(4bytes), long(8bytes)

   float type ---------float(4bytes), double(8bytes)

   char type -----char(2bytes)

   boolean -------false or true

Primitive Data Types:

There are totally eight primitive data types in Java. They can be categorized as given below:

- Integer types (Does not allow decimal places)
  - **byte**
  - **short**
  - **int**
  - **long**
- Rational Numbers(Numbers with decimal places)
  - **float**
  - **double**
- characters
  - **char**
- conditional
  - **boolean**

Please notice that all the data type keywords are in small letters. These are part of the java keywords and every keyword in java is in small letters.

In the **integer data types**, we have four different data types. But, why do we need four different types when one can do the job. Yes, it is extremely important to understand that each and every data type has limitations to the amount of numbers it can represent. There is a memory constraint defined for every data type.

Understanding the memory limitations is extremely important in deciding which data type should be used. For example, when you are representing the age of a person, for sure it will not cross 120, so, using short data type is enough instead of long which has very big memory foot print.

The following should be understood for every data type:

1. Memory size allocated.
2. Default value
3. Range of values it can represent.

Memory size for data types:

Every data type has some memory size defined. This enables that whenever a variable is declared, the memory size defined will be blocked irrespective of the value that is going to get assigned.

Default value:

Every primitive data type has default values defined. When the programmer does not declare to assign any values to the variables, these default values will be assigned by the Virtual Machine during the object instantiation.

Range of values the data types can represent

It is extremely important to understand what are the min and max range of values a data type can be able to hold. Without this, whenever a value that cannot fit the data type is assigned, the applications crash with fatal errors or exceptions.

Please note that all the data types that can represent numbers can hold both positive and negative numbers.

2. **non-primitive data types** - which include Classes(eg: String), Interfaces, and Arrays.
3.

Reference Data Types or Reference/Object Data Types or non-primitive data types are objects.

Reference Data Types:

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Student etc.

Class objects and various types of array variables come under reference data type.

Default value of any reference variable is null.

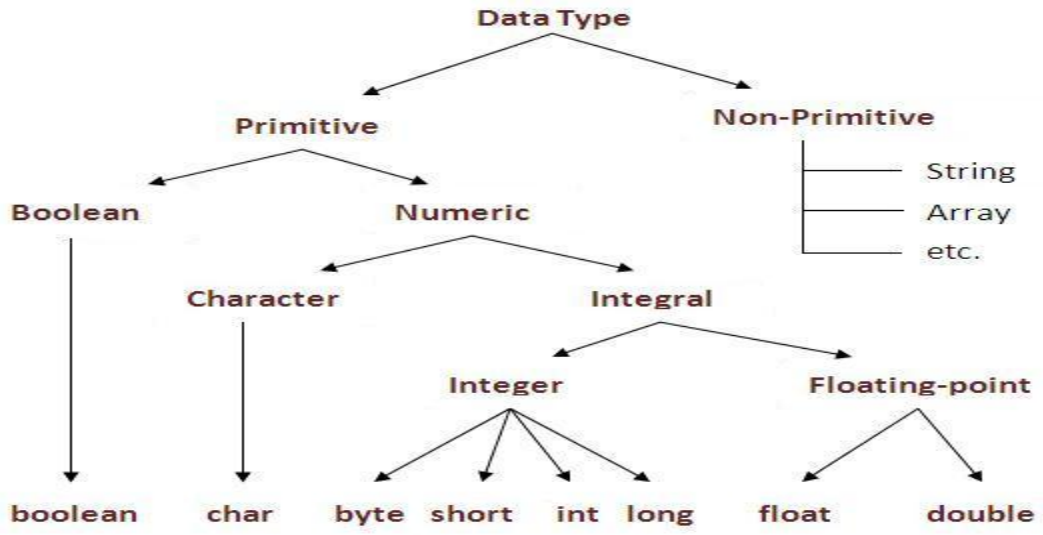
A reference variable can be used to refer to any object of the declared type or any compatible type.

Example: Animal animal = new Animal("giraffe");

The above like looks like

```
Animal animal; // reference object for Animal
Animal animal = new Animal("giraffe"); // object for Animal class
```

For more information please refer below diagram:



The following chart summarizes the default values for the above data types.

Data type	Memory it takes	Default value	Range of values accepted
boolean	0 byte (1 bit)	False	True or False
byte	1 byte (8bits)	0	– 128 to +127 (2^7 to 2^7-1)
char	2 bytes (16 bits)	'\u0000'	0 to 65,535 (0 to 2^8-1)
short	2 bytes (16 bits)	0	-32,768 to + 32,767 (2^{15} to 2^{15}-1)
int	4 bytes (32 bits)	0	-2,147,483,648 to +2,147,483,647 (2^{31} to 2^{31}-1)
float	4 bytes(32 bits)	0.0	single-precision. 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative)
double	8 bytes(64 bits)	0.0	double-precision. 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)
long	8 bytes(64 bits)	0	-9.223372036854776E18 to +9.223372036854775E18 (2^{63} to 2^{63}-1)
String		null	

Data Type conversions:

1. **Implicit conversion(type conversion or type casting)**

   Condition: For memory locations are computable.

2. **Explicit conversion(type casting):**

   Condition: For memory locations are incomputable. i.e. source memory greater than destination memory in an expression.

Example for implicit conversion:

Ex: double x;
int y =25;

x = y; // we can assign int value into double bz double size 8 bytes where as int size 4 bytes.

Ex: 2/3 = 0.6(treats as zero) = 0 ------there is no memory incomputable

Example for explicit conversion:

1) byte x;
   int a = 50;

   x = a; //error, due to memory difference. Bz we can't store 4 bytes into 1 byte

   x = (byte) a; // now int converted to byte.

   destination(x) = source(a)

---

2) byte a, b;
   a = 10;
   b = 20;
   a = (byte)a+b

Topic Name5: Methods (main & command line arguments, Getters & Setters, static & non- static methods, call-by-value & call-by-reference):

5.1 Methods:

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

1. For methods, allocates the memory in stack area.

Syntax: modifier returnValueType methodName(list of parameters)
{
// Method body;
}

Ex: public void display (){
System.out.println ("Java");
}

Modifier: The modifier, which is optional, tells the compiler how to call the method.
This defines the access type of the method.

Return Type: A method may return a value.
The returnValueType is the data type of the value the method returns.
Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void**.

Method Name: This is the actual name of the method.
The method name and the parameter list together constitute the method signature.

Parameters: A parameter is like a placeholder.
When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

Method Body: The method body contains a collection of statements that define what the method does.

Note: without main method we can compile any program but at run time we will get exception. i.e. w/o main method we can't execute any class but we can compile.

How to Write and Use a Method in Java:

Sometimes the terms function and method are used interchangeably. They are virtually the same. The correct terminology for Java is method. It is a set of commands that can be used over again. In this, they share similarities with sub routines in the early days of programming.

1. A simple Java method requires a minimum of three items:

Visibility: public, private, protected
Return Type: void, int, double, (etc.)
name: whatever you want to call the method

Visibility means who can access it. If it is public, anyone who has access to your class file can access the method. In some circumstances, this is perfectly fine. If the method is private, only the functions inside of that class can call the method. This is used as utility methods to do something you do not want just anyone who uses the class to do. Protected gives public function to all child classes.

Return type is void if you do not want the method to give you any data back. It would be used for such things as a method that prints out something. Any other return requires a return statement with the type of data it returns. For example, if you add two integers and want the results of that integer, your return type would be int.

Name of the method is anything you choose that is not already used in the class (unless you are overloading the method which is beyond the scope of this article)

If you want the method to do something with the data you supply it, you also need to include **parameters** within the ( ) You include what data type it is and give that parameter a name (ie: you are declaring a local variable for that method only)

Method Calling:

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- return statement is executed.
- reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Let's consider an example:

```
System.out.println("This is the content to display console as output!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

The void Keyword:

The void keyword allows us to create methods which do not return a value.

Passing Parameters by Value:

While working under calling process, arguments are to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter.

5.2 main method:

Command-Line arguments: The values which we are passing to main method at execution of the program or runtime of the program.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

The command line arguments of a Java program:

Arguments (parameters) of the `main` method

- The **data type** of the ***parameter*** **variable** `args` of the `main` **method** is an ***array*** **of** `String`
- `args[0]` is the **first element** of this array.
  The **data type** of `a[0]` is `String`
- `args[1]` is the **second element** of this array.
  The **data type** of `a[1]` is `String`
- And so on.
- `args.length()` is the **length** of the array.

Passing *command arguments* to a Java program

When we **execute** a **Java program** `ProgramName` using the following **syntax**:

```
java  ProgramName  word0  word1  ...
```

the **Java system** will ***pass*** the **words**:

- `word0` to the ***first*** `String` **parameter variable**
- `word1` to the ***second*** `String` **parameter variable**
- And so on.

Example: how to parse command line arguments:

parsing is a process of converting a string into required primitive type.

We know that when we use command line arguments, we must pass the values at the time of running the program.
We know that java treats, the values entered through the command prompt treats as strings.

Example:
```
class Example{
    Public static void main(String args[]){
        String name =args[0];
        String fname = args[1];
        int a = args[1]; //
    int a = Integer.parseInt(args[1])
// why we are parsing/type casting into integer means, the values passed through command line arguments treats as string and here type of a is interger. So we need to parse the string values into integer format.
// Integer class available in java.lang packege
        System.out.println(a);
        System.out.println(args.lenth);// here length is property not method.
// args.lenth is used to know length of an array.
    }
}
```

5.3 Getter & Setter Methods:

Define getter and setter methods

- Define methods which allow you to read the values of the instance variables and to set them. These methods are called *setter* and *getter*.
- Getters should start with `get` followed by the variable name whereby the first letter of the variable is capitalized.
- Setter should start with `set` followed by the variable name whereby the first letter of the variable is capitalized.

For example, the variable called *firstName* would have the `getFirstName()` getter method and the `setFirstName(String s)` setter method.

Change your `main` method so that you create one `Person` object and use the setter method to change the last name.

A getter is simply a method on a class to retrieve the value of a private member of the class (and a setter is used to set its value).

5.4 static method:

If you 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 can be 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.

Note:

1. The static method can't use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

Note: why main method is static?

Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

5.5 Call-by-Value and Call-by-Reference in Java:

pass-by-value: pass-by-value means we will pass primitive value to methods and these values will effect in locally.

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

In case of call by value original value is not changed.

pass-by-reference: we are passing obj as parameter to method. i.e. changes will be in the instance variable

In case of call by reference original value is changed if we made changes in the called method.

If we pass object in place of any primitive value, original value will be changed.

In this example we are passing object as a value.

5.6 method overloading and method overriding:

Note: I will cover 'method overloading' and 'method overriding' in polymorphism

Method Overloading:

In same class, if name of the method remains common but the number and type of parameters are different, then it is called method overloading in Java.

1. appear in the same class or a subclass
2. have the **same name** but,
3. have different **parameter lists**, and
4. can have different **return types**

Note: When a class has two or more methods by same name but different parameters, it is known as method overloading. It is different from overriding. In overriding a method has same method name, type, number of parameters etc.
the concept of Overloading will be introduced to create two or more methods with the same name but different parameters.

Method overriding:

When a class defines a method using the same name,
return type, and
arguments as a method in its superclass, the method in the class overrides the method in the super class.
When the method is invoked for an object of the class, it is the new definition of the method that is called, and not the method definition from superclass.
Methods may be overridden to be more public, not more private.

method overloading	method overriding
overloading is the same method name but different arguments	Overriding is a method with the same name and arguments as in a parent
In overloading, there is a relationship between methods available in the same class	In overriding, there is relationship between a superclass method and subclass method.
Overloading does not block inheritance from the	Overriding blocks inheritance from the superclass.

superclass	
In overloading, separate methods share the same name	In overriding, subclass method replaces the superclass.
Overloading must have different method signatures	Overriding must have same signature

Topic Name6: Inheritance

Inheritance in java is a mechanism in which one object(child class object) acquires all the properties and behaviours of parent object. i.e. Acquiring the properties and behaviours of parent class into child class.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as ***parent-child*** relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

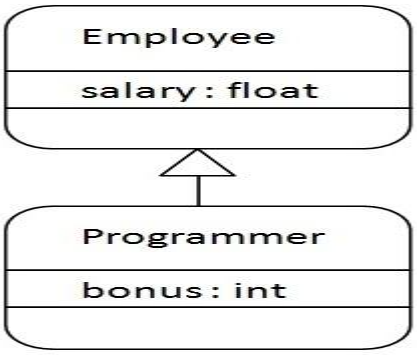
Syntax of Java Inheritance:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

Types of inheritance in java:

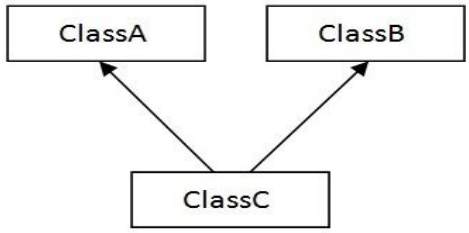
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.

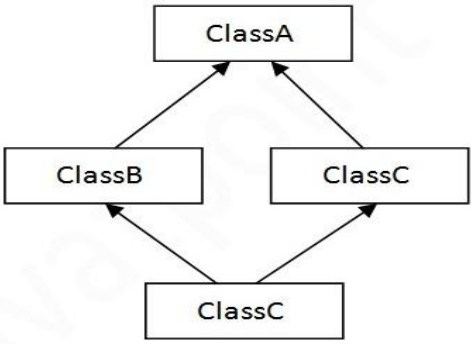
Note: Multiple inheritance is not supported in java through classes.

Multiple inheritance is supported in java through interfaces.

When a class extends multiple classes i.e. known as multiple inheritance.



4) Multiple



5) Hybrid

.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes.

IS-A and HAS-A relationship:

Inheritance is another important feature of object oriented programming, it provide the feature of code reusability, where one class inherit the features of another class, so code redundancy is reduce.

Inheritance support two type of relationship

- IS-A relationship - used when you want to use all the features(variable and methods) of the base class.
- Has-A relationship - used when you want to use only some features of the base class.

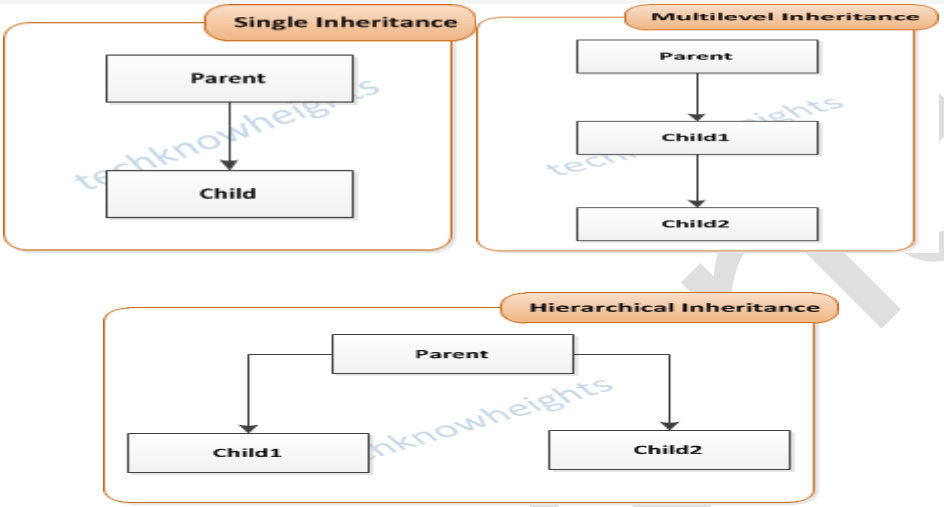
IS_A relationship is further divided

- Single level Inheritance

- Multilevel Inheritance
- Hierarchical Inheritance

NOTE :

Java does not support Multiple Inheritance. Multiple Inheritance can be achieved in java through the implementation of Interfaces.



Difference between IS-A and HAS-A relationship

IS-A	**HAS-A**
Used to create a parent child relationship. IS-A is based on class inheritance or interface.	HAS-A relationship is based on usage, rather than inheritance.
Example: BMW is a car, horse is an Animal, Auto-Loan IS-A.	**Example:** Horse HAS-A Halter, Customer HAS-A Loan.
IS-A can be used by extends or implements keyword.	HAS-A used by making an object of the other class in your class.
Advantage of IS-A • To promote code reusability, so code redundancy is reduced. • To use polymorphism.	Advantage of HAS-A • Can work on other class instance variables & methods without inheriting that class.

NOTE :

- we cannot make a single program without inheritance in java, because every class inherit**java.lang.Object** class.
- Java is a package centric language; the developer assumed that for good organization and name scoping, you would put all your classes into packages.

Combination of IS-A and HAS-A Relation:

IS-A relation means Inheritance,

HAS-A relation means Composition.

Ex:

```
class Engine{
...
..
}

Class Vehicle{
Engine obj; // example of composition
        obj = new Engine();
..... // HAS-A relation (Vehicle HAS-A Engine )
....
..
}

class Car extends Vehicle { //example of Inheritance
.... // IS-A Relation (Car IS-A Vehicle)
...
..
}
```

Aggregation in Java:

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Why we need Aggregation?

- **To maintain code re-usability**.

Summary:

- IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
- Has-a relationship is composition relationship which is productive way of code reuse.
- **Why we use inheritance in java:**
  - For Method Overriding (so runtime polymorphism can be achieved).
  - For Code Reusability.
- We will get Is-A relation using extends keyword and
- We will get Has-A relation using new keyword.

Topic Name7: Polymorphism

Polymorphism:

1) Polymorphism, which is one of the features of Object Oriented Programming (OOPs).
2) Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words,
3) Polymorphism allows you define one interface and have multiple implementations.

4) Polymorphism is one of the most important feature of Object Oriented Programming. Polymorphism means one thing having several forms.
5) It is a feature that allows one interface to be used for a general class of actions.
6) An operation may exhibit different behaviour in different instances.
7) The behaviour depends on the types of data used in the operation.
8) It plays an important role in allowing objects having different internal structures to share the same external interface.
9) Polymorphism is extensively used in implementing inheritance.

Following concepts demonstrate different types of polymorphism in java.

1) Method Overloading

2) Method Overriding

Method Definition:

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.

1) Method Overloading:

In Java, it is possible to define two or more methods of same name in a class, provided that their argument list or parameters are different. This concept is known as Method Overloading. (or)
Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different.

Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters
3. Sequence of Data type of parameters.
4. **Method return type doesn't matter in case of overloading.**

Method overloading is also known as **Static Polymorphism**.

Points to Note:

1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Rules for Method Overloading:

1. Overloading can take place in the same or in its sub-class.
2. Constructor in Java can be overloaded
3. Overloaded methods must have a different argument list.
4. Overloaded method should always be in part of the same class, with same name but different parameters.

5. The parameters may differ in their type or number, or in both.
6. They may have the same or different return types.
7. It is also known as compile time polymorphism.
8. You can overload a method by changing its signature. Method signature is made of number of arguments, type of arguments and order of arguments.
9. Return type is not a part of method signature, so changing method return type means no overloading unless you change argument with return type.
10. A method can be overloaded in the same class or in a subclass. if class A defines a show(int i) method, the subclass B could define a show(String s) method without overriding the superclass version that takes an int. So two methods with the same name but in different classes can still be considered overloaded, if the subclass inherits one version of the method and then declares another overloaded version in its class definition.
11. To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call.
12. Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method.
13. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
14. It allows the user to achieve compile time polymorphism.
15. An overloaded method can throw different exceptions.
16. It can have different access modifiers.
17. Method Overloading is also called compile time polymorphism(static binding).
18. Overloaded methods can be reused as the same method name in a class, but with different arguments and optionally, a different return type.

2) Method Overriding

Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class. This feature is known as method overriding.

Method Overriding is achieved when a subclass overrides non-static methods defined in the superclass, following which the new method implementation in the subclass that is executed.

The new method definition must have the same method signature (i.e., method name and parameters) and return type. Only parameter types and return type are chosen as criteria for matching method signature. So if a subclass has its method parameters as final it doesn't really matter for method overriding scenarios as it still holds true. The new method definition cannot narrow the accessibility of the method, but it can widen it. The new method definition can only specify all or none, or a subset of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the super class.

Rules for Method Overriding:

1. method must have same name as in the parent class.
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).
4. applies only to inherited methods
5. object type (NOT reference variable type) determines which overridden method will be used at runtime
6. Overriding methods must have the same return type
7. Overriding method must not have more restrictive access modifier
8. Abstract methods must be overridden
9. static and final methods cannot be overridden
10. It is also known as Runtime polymorphism.
11.
12. Argument list: The argument list of overriding method must be same as that of the method in parent class. The data types of the arguments and their sequence should be maintained as it is in the overriding method. i.e The argument list must exactly same that of the overridden method. If they don't match, you can end up with an overloaded method.
13. The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass (also called covariant return type).
14. Access Modifier: The Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of base class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier as all of the three are more restrictive than public.

    For e.g. This is not allowed as child class disp method is more restrictive(protected) than base class(public).
15. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
16. Instance methods can be overridden only if they are inherited by the subclass.

17. A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass).
18. The overriding method(method of child class) can throw any unchecked (runtime) exception, regardless of whether the overridden method(method of parent class) declares the exception.
19. The overriding method must not throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.
20. Binding of overridden methods happen at runtime which is known as dynamic binding.
21. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is an abstract class.
22. You cannot override a method marked final.
23. You cannot override a method marked static.
24. If a class is not inherited, you cannot override its methods.
25. Method Overriding is also called Runtime polymorphism (dynamic binding)
26. The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.
27. In object-oriented terms, overriding means to override the functionality of an existing method.
28. An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.
29. The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method.
30. *Method Overriding is possible in the case of inheritance where child class inherits the property of parent class, i.e; if parent class has some method then child class can override the method but keep in mind that signature of the method must be same as on parent class. At runtime JVM will decide which is to call depending upon its reference.*
31. The access level cannot be more restrictive than the overridden method's access level. For example: if the super-class method is declared public then the overriding method in the sub class cannot be either private or protected.
32. A method declared final cannot be overridden.
33. A method declared static cannot be overridden but can be re-declared.
34. If a method cannot be inherited, then it cannot be overridden.
35. A subclass within the same package as the instance's super-class can override any super-class method that is not declared private or final.

36. A subclass in a different package can only override the non-final methods declared public or protected.
37. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
38. Constructors cannot be overridden.

Super keyword in Overriding

super keyword is used for calling the parent class method/constructor.`super.methodname()` calling the specified method of base class while `super()` calls the constructor of base class. Let's see the use of super in Overriding.

1. Can we override static method?
No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

2. Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

3. Can we override java main method?

No, because main is a static method.

Types of polymorphism in java- Runtime and Compile time polymorphism

There are two types of polymorphism in java- **Runtime polymorphism (Dynamic polymorphism)** and **Compile time polymorphism (static polymorphism)**.

Compile time Polymorhism (or Static polymorphism):

Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both.

Runtime Polymorhism(or Dynamic polymorphism):

Method overriding is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of class X can hold object of class X or an object of any sub classes of class X. For e.g. if class Y extends class X then both of the following statements are valid:

```
Y obj = new Y();
//Parent class reference can be assigned to child object
X obj = new Y();
```

Since in method overriding both the classes(base class and child class) have same method, compile doesn't figure out which method to call at compile-time. In this case JVM(java virtual machine) decides which method to call at runtime that's why it is known as runtime or dynamic polymorphism.

Topic Name8: Constructors and types of Constructors:

Constructor in Java is block of code which is executed at the time of Object creation. But other than getting called, Constructor is entirely different than methods and has some specific properties like name of constructor must be same as name of Class. Constructor also can not have any return type.

1. Origin of constructor: Instead of writing separately both object creation and object initialization, we can write both are in same line.
   Purpose and function of constructor:
   Constructors have one purpose in life: to create an instance of a class. This can also be called creating an object.
   Example obj = new Example();
2. It has the same name, as the class in which it resides and looks from syntax point of view it looks similar to a method.
3. Constructor is the method which name is same to the class.
4. Constructor is automatically called immediately after the object is created, before the new operator completes.
5. Constructor will be automatically invoked when an object is created where as method has to be called explicitly.
6. **Constructors have no return type, not even void.** This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object.
7. Every class has at least one its own constructor. If we have not defined any constructor then JVM itself creates default constructor.
8. Constructor creates an instance for the class.
9. Constructor initiates something related to the class's methods.
10. Constructors are required to create objects for a class.

11. Constructors are used to initialize the instance variables of an object.
12. Constructor declaration looks like method declaration. It must have the same name as that of the class and have no return type.
13. If you don't define a constructor, then the compiler creates a default constructor. Default constructors do not contain any parameters. Default constructors are created only if there are no constructors defined by us.
14. Constructors can be classified into four types.

    Dummy constructor
    Default/non-parameterized constructor
    Parameterized constructor
    Copy constructor
15. Parameterized constructors are required to pass parameters on creation of objects. We can overload constructors with different data types as its parameters.
16. Use 'this()' to communicate from one constructor to another constructor in the same class.
17. Use 'super()' to communicate with super class constructor.

Signature differences with method:

Constructors and methods differ in three aspects of the signature:

1) modifiers
2) return type and
3) name.

1. Like methods, constructors can have any of the access modifiers: public, protected, private, or none(default access modifier) (often called *package* level).
2. Unlike methods, constructors can take only access modifiers. Therefore, constructors cannot be abstract, final, native, static, or synchronized.
3. The return types are very different too. Methods can have any valid return type, or no return type, in which case the return type is given as void. Constructors have no return type, not even void.
4. Finally, in terms of the signature, methods and constructors have different names. Constructors have the same name as their class; by convention, methods use names other than the class name. If the Java program follows normal conventions, methods will start with a lowercase letter, constructors with an uppercase letter. Also, constructor names are usually nouns because class names are usually nouns; method names usually indicate actions.

Summarizing the differences between constructors & methods:

- Specifically in their purpose, signature, and use of `this` and `super`. Additionally, constructors differ with respect to inheritance and compiler-supplied code. Keeping all these details straight can be a chore; the following table provides a convenient summary of the salient points.

Topic	Constructors	Methods

Purpose	Create an instance of a class	Group Java statements
Modifiers	Cannot be abstract, final, native, static, or synchronized	Can be abstract, final, native, static, or synchronized
Return type	No return type, not even void	void or a valid return type
Name	Same name as the class (first letter is capitalized by convention) -- usually a noun	Any name except the class. Method names begin with a lowercase letter by convention -- usually the name of an action
This	Refers to another constructor in the same class. If used, it must be the first line of the constructor	Refers to an instance of the owning class. Cannot be used by static methods
Super	Calls the constructor of the parent class. If used, must be the first line of the constructor	Calls an overridden method in the parent class
Inheritance	Constructors are not inherited	Methods are inherited
Compiler automatically supplies a default constructor	If the class has no constructor, a no-argument constructor is automatically supplied	Does not apply
Compiler automatically supplies a default call to the superclass constructor	If the constructor makes no zero-or-more argument calls to super, a no-argument call to super is made	Does not apply

Types of constructors:

1. dummy constructor(creates by JVM and we can see this constructor in .class file by using JAD(JavaDecompiler))
2. Default/ non-parameterized /constructor without parameters (we use this to create object without parameters.)
3. parameterised constructors(we use to create object with parameters)
4. copy constructor

1. Dummy constructor: If there is no constructor in the class when the time of compile a program compiler will creates the dummy constructor with the same name with empty body.

Example.java
```
class Example {
        Void display()
            {}
}
```

Example.class
```
class Example{
    Example()  // dummy constructor created by compiler
            {}
        Void display()
            {}
}
```

2. Default Constructor

- Default constructor refers to a constructor that is automatically created by compiler in the absence of explicit constructors.
- You can also call a constructor without parameters as default constructor because all of its class instance variables are set to default values.

3. Parameterized Constructor:

- Parameterized constructors are required to pass parameters on creation of objects.
- If we define only parameterized constructors, then we cannot create an object with default constructor. This is because compiler will not create default constructor. You need to create default constructor explicitly.

4. Copy Constructor:

- A copy constructor is used to create another object that is a copy of the object that it takes as a parameter. But, the newly created copy is actually *independent* of the original object. It is independent in the sense that the copy is located at a completely different address in memory than the original.

Use of this and super keywords in constructors:

The use of "this":

- Constructors and methods use the keyword this quite differently. A method uses this to refer to the instance of the class that is executing the method. Static methods do not use this; they do not belong to a class instance, so this would have nothing to reference. Static methods belong to the class as a whole, rather than to an instance. Constructors use this to refer to another constructor in the same class with a different parameter list.

The use of "super":

- Methods and constructors both use super to refer to a superclass, but in different ways.
- Methods use super to execute an overridden method in the superclass.

Topic Name9: Access Modifiers (Default, Public, Private, Protected):

1. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible.
2. Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified.
3. The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.
4. Java Access Modifiers regulate access to classes, fields and methods in Java. These Modifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class.
5. Access Modifiers can be used to restrict access. Ex. Private access Modifiers

6. Access Modifiers are an integral part of object-oriented programming.
7. Access Specifier/Modifier in defines the boundary and scope for accessing the method, variable and class etc.

Java has defined 4 types of access modifiers:

1. Default (Visible to the package. No modifiers are needed.)
2. Public(Visible to the world)
3. Private(Visible to the class only)
4. Protected(Visible to the package and all subclasses)

The following rules for inherited methods are enforced:

1. Methods declared public in a superclass also must be public in all subclasses.
2. Methods declared protected in a superclass must either be protected or public in subclasses; they can't be private.
3. Methods declared without access control (no modifier was used) can be declared more private in subclasses.
4. Methods declared private are not inherited at all, so there is no rule for them.

Syntax for declaring a class, variables & methods with access specifiers :

For classes: <access-modifier> <class-keyword> <class-name>

For variables: <access-modifier> <data type> <instance variable>

For methods: <access-modifier> <return type> <method name>

Default Access Modifiers:

1. If you don't use any modifier for variables/methods/class/constructor, it is treated as **default** by default. The default modifier is accessible only within package.
2. Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
3. A variable or method declared without any access modifier is available to any other class in the same package.
4. The fields in an interface are implicitly public static final and the methods in an interface are by default public.
5. Java provides default modifiers which are used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Public Access Modifiers:

1. The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
2. If you declare any method/function as the public access modifier than that variable/method can be used anywhere.
3. The instance variables, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

4. In any other package if we want to access the class defined method with public access modifier then we only need to import that package. i.e. we can access the public access modifier methods in outside the package also.
5. Public modifiers achieves the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.
6. Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.
7. A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Private Access Modifiers:

1. The private access modifier is accessible only within class.
2. Private Modifiers achieve the lowest level of accessibility.
3. private methods and fields or instance variables can only be accessed within the same class to which the methods and fields belong.
4. private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access modifier is opposite to the public access modifier.
5. Using Private access modifier we can achieve encapsulation and hide data from the outside world.
6. If you declare any method/variable as private than it will only accessed in the same class which declare that method/variable as private.
7. The private members cannot access in the outside world.
8. If any class declared as private than that class will not be inherited.
9. The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.
10. Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.
11. Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
12. Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

13. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Protected Access Modifier:

1. The **protected access modifier** is accessible within package and outside the package but through inheritance only.
2. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
3. It has same properties as that of private access modifier but it can access the methods/variables in the child class of parent class in which they are declared as protected.
4. When we want to use the private members of parent class in the child class then we declare those variables as protected.
5. Variables, methods and constructors which are declared protected in a super class can be accessed only by the subclasses in other package or any class within the package of the protected members' class.
6. The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in an interface cannot be declared protected.
7. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.
8. Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package of the protected members' class.
9. The protected fields or methods cannot be used for classes and Interfaces.
10. It also cannot be used for fields and methods within an interface.
11. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages.
12. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.
13. The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Topic Name10: Imp keywords in Java (super, this, static, abstract, final):

Java Language Keywords:

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

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

10.1 Keyword This:

1. ***this*** is a keyword in Java. Which can be used inside *method* or *constructor* of class.
2. It(***this)*** works as a reference to current object whose method or constructor is being invoked.
3. ***this*** keyword can be used to refer any member of current object from within an instance method or a constructor.
4. **this** keyword can be used to refer current class instance variable.

this keyword with field(instance Variable):

1. *this* keyword can be very useful in case of Variable Hiding.
2. We can't create two instance/Local Variable with same name. But it is legal to create One instance Variable & One Local Variable or Method parameter with same name. In this scenario Local Variable will hide the Instance Variable which is called Variable Hiding.
3. As you can instance Variable is hiding here and value getting displayed is the value of Local Variable(or Method Parameter) and not Instance Variable. To solve this problem ***this*** keyword can be used with field to point Instance Variable instead of Local Variable.
4. If there is ambiguity between the instance variable & constructor parameter, this keyword resolves the problem of ambiguity.

```
Ex: class Example{
    int id; // instance variable
    String name; // instance variable
        Example( int id, String name){ //constructor with parameters id, name
     id = id;
     name = name;
     }
   }
```

5. In the above example, parameters (arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

In java, this is a **reference variable** that refers to the current object.

Ex: class Example{

```
    int id; // instance variable
    String name; // instance variable
    Example(int id, String name){ //constructor with parameters id, name
    this.id = id;
    this.name = name;
     }
   }
```

6. If local variables (arguments) and instance variables are different, there is no need to use this keyword.
7. bbbbbbbb

***this* keyword with Constructor:**

1. ***this*** keyword can be used inside Constructor to call another overloaded Constructor in same class. It is called Explicit Constructor Invocation.
2. If a class has two overloaded constructor one without argument and another with argument. Then ***this*** keyword can be used to call Constructor with argument from Constructor without argument. As constructor can't be called explicitly. (or)
3. this() function or method can be used to invoke current class constructor.
4. The this() can be used to invoke the current class constructor (more than one constructor). This approach is better if you have many constructors in the class and want to reuse that constructor. Where we use the this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.
5.

Note: 1. ***this*** keyword can only be the first statement in Constructor
1. A constructor can have either ***this*** or ***super*** keyword but not both.

***this* keyword with Method:**

this keyword can also be used inside methods to call another method from same class.

10.2 Static Keyword:

The **static keyword** is used in java mainly for memory management.
We may apply static keyword with variables, methods, blocks and nested class.
The static keyword belongs to the class (class area) than instance of the class(heap area).
We can use static keyword for

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class (we can't use static keyword for main class)

10.2.1) static variable:

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading(loading into JVM).
- instance variable gets the memory at the time of object creation.

Advantage of static variable:

It makes your program **memory efficient** (i.e. it saves memory).

Understanding problem without static variable:

```
class Example{
    int empno;
    String name;
   static String company="CTS";
}
```

Suppose there are 500 employees in any company, now all instance data members will get memory each time when object is created. All emps have its unique rollno and name so instance data member is good. Here, company refers to the common property of all objects. If we make it static, this field will get memory only once.

Note: static variable/static property is shared to all objects.

10.2.2) static method:

If you 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 can be 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.

Note:

3. The static method cann't use non static data member or call non-static method directly.
4. this and super cannot be used in static context.

Note: why main method is static?

Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

10.2.3) static block:

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading(loading .class file into JVM).

10.3 Keyword Super:

Accessing Super class Members:

2. If your method overrides one of its super class's methods, you can invoke the overridden method through the use of the keyword super.

Ex. Within Subclass, the simple name method() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to method() inherited from Superclass, Subclass must use a qualified name, i. e. using super key word we can call the super class method from subclass. When we compiling and executing Subclass then it prints superclass method and subclass method.

Subclass Constructors

Here we will know how to use the super keyword to invoke a superclass's constructor.
Invocation of a superclass constructor must be the first line in the subclass constructor.
The syntax for calling a superclass constructor is
super();

(or)

super(parameter list);
With super(), the superclass no-argument constructor is called.
With super(parameter list), the superclass constructor with a matching parameter list is called.
Ex. See program `DefaultConstructor.java`

The **super** is a reference variable that is used to refer immediate parent class object.

Advantages of Keyword Super:

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.
4. super is used to invoke immediate parent class method and also if both classes have same method if we call that method from Derived class, first it will call the method of Derived class not of Parent class because priority is given to local.
5. In case there is no method in subclass as parent, there is no need to use super. For example the method is invoked from Derived class but Derived class does not have the Parent class method, so you can directly call Parent class method.

10.4 Final Keyword:

What is final in Java? Final variable , Method and Class Example
1. Final in java is very important keyword and can be applied to class, method and variables in java.
2. Final is often used along with static keyword in java to make static final constant and you will see how final can increase performance of java application.

Final keyword in Java: Final is a keyword or reserved word in java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise **compilation error** if you try to re-initialized **final variables in java**.

Final variable in Java: Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable. Final variables are often declare with static keyword in java and treated as constant. Here is an example of final variable in Java

```
public static final String LOAN = "loan";

LOAN = new String("loan") //invalid compilation error
```

Final variables are by default read-only.

Final method in Java: Final keyword in java can also be applied to methods. A java method with final keyword is called final method and it can't be overridden in sub-class. You should make a method final in java if you think it's complete and its behavior should remain constant in sub-classes. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time. Here is an *example of final method in Java*:

```
class Parent{
 public final String getName(){
     return "parent";
 }
}
class Child extends Parent{
    public final String getName(){
        return "child"; //compilation error: overridden method is final
```

```
    }
}
```

What is final Class in Java

Java class with final modifier is called final class in Java. Final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer and other wrapper classes. Here is an *example of final class in java*

```
final class Parent{

}

Class Child extends Parent{  //compilation error: cannot inherit from final class

}
```

Benefits of final keyword in Java

Here are few benefits or advantage of using final keyword in Java:

1. Final keyword improves performance. Not just JVM can cache **final variable** but also application can cache frequently use final variables.

2. Final variables are safe to share in multi-threading environment without additional synchronization overhead.

Final and Immutable Class in Java

Final keyword helps to write immutable class. Immutable classes are the one which can not be modified once it gets created and String is primary example of immutable and final class. Immutable classes offer several benefits one of them is that they are effectively read-only and can be safely shared in between multiple threads without any synchronization overhead. You can not make a class immutable without making it final and hence final keyword is required to make a class immutable in java.

Example of Final in Java

Java has several system classes in JDK which are final, some example of final classes are String, Integer, Double and other wrapper classes. You can also use final keyword to make your code better whenever it required.

Summary on Final Keyword:

1. **Final keyword** can be applied to member variable, local variable, method or class in Java.
2. **Final member variable** must be initialized at the time of declaration or inside constructor, failure to do so will result in compilation error.
3. You can't reassign value to *final variable in Java*.
4. **Local final variable** must be initializing during declaration.
5. Only final variable is accessible inside anonymous class in Java.
6. **Final method** can't be overridden in Java.
7. **Final class** can't be inheritable in Java.
8. **Final** is different than **finally** keyword which is used on Exception handling in Java.
9. Final should not be confused with finalize() method which is declared in object class and called before an object is garbage collected by JVM.
10. All variable declared inside java interface are implicitly final.
11. **Final and abstract** are two opposite keyword and a final class can't be abstract in java.
12. Final methods are bonded during compile time also called static binding.
13. *Final variables* which is not initialized during declaration are called blank final variable and must be initialized on all constructor either explicitly or by calling this(). Failure to do so compiler will complain as "*final variable (name) might not be initialized*".
14. Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.
15. As per Java code convention **final variables are treated as constant** and written in all Caps
e.g: private final int COUNT=10;
16. Making a collection reference variable final means only reference can not be changed but you can add, remove or change object inside collection. For example:

```
private final List Loans = new ArrayList();
list.add("home loan");  //valid
list.add("personal loan"); //valid
loans = new Vector();  //not valid
```

10.5 keyword abstract:

Abstraction:

1. Abstraction refers to the ability to make a class abstract in OOPs.
2. **Abstraction** is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details.

   Ex: sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

3. Abstraction lets you focus on what the object does instead of how it does it.
4.
5. An abstract class is one that cannot be instantiated (You just cannot create an instance of the abstract class.). All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.
6.
7.

Ways to achieve Abstaction:

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

abstract class:

1. A class that is declared with abstract keyword is known as **abstract class**.
2. It needs to be extended and its method implemented. we don't know about the implementation class (i.e. hidden to the end user)
3. Ex: In my module, we are writing API calls to connect and to get data from AMDOCs maintained my IBM. i.e abstract part maintained by my module & implementation part maintained by AMDOCs.
4. It cannot be instantiated. i.e. can't create object for abstract class directly but it must be extend.
5.

Syntax to declare the abstract class:

```
abstract class <class_name>{}
```

abstract() method:

A method that is declared as abstract and does not have implementation is known as abstract method.

Syntax to define the abstract method:

```
abstract return_type <method_name>();//no curly braces{}
```

Imp points:

1. An abstract class can have data member, abstract method, method body, constructor and even main() method.
2. An abstract class that have method body and not possible in interface. i.e. abstract class have implemented and unimplemented methods whereas interface have only unimplemented methods(just declaring the methods).
3. Abstract classes have declared and implemented methods.
4. If there is any abstract method in a class, that class must be abstract. Otherwise we will get compile time error.
5. If you are extending any abstact class that has abstract method, you must either provide the implementation of the method.

Topic Name11: Interfaces

Interfaces: looks like class and it contains abstract methods and public static final variables.

Some of the imp advantages and disadvantages of Abstract Classes:

Java Abstract classes are used to declare common characteristics of subclasses.

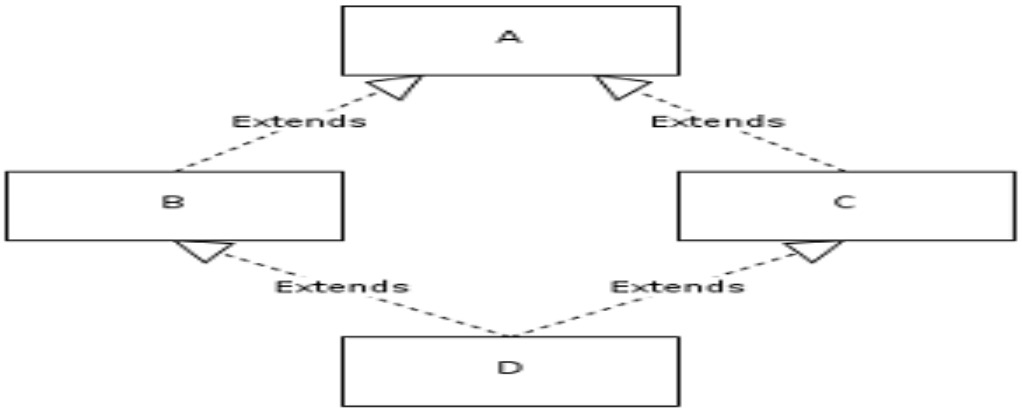
1. An abstract class cannot be instantiated. It can only be used as a super class for other classes that extend the abstract class.
2. Abstract classes are declared with the abstract keyword.
3. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.
4. Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform.
5. An abstract class can include methods that contain no implementation. These are called abstract methods.
6. The abstract method declaration must then end with a semicolon rather than a block.
7. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract.
8. Abstract methods are used to provide a template for the classes that inherit the abstract methods.
9. Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods.
10. Any implementation specified can, of course, be overridden by additional subclasses.
11. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

Disadvantage of Abstract class: A big Disadvantage of using abstract classes is not able to use multiple inheritances. In the sense, when a class extends an abstract class, it can't extend any other class.

Introduction to interfaces:

In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**.

1. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface.
2. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final).
3. Interface definition begins with a keyword interface.
4. An interface like that of an abstract class cannot be instantiated.
5. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.
6. Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.
7. As we know that in Java, we cannot use multiple inheritance means one class cannot be inherited from more than one classes. To use this process, Java provides an alternative approach known as "Interface".
   Ex:



We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method, which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

8. If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

9. Java expands the concept of Abstract class with Interface means its members are not Implemented. A java class cannot be a sub class of more than one class, but a class can implements more than one Interfaces.

10. An Interface is usually a kind of class like classes and interface both contain methods and variables, but with a major difference i.e. Interface contain only Abstract methods and Final variable. This means that interface do not specify any code to implements these methods and data fields.

11. A class implements an interface, thereby inheriting the abstract methods of the interface.

12. General example for interface: objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off. (or)

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car.

13.

An interface is similar to a class in the following ways:

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

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

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

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Implementing Interfaces:

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

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration

When overriding methods defined in interfaces there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

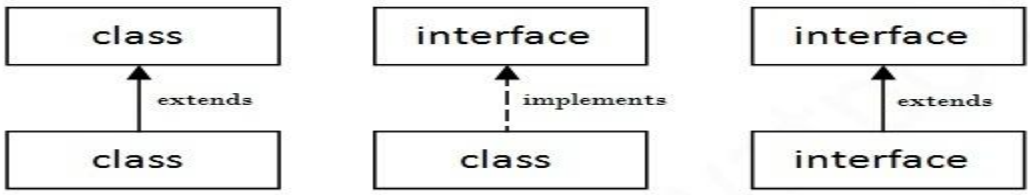
- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, similarly to the way that a class can extend another class.
-

Extending Interfaces:

1. An interface can extend another interface,

similarly to the way that a class can extend another class.

2. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.



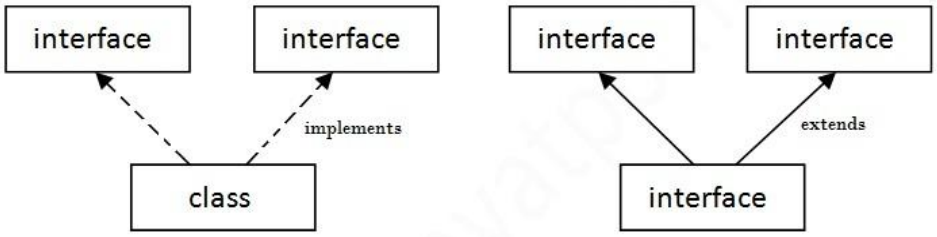
Extending Multiple Interfaces:

1. A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.
2. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list. For example, if the Hockey interface extended both interfaces Sports and Event, it would be declared as:

public interface Hockey **extends** Sports, Event

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

Syntax to create an interface:

```
<modifier> interface <interface name>
    {
      …….  //interface fields(By default Public, Static and Final)
      …….  //interface methods(By default Public and Abstract)
    }
```

Ex:

```
public interface interfaceA{
        public static final String empName = "BV Narayana";
        public void display();
}
public interface interfaceB {
        String empName = "Namish";  // by default it is public static final
        void show(); // by default it is public
 }

public class Example implements interfaceA, interfaceB {
         public void display(){
             System.out.println("I am in implemented class display method");
          }
         public void show(){
             System.out.println("I am in implemented class show method");
          }
    public static void main(String args[]){
      Example obj = new Example();
             obj.display();
             obj.show();
    }
}
```

compilation: javac Example.java
execution: java Example
output: I am in implemented class display method
I am in implemented class show method

There are some points that are followed:

a. The methods in interface are "public" and "abstract" by default.

b. The variables/Fields in interface are "public", "static" and "final" By default.

c. To inherit an interface to a class we use "implements" keyword.

d. To inherit two interfaces or two classes we use "extends" keyword.

Possible combinations of extends and implements keywords:

1. class classname implements interfacename
2. **class** classname **implements Interface1, interface2**
3. class classname extends classname implements Interfacename
4. **class** classname **extends classname implements Interface1, inteface2**
5. **class** classname **extends class1, class2 ----------illegal(this is the problem in inheritance)**
6. class classname implements classname extends Interfacename ----not acceptable –we will get compile time error.
7. interface interfacename extends interfacename
8. **interface interfacename extends interface1, interface2**
9. class classname extends interface1, interface2 -----not acceptable

Topic Name12: Arrays

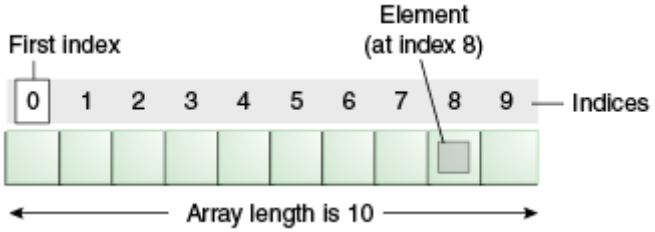
Arrays :

1. In java Arrays acts like an object.
2. Fully qualified name: java.util.Arrays
3. This class contains various methods for manipulating arrays (such as sorting and searching). Bz arrays are massively used in programming, as they provide a very fast and easy way to store and access pieces of data.
4. The methods in this class all throw a `NullPointerException` if the specified array reference is null, except where noted.
5. We have sort(int[] a): Sorts the specified array of into ascending numerical order's.
   The `java.util.Arrays` class has static methods for sorting arrays, both arrays of primitive types and object types. The sort method can be applied to entire arrays, or only a particular range. For Object types you can supply a *comparator* to define how the sort should be performed.

6. In java, array is an object that contains elements of similar data type. i.e. Array is a collection of similar type of elements. For example a String array can only contain Strings and an int array can only contain int objects.
   a) An array variable does not actually store the array - it is a reference variable that points to an array object.
   b) Declaring the array variable does not create an array object instance; it merely creates the reference variable - the array object must be instantiated separately.
   c) Once instantiated, the array object contains a block of memory locations for the individual elements.
   d) If the individual elements are not explicitly initialized, they will be set to zero.
   e) Arrays can be created with a size that is determined dynamically (but once created, the size is fixed).
7. It is a data structure where we store similar elements.

8. We can store only fixed elements in an array.

9. Array is index based; first element of the array is stored at 0 index.
   Example: An array of 10 elements. Each item in an array is called an *element* and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.



Declaring a Variable to Refer to an Array:

The preceding program declares an array (named `anArray`) with the following line of code:

```
// declares an array of integers
    int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *type*`[]`, where *type* is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array.
The size of the array is not part of its type (which is why the brackets are empty).
An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.
Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

More info about declare an array variables:

1. **To Declare an array variable by specifying the type of data to be stored, followed by square brackets [].**

   ```
   dataType[] variableName;
   ```

   You can read the `[]` as the word "array".
2. To declare a variable for an array of integers:

   ```
   int[] nums;
   ```

   ...which you can read as "`int` array `nums`".
3. To declare a variable for an array of `String` objects:

   ```
   String[] names;
   ```

   ...which you can read as "`String` array `names`" - the array holds `String` references.

Note: you may also put the brackets after the variable name (as in C/C++), but that is less clearly related to how Java actually works.

```
int nums[]; // not recommended, but legal
```

But, that syntax does allow the following, which is legal, but seems like a bad practice.

```
int nums[], i, j; // declares one array and two single int values
```

Creating, Initializing and Accessing an Array:

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for 10 integer elements and assigns the array to the `anArray` variable.

// create an array of integers

```
int[] anArray;

anArray = new int[10];          (or)

int[] anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following, and compilation fails: XXXXXX(classname).java:4: Variable anArray may not have been initialized.

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

More info:

Instantiate an array object using `new`, the data type, and an array size in square brackets

```
int[] nums;
    nums = new int[10];
```

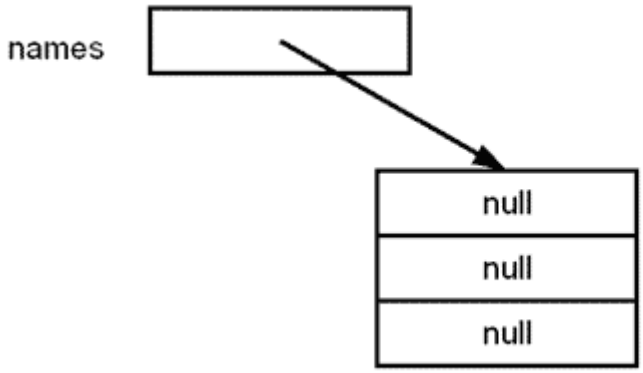
The second line constructs a new array object with 10 integer elements, all initialized to 0, and stores the reference into `nums`.

```
int[] moreNums;
int size = 7;
moreNums = new int[size];
```

You can declare and instantiate all at once:

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

The elements of the array, `String` references, are initialized to `null`.



As objects, arrays also have a useful property: `length`:

- In the above example, `names.length` would be 3.
- The property is fixed (i.e., it is read-only).

You can reassign a new array to an existing variable:

```
int[] nums;
nums = new int[10];
nums = new int[20];
```

The original ten-element array is no longer referenced by `nums`, since it now points to the new, larger array.

Note: When you declare a Java array, you need to give it a data type when you declare it just like you would give a variable a datatype.
Example: we will declare an integer array containing five elements which we will name arrayName

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

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

More info:

An array can be initialized when it is created

The notation looks like this:

```
String[] names = { "Joe", "Jane", "Herkimer" };
```

or

```
String[] names = new String[] {"Joe", "Jane", "Herkimer" };
```

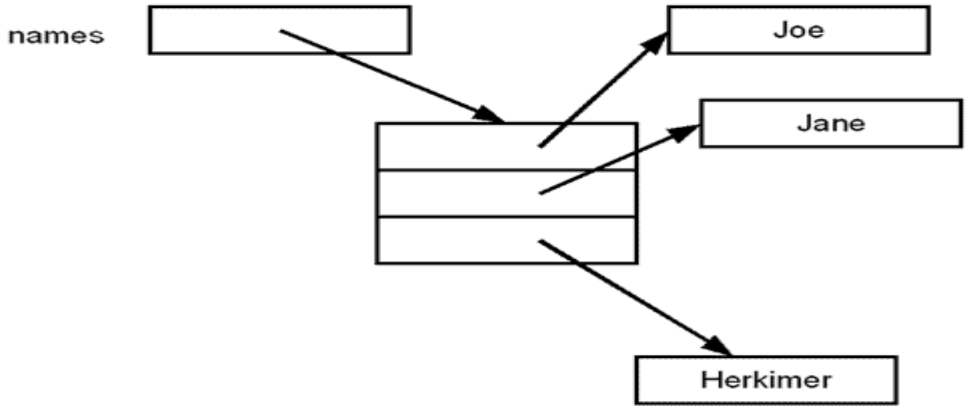
This automatically creates an array of length 3, because there were 3 items supplied.

```
int[] nums = new int[] { 2, 4, 6, 8, 10, 12 };
```

This array will have a length of 6.

If a new array is being assigned to an existing variable, you cannot use the shorter variant; you must use the new keyword and the data type:

```
String[] names;
names = new String[] {"Joe", "Jane", "Herkimer" };
```



Two Dimensional Arrays:

In many applications, a natural way to organize information is to use a table of numbers organized in a rectangle and to refer to rows and columns in the table. The mathematical abstraction corresponding to such tables is a *matrix*; the corresponding Java construct is a two-dimensional array.

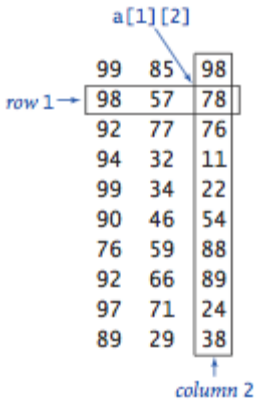
- *Two-dimensional arrays in Java.* To refer to the element in row i and column j of a two-dimensional array a[][], we use the notation a[i][j]; to declare a two-dimensional array, we add another pair of brackets; to create the array, we specify the number of rows followed by the number of columns after the type name (both within brackets), as follows:

```
double[][] a = new double[M][N];
```

We refer to such an array as an *M-by-N array*. By convention, the first dimension is the number of rows and the second dimension is the number of columns. As with one-dimensional arrays, Java initializes all entries in arrays of numbers to 0 and in arrays of Booleans to false.

- *Initialization.* Default initialization of two-dimensional arrays is useful because it masks more code than for one-dimensional arrays. To access each of the elements in a two-dimensional array, we need nested loops:

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    a[i][j] = 0;
```



99	85	98
98	57	78
92	77	76
94	32	11
99	34	22
90	46	54
76	59	88
92	66	89
97	71	24
89	29	38

Anatomy of a two-dimensional array

Advantage of Array:

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

Disadvantage of Arrays:

Size Limit: We can store only 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.

Topic Name13: Java Control flow statements

Java Control statements control the order of execution in a java program, based on data values and conditional logic. There are three main categories of control flow statements;

These control structures can be classified into three groups:

1. **Decision Making statements or Selection statements**
2. **Repetition statements or Loop statements**
3. **Branching statements or Transfer statement**

· Selection or Decision making statements**(if, if-else & switch**): Decision making statements are used to decide whether or not a particular statement or a group of statements should be executed.

· Loop or Iteration or Repetition statements(**while, do-while & for**): Repetition statements are used to execute a statement or a group of statements more than once.

Transfer or Branching statements(**break, continue, return & assert**): Branching statements are used to transfer control to another line in the code.

We use control statements when we want to change the default sequential order of execution.

1) Decision Making or Selection statements(if, if-else, if-else..if-else **& switch)**

Decision making statements are used to decide whether or not a particular statement or a group of statements should be executed.

2) **Repetition or Loop statements(while, do-while and for):**
   Repetition statements are used to execute a statement or a group of statements more than once.
3) **Branching or Transfer statements(break, continue, return and assert):**
   Branching statements are used to transfer control to another line in the code.

Decision Making or Selection Statements:

The if Statement:

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program. You can either have a single statement or a block of code within an if statement. Note that the conditional expression must be a Boolean expression.

The simple if statement has the following syntax:

```
if (<conditional expression>)
  <statement action>
```

Below is an example that demonstrates conditional execution based on if statement condition.

The If-else Statement

The if/else statement is an extension of the if statement. If the statements in the if statement fails, the statements in the else block are executed. You can either have a single statement or a block of code within if-else blocks. Note that the conditional expression must be a Boolean expression.

The if-else statement has the following syntax:

```
if (<conditional expression>)
<statement action>
else
<statement action>
```

Switch Case Statement:

The switch case statement, also called a case statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements. The switch statement begins with a keyword, followed by an expression that equates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label. The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed.

Java includes a default label to use in cases where there are no matches. We can have a nested switch within a case block of an outer switch. Its general form is as follows:

```
switch (<non-long integral expression>) {
case label1: <statement1>
case label2: <statement2>
...
case labeln: <statementn>
default: <statement>
} // end switch
```

if you want to exit in the middle of the switch statement code block, you must insert a break statement, which causes the program to continue executing after the current code block.

Ternary operator:

The last of the decision making statements is the ternary operator which provides a concise form for short and simple if lese statements. The ternary operator is represented by ?: Following is the syntax of ternary operator:

expression1 ? expreession2 : expression3

If expression1 evaluates to true, then expression 2 is evaluated, otherwise expression 3 is evaluated. The ternary operators can be used as a standalone statement or in conjugation with other statements like declaration and print statements. Following examples illustrate the use of ternary operator:

Repetitive or Iteration or Loop Statements

While Statement:

The while statement is a looping construct control statement that executes a block of code while a condition is true. You can either have a single statement or a block of code within the while loop. The loop will never be executed if the testing expression evaluates to false. The loop condition must be a boolean expression.

The syntax of the while loop is:

```
while (<loop condition>)
<statements>
```

Do-while Loop Statement:

The do-while loop is similar to the while loop, except that the test is performed at the end of the loop instead of at the beginning. This ensures that the loop will be executed at least once. A do-while loop begins with the keyword do, followed by the statements that make up the body of the loop. Finally, the keyword while and the test expression completes the do-while loop. When the

loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop. You can either have a single statement or a block of code within the do-while loop.

The syntax of the do-while loop is:

```
do
<loop body>
while (<loop condition>);
```

For Loops:

The for loop is a looping construct which can execute a set of instructions a specified number of times. It's a counter controlled loop.

The syntax of the loop is as follows:

```
for (<initialization>; <loop condition>; <increment expression>)
<loop body>
```

The first part of a for statement is a starting initialization, which executes once before the loop begins. The <initialization> section can also be a comma-separated list of expression statements. The second part of a for statement is a test expression. As long as the expression is true, the loop will continue. If this expression is evaluated as false the first time, the loop will never be executed. The third part of the for statement is the body of the loop. These are the instructions that are repeated each time the program executes the loop. The final part of the for statement is an increment expression that automatically executes after each repetition of the loop body. Typically, this statement changes the value of the counter, which is then tested to see if the loop should continue.
All the sections in the for-header are optional. Any one of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the <loop condition> signifies that the loop condition is true. The (;;) form of for loop is commonly used to construct an infinite loop.

Transfer Statements

Continue Statement:

A continue statement stops the iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop. You use a continue statement when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.

The syntax of the continue statement is

```
continue; // the unlabeled form
continue <label>; // the labeled form
```

You can also provide a loop with a label and then use the label in your continue statement. The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

Break Statement:

The break statement transfers control out of the enclosing loop (for, while, do or switch statement). You use a break statement when you want to jump immediately to the statement following the enclosing control structure. You can also provide a loop with a label, and then use the label in your break statement. The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

The syntax for break statement is as shown below;

break; // the unlabeled form
break <label>; // the labeled form

Topic Name14: Object, String, StringBuffer, StringBuilder and StringTokenizer classes

java.lang.Object,

java.lang.String,

java.lang.StringBuffer,

java.lang.StringBuilder,

java.util.StringTokenizer classes.

Object Class:

1. Fully qualified name is java.lang.Object
2. **Object Class is a super class for all classes.**
3. The Object class, in the java.lang package, sits at the top of the class hierarchy tree.
4. Every class is a descendant, direct or indirect, of the Object class.
5. Every class you use or write inherits the instance methods of Object.
6. You need not use any of these methods but if you choose to do so, you may need to override them with code that is specific to your class.

	Object class methods & description
1	protected Object clone() This method creates and returns a copy of this object.
2	boolean equals(Object obj) This method indicates whether some other object is "equal to" this one.

3	protected void finalize() This method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
4	Class getClass() This method returns the runtime class of this Object.
5	int hashCode() This method returns a hash code value for the object.
6	void notify() This method wakes up a single thread that is waiting on this object's monitor.
7	void notifyAll() This method wakes up all threads that are waiting on this object's monitor.
8	String toString() This method returns a string representation of the object.
9	void wait() This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
10	void wait(long timeout) This method causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
11	void wait(long timeout, int nanos) This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

More info about Object class methods:

getClass() - Returns the runtime class of an object.

hashCode() - Returns a hash code value for the object.

toString() - Returns a string representation of the object.

clone() - Creates and returns a copy of this object.

equals() - Indicates whether some other object is "equal to" this one.

finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. system.gc();

wait() - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

notify() - Wakes up a single thread that is waiting on this object's monitor.

notifyAll() - Wakes up all threads that are waiting on this object's monitor.

More info about clone():

Cloning of Object:

Clone is nothing but the process of copying one object to produce the exact object, which is not guaranteed. We all know in Java object is referred by reference we can't copy one object directly to another object. So we have cloning process to achieve this objective. Now one question arise in mind why we need this process so the answer is whenever we need local copy of the object to modify the object in some method but not in method caller. So we can define Cloning as "**create a copy of object** ".

What is cloning: Cloning is the process of creating another copy of an object. A clone of an object can be created by calling the clone() method of the class implementing the Cloneable interface.
By default, all classes inherit the clone() method from Object class but to actually clone an object of a class, implementation of the Cloneable interface is mandatory. If the clone() method is called on an object without its class implementing the Cloneable interface then a CloneNotSupported exception is thrown. The clone() method inherited from Object class can be overridden to make the cloning operation customized. The Cloneable interface is a marker/tag interface like Serializable interface.

A shallow copy is what the default implementation of the clone method in Object class returns. For deep cloning, the clone() method needs to be overridden.

Depending upon how we are doing this copy, we can divide cloning in two types.

- Shallow Copy: The default behavior of clone() method inherited from the Object class is known as shallow cloning.
- Deep Copy: deep cloning can be achieved by overriding the clone() method.

Before going into the deep of shallow and deep copy we need to understand how we achieve cloning in java

Object Cloning in Java:

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. protected Object clone() throws CloneNotSupportedException

How to Clone in java?

In Java everything is achieved through class, object and interface .By default no Java class support cloning but Java provide one interface called Cloneable, which is a marker interface and by implementing this interface we can make duplicate copy of our object by calling clone() method of java.lang.Object class. This Method is protected inside the object class and Cloneable interface is a marker interface and this method also throw **CloneNotSupportedException** if we have not implement this interface and try to call clone() method of Object class. By default any clone() method

gives **shallow copy** of the object i.e. if we invoke super. clone() then it's a shallow copy but if we want to **deep copy** we have to override the clone() method and make it public and give own definition of making copy of object. Now we let's see what is shallow and deep copy of object in Java programming language.

Depending upon how we are doing this copy, we can divide cloning in two types.

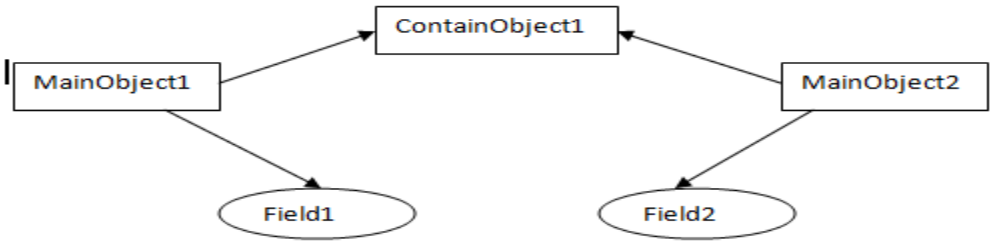
- Shallow Copy
- Deep Copy

Shallow Copy

Whenever we use default implementation of clone method we get shallow copy of object means it create new instance and copy all the field of object to that new instance and return it as **object type** we need to explicitly cast it back to our original object. This is shallow copy of the object. clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non-primitive or reference type variable In shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name shallow copy or shallow cloning in Java. If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

What is Shallow Copy?

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.



In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a shallow copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and still pointing to ContainObject1 itself. Observe here and you will find that since field1 is of primitive type, the values of it are copied to field3 but ContainedObject1 is an object, so MainObject2 is still pointing to ContainObject1. So any changes made to ContainObject1 in MainObject1 will reflect in MainObject2.

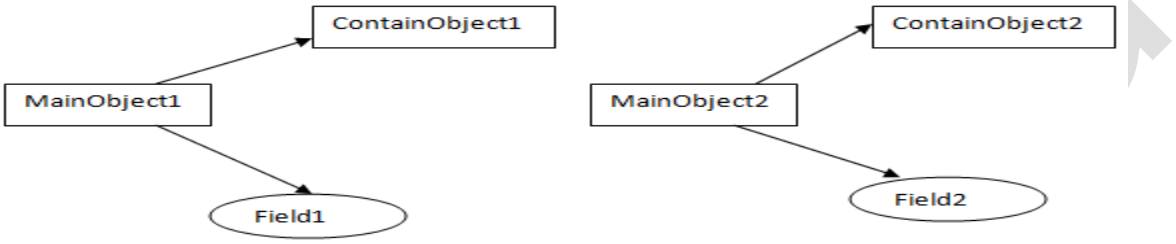
Deep Copy

Whenever we need own meaning of copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need. So for deep

copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class. After that we override the clone() method in all those classes even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone() method of Object class on the instances of those classes inside some other class. It's typical restriction of the protected access.

What is Deep Copy?

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.



In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a deep copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and "ContainObject2" containing the copied value of ContainObject1.So any changes made to ContainObject1 in MainObject1 will not reflect in MainObject2.

What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
        public interface Serializable{
                                }
```

Marker Interface in java is an interface with no fields or methods within it. It is used to convey to the JVM that the class implementing an interface of this category will have some special behavior. Hence, an empty interface in java is called a marker interface. Marker interface is also called tag interface by some java gurus. In java we have the following major marker interfaces as under:

- Searilizable interface
- Cloneable interface
- Remote interface
- ThreadSafe interface

Usage of Marker Interface in java:

Marker interface in Java e.g. Serializable, Clonnable and Remote is used to indicate something to compiler or JVM that the class implementing any of these would have some special behavior. Hence, if the JVM sees a Class is implementing the Serializable interface it does some special operation on it and writes the state of the object into object stream. This object stream is then available to be read by another JVM. Similarly if JVM finds that a class is implementing Clonnable interface, it performs some special operation in order to support cloning. The same theory goes for RMI and Remote interface. This indication (to the JVM) can also be done using a boolean flag or a String variable inside the class.

Apart from using the built-in marker interface, to mark a class as serializable or clonnable, we can also have our own marker interface. Marker interface is a good way to logically segregate the code and also if we have our own tool to perform some preprocessing operation on the classes. It is very useful for developing frameworks or APIs e.g. struts or spring.
With the introduction of annotation in java 5, annotation has become a better choice over maker interface.

The Serializable interface:

1. The serializable interface provides a mechanism of object serialization where an object is represented as a sequence of bytes.
2. Object serialization writes the object into stream. Once an object is serialized and written into a file, it can be read from the file and deserialized.
3. Deserialization is the process of reading the object from the file and reconstructing the object in the memory.
4. The entire process of serialization and deserialization is JVM independent.
5. An object serialized by one JVM can easily be read across another JVM.

String class:

1. Fully qualified name is java.lang.String.
2. The **String** class represents character sequence of fixed size. All string literals in Java programs, such as String s = "abc"; are implemented as instances of this class.
3. Strings are constant; their values cannot be changed after they are created.
4. The String class available in every java programming because p s v m(String args[]).
5. Every String is an immutable (can't modify) object.
6. Every modification allocates new memory.
7. Java.lang is the default package; by default in every application we can use String class.
8. In java can create a String literal or String object.

   String s = "abc";// String literal

   String s = new String("abc");// String object
9. A String in java represents a char sequence of fixed size.
10. Strings are **thread safe** means multiple threads trying to change the properties of a single object, like suppose for a vector one thread is trying to take an object from vector and other trying to remove an object from the same vector. Then they will happen in an order and not both at the same time. To make this happen with String (which is immutable) is highly impossible. You can't change the properties of String object once it is created. If you want to do any operations over String, it will create a new string. So **Strings are Thread safe.**
11.

We can create String instance in two ways:

1. By using new dynamic memory operator:
   Note: it creates instances in both the places (Heap and SCP (secondary collection pooling)) at a time.

   String p1 = new String ("ABC");

2. By using String literal or constants:
   Note: it creates instances in heap area only.

String s1 = "ABC";
String s2 = "ABC";
String s3 = "ABC123";
String s4 = "ABC123D";

While we are creating instance s1 using string literal technique, we will create instance only in heap area. Where as in scp instead of creating once again instance s1 with same content, we will refer to existing instance or we will pointing to existing instance.
Note: if we want to create instance s3 then we will create in heap area directly.

3.

Constructors of String class:

1. **String()**
   This initializes a newly created String object so that it represents an empty character sequence.
   String obj = new String();

2. **String(String original)**
   This initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
   String obj = new String("ABCDEFG");
3. **String(char[] value)**
   This allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

String abj = new String(char[]);
Char ch[] = {'j', 'a', 'v', 'a'};
String str = new String(ch);

More constructors:

String()
Constructs a new empty String.
String(String)
Constructs a new String that is a copy of the specified String.
String(char[])
Constructs a new String whose initial value is the specified array of characters.
String(char[], int, int)
Constructs a new String whose initial value is the specified sub array of characters.
String(byte[], int, int, int)
Constructs a new String whose initial value is the specified sub array of bytes.
String(byte[], int)
Constructs a new String whose initial value is the specified array of bytes.

==========

Difference between == and equals() method in Java:

1. Now we know what is equals method, how it works and what is equality operator (==) and How it compare objects, it's time to compare them. Here is some worth noting difference between equals() method and == operator in Java:
2. First difference between them is, equals() is a method defined inside the java.lang.Object class and == is one type of operator and you can compare both primitive and objects using equality operator in Java.
3. Second difference between equals() and == operator is that,
   == is used to check reference or memory address of the objects whether they point to same location or not, and **equals()** method is used to compare the contents of the object e.g. in

case of comparing String its characters, in case of Integer it's there numeric values etc. You can define your own equals() method for domain object as per business rules e.g. two Employes objects are equal if there EmployeeId is same.

4. Third difference between equals and == operator is that, You can't change the behavior of == operator but we can override equals() method and define the criteria for the objects equality.

Let clear all these differences between equals and == operator using one Java example:

```
String s1=new String("Hyderabad");
String s2=new String("Hyderabad");
```

Here we have created two string s1 and s2 now will use == and equals () method to compare these two String to check whether they are equal or not.
First we use equality operator == for comparison which only returns true if both reference variable are pointing to same object.

```
if(s1==s2) {
    System.out.println("s1==s2 is TRUE");
} else{
    System.out.println("s1==s2 is FALSE");
}
```

Output of this comparison is FALSE because we have created two objects which have different location in heap so == compare their reference or address location and return false. Now if we use equals method to check their equivalence what will be the output .

```
if(s1.equals(s2)) {
     System.out.println("s1.equals(s2) is TRUE");
} else {
     System.out.println("s1.equals(s2) is FALSE");
}
```

Output of this comparison is TRUE because java.lang.String class has already overridden the equals() method of Object class and check that contents are same or not because both have same value hello so they are equal according to String class equals() method .

Points to remember:

1. If you have not overridden equals() method in a user defined object, it will only compare the reference or memory address, as defined in default equals() method of java.lang.Object class and return true only if both reference variable points to same object.
2. In a user defined class, both equals() and == operator behave similarly but that may not be logically correct and that's why we should always define the equivalence criteria for custom or domain objects.

That's all on difference between equals() method and == operator in Java. Both can compare objects for equality but equals() is used for logical and business logic comparison while == mostly for object reference comparison in Java.

How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable.
We can also create immutable class by creating final class that has final data members as the **example:**

1. Create a final class named as final class Example
2. Create one final data member,
3. Create a parameterized constructor and

4. Create getter method.
5. The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
6. The class is final so we cannot create the subclass.
7. There is no setter methods i.e. we have no option to change the value of the instance variable.

Disadvantage of String class:

1. One of its biggest strength Immutability is also biggest problem of Java String if not used correctly. many a times we create a String and then perform a lot of operation on them e.g. converting string into uppercase, lowercase, getting substring out of it , concatenating with other string etc. Since String is an immutable class every time a new String is created and older one is discarded which creates lots of temporary garbage in heap.
2. If String are created using String literal they remain in String Collection Pool. To resolve this problem Java provides us two Classes **StringBuffer and StringBuilder**.
3. String Buffer is an older class but StringBuilder is relatively new and added in JDK 5.

Some of the important String class methods:

charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

compareTo

`public int compareTo(String anotherString)`

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns `0` exactly when the `equals(Object)` method would return `true`.

compareToIgnoreCase

`public int compareToIgnoreCase(String str)`

Compares two strings lexicographically, ignoring case differences. This method returns an integer whose sign is that of calling `compareTo` with normalized versions of the strings where case differences have been eliminated by calling `Character.toLowerCase(Character.toUpperCase(character))` on each character.

Note that this method does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *collators* to allow locale-sensitive ordering.

concat

`public String concat(String str)`

Concatenates the specified string to the end of this string.

If the length of the argument string is `0`, then this `String` object is returned. Otherwise, a new `String` object is created, representing a character sequence that is the concatenation of the character sequence represented by this `String` object and the character sequence represented by the argument string.

contains

`public boolean contains(CharSequence s)`

Returns true if and only if this string contains the specified sequence of char values.

equals

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

equalsIgnoreCase

```
public boolean equalsIgnoreCase(String anotherString)
```

Compares this `String` to another `String`, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Two characters `c1` and `c2` are considered the same ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the == operator)
- Applying the method `Character.toUpperCase(char)` to each character produces the same result
- Applying the method `Character.toLowerCase(char)` to each character produces the same result

indexOf

```
public int indexOf(int ch)
```

Returns the index within this string of the first occurrence of the specified character. If a character with value `ch` occurs in the character sequence represented by this `String` object, then the index (in Unicode code units) of the first such occurrence is returned. For values of `ch` in the range from 0 to 0xFFFF (inclusive), this is the smallest value *k* such that:

```
this.charAt(k) == ch
```

is true. For other values of `ch`, it is the smallest value *k* such that:

```
this.codePointAt(k) == ch
```

is true. In either case, if no such character occurs in this string, then `-1` is returned.

StringBuffer class:

1. Full name is java.lang.StringBuffer
2. StringBuffer is mutable object.
3. The StringBuffer class is used to created mutable (modifiable) string.
4. The StringBuffer class is same as String except it is mutable i.e. it can be changed.
5. StringBuffers are mutable objects, means StringBuffer allocates extra empty space to allow modifications on to the given string within the same memory.
6. StringBuffer is a mutable sequence of characters. It is like a String but the contents of the StringBuffer can be modified after creation.
7. A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
8. String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

9. StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously.

10. StringBuffer is mutable sequence of characters. It can store character, append character, delete character easily.
StringBuffer is a thread safe sequence of characters, and it is synchronized.
StringBuffer is like a normal String but it can modify at any point.
StringBuffer can easily use in multi thread programs.

Differences between String and StringBuffer classes:

1. String is immutable while **StringBuffer and StringBuilder is mutable** object.
2. **StringBuffer is synchronized** while **StringBuilder is not** which makes StringBuilder faster than StringBuffer.
3. StringBuffer is thread safe and StringBuilder is not.
4. Concatenation operator "+" is internal implemented using either StringBuffer or StringBuilder.
5. Use String if you require immutability,
   use Stringbuffer in java if you need mutable + thread-safety and
   use StringBuilder in Java if you require mutable + without thread-safety.

Constructors of StringBuffer class:

StringBuffer()
Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

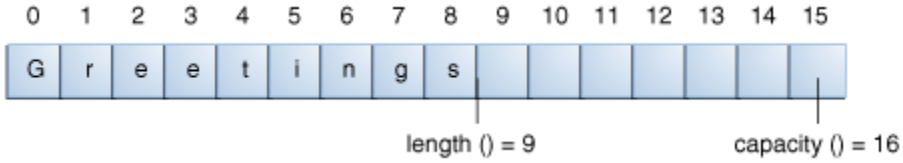
StringBuffer(String str)
Constructs a string buffer initialized to the contents of the specified string.

StringBuffer(CharSequence seq)
Constructs a string buffer that contains the same characters as the specified CharSequence.

StringBuffer(int capacity)
Constructs a string buffer with no characters in it and the specified initial capacity.

Regularly used constructors:

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.



2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Methods of StringBuffer class:

Commonly used methods of StringBuffer class:

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.

5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public int length():** is used to return the length of the string i.e. total number of characters.

8. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
9. **public char charAt(int index):** is used to return the character at the specified position.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.
12.

Some of the IMP Methods in StringBuffer class:

Method1: Java.lang.StringBuffer.ensureCapacity() Method

Description:

The **java.lang.StringBuffer.ensureCapacity()** method ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:

- The **minimumCapacity** argument.
- Twice the old capacity, plus 2.

If the **minimumCapacity** argument is nonpositive, this method takes no action and simply returns.

Declaration:

Following is the declaration for **java.lang.StringBuffer.ensureCapacity()** method

```
public void ensureCapacity(int minimumCapacity)
```

Parameters:

- **minimumCapacity** -- This is the minimum desired capacity.

Return Value:

This method does not return any value.

Method2: Java.lang.StringBuffer.ensureCapacity(int minCapacity)

ensureCapacity(int minCapacity): ensures that the capacity is at least equal to the specified minimum.

Syntax: **public void ensureCapacity(int minCapacity).**

Imp methods:

Replaces the characters in a substring of this sequence with characters in the specified `String`.

`StringBuffer` **`append`**`(boolean b)`
Appends the string representation of the `boolean` argument to the sequence.

`StringBuffer` **`append`**`(char c)`
Appends the string representation of the `char` argument to this sequence.

`StringBuffer` **`reverse`**`()`
Causes this character sequence to be replaced by the reverse of the sequence.

`void` **`setCharAt`**`(int index, char ch)`
The character at the specified index is set to `ch`.

`void` **`setLength`**`(int newLength)`
Sets the length of the character sequence.

`CharSequence` **`subSequence`**`(int start, int end)`
Returns a new character sequence that is a subsequence of this sequence.

`String` **`substring`**`(int start)`
Returns a new `String` that contains a subsequence of characters currently contained in this character sequence.

`String` **`substring`**`(int start, int end)`
Returns a new `String` that contains a subsequence of characters currently contained in this sequence.

`String` **`toString`**`()`
Returns a string representing the data in this sequence.

`void` **`trimToSize`**`()`
Attempts to reduce storage used for the character sequence.

StringBuilder class:

Introduction:

1. The **java.lang.StringBuilder** class is mutable sequence of characters. This provides an API compatible with StringBuffer, but with no guarantee of synchronization.
   Note: **Synchronization** is the capability to control the access of multiple threads to shared resources. **(or)** Synchronization is the mechanism that ensures that only one thread is accessed the resources at a time.

2. The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.
3. Unlike Strings objects of type StringBuffer and Stringbuilder can be modified over and over again without leaving behind a lot of new unused objects.
4. The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe (not Synchronised).
5. It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is **StringBuffer** objects.

StringBuffer	StringBuilder
StringBuffer is Thread safe, so it comes with Performance issue.	StringBuilder is not thread safe(not synchronized).
StringBuffer is synchronized to handle sharing of the object by multiple threads.	you will use StringBuilder if you are using Java 5.0 or later

Class declaration for **java.lang.StringBuilder** class:

```
public final class StringBuilder
                    extends Object
                     implements Serializable, CharSequence
```

Note: The principal operations on a `StringBuilder` are the `append()` and `insert()` methods.

---

1. `StringBuilder` objects are like `String` objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.
2. Strings should always be used unless StringBuilders offer an advantage in terms of simpler code or better performance. For example, if you need to concatenate a large number of strings, appending to a `StringBuilder` object is more efficient.

Length() and Capacity() methods:

1. The `StringBuilder` class, like the `String` class, has a `length()` method that returns the length of the character sequence in the builder.
2. Unlike Strings, every StringBuilder also has a *capacity*, the number of character spaces that have been allocated. The capacity, which is returned by the `capacity()` method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

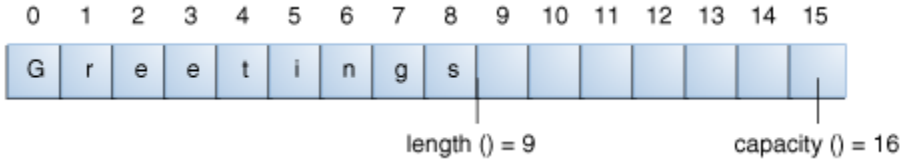
StringBuilder Constructors:

Constructor	Description
StringBuilder()	Creates an empty string builder with a capacity of 16 (16 empty elements).
StringBuilder(CharSequence cs)	Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.
StringBuilder(int initCapacity)	Creates an empty string builder with the specified initial capacity.
StringBuilder(String s)	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

For example:

```
StringBuilder sb = new StringBuilder();// creates empty builder, capacity 16
sb.append("Greetings"); // adds 9 character string at beginning
```

will produce a string builder with a length of 9 and a capacity of 16:



The `StringBuilder` class has some methods related to length and capacity that the `String` class does not have:

Length and Capacity Methods

Method	Description
void setLength(int newLength)	Sets the length of the character sequence. If newLength is less than length(), the last characters in the character sequence are truncated. If newLength is greater than length(), null characters are added at the end of the character sequence.
void ensureCapacity(int minCapacity)	Ensures that the capacity is at least equal to the specified minimum.

StringBuilder Operations:

The principal operations on a `StringBuilder` that are not available in `String` are the `append()` and `insert()` methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder.

The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Note: You can use any String method on a StringBuilder object by first converting the string builder to a string with the toString() method of the StringBuilder class. Then convert the string back into a string builder using the StringBuilder(String str) constructor.

Important methods in StringBuilder class:

1. Java.lang.StringBuilder.append() Method

Description:

The **java.lang.StringBuilder.append(String str)** method appends the specified string to this character sequence.The characters of the String argument are appended, in order, increasing the length of this sequence by the length of the argument.

StringTokenizer Class:

java.lang.Object

Full name: Java.util.StringTokenizer

```
public class StringTokenizer
extends Object
implements Enumeration<Object>
```

1. The StringTokenizer class allows an application to break a String into tokens.
2. The tokenization method is much simpler than the one used by the `StreamTokenizer` class. The `StringTokenizer` methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.
3. The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

Delimiters: a delimiter is a character that identifies the beginning or the end of a character string (a contiguous sequence of characters).

The delimiting character is not part of the character string.

In command syntax, a space or a backslash (\) or a forward slash (/) is often a delimiter, depending on the rules of the command language. The program interpreting the character string knows what the delimiters are.

A **delimiter** is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data streams. An example of a delimiter is the comma character, which acts as a *field delimiter* in a sequence of comma-separated values.

Delimiters can be broken down into:

- Field and record delimiters; and
- Bracket delimiters.

Field and record delimiters

Field delimiters separate data fields. Record delimiters separate groups of fields.
For example, the CSV file format uses a comma as the delimiter between fields, and an end-of-line indicator as the delimiter between records. For instance:

An instance of `StringTokenizer` behaves in one of two ways, depending on whether it was created with the `returnDelims` flag having the value `true` or `false`:

- If the flag is `false`, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.
- If the flag is `true`, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters.

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.
In order to get each token, you just need to loop, until `hasMoreTokens()` return false. Now to get the word itself, just call `nextToken()` method of StringTokenizer. This is similar to Iterating over Java Collection using Iterator, where we use `hasNext()` method as while loop condition and `next()` method to get next element from Collection.
Now `StringTokenizer` will create token if any of this is found in target String.

Method & Description of Stringtokenizer class:

int countTokens()
This method calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
boolean hasMoreElements()
This method returns the same value as the hasMoreTokens method
boolean hasMoreTokens()
This method tests if there are more tokens available from this tokenizer's string.
Object nextElement()
This method returns the same value as the nextToken method, except that its declared return value is Object rather than String.
String nextToken()
This method returns the next token from this string tokenizer.
String nextToken(String delim)
This method returns the next token in this string tokenizer's string.

Differences between StringBuilder, StringBuffer and StringTokenizer

`StringBuffer` - introduced in JDK 1.0 - is thread safe (all of its methods are `synchronized`), StringBuffer is designed to be thread-safe and all public methods in StringBuffer are synchronized.

`StringBuilder` - since JDK 1.5 - is not. Thus it is recommended to use the latter under normal circumstances.

StringBuilder does not handle thread-safety issue and none of its methods is synchronized.

StringBuilder has better performance than StringBuffer under most circumstances.

`StringTokenizer` is meant for a whole different purpose then the former two: cutting strings into pieces, rather than assembling. it is recommended to use `String.split` instead.

StringoTokeizer class allows an application to break a string into tokens.

Note: **`By default StringTokenizer breaks String on space.`**

Topic Name15: Miscellaneous classes:

1) Immutable classes
2) Wrapper Classes(to convert primitive data type to object & vice-versa)
3) Nested classes
4) non-static nested (inner) classes
   a. Anonymous classes
   b. Local classes
   c. Member classes
5) static nested class
6) Abstract class ---------I already covered in keywords and before interfaces.

15.1 Immutable classes:

What is an immutable class?

1. Immutable classes are those classes, whose object can't be modified once created; it means any modification on immutable object will result in another immutable object.
2. Example to understand immutable and mutable objects are, String and StringBuffer. Since String is immutable class, any change on existing string object will result in another string e.g. replacing a character into String, creating substring from String, all result in new objects. While in case of mutable object like StringBuffer, any modification is done on object itself and no new objects are created.

How to write immutable class in Java:

Immutable object still offers several benefits in multi-threaded programming and it's a great choice to achieve thread safety in Java code. Here are few rules, which help to make a class immutable in Java:

1. State of immutable object can't be modified after construction; any modification should result in new immutable object.
2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.
4. Object should be final in order to restrict sub-class for altering immutability of parent class.

Benefits of Immutable Classes in Java:

1. Immutable objects are by default thread safe, can be shared without synchronization in concurrent environment.
2. Immutable object simplifies development, because it's easier to share between multiple threads without external synchronization.
3. Immutable object boost performance of Java application by reducing synchronization in code.
4. Another important benefit of Immutable objects is **reusability**, you can cache Immutable object and reuse them, much like String literals and Integers. You can use static factory methods to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Disadvantage of Immutable object:

1. immutable object has disadvantage of creating garbage as well. Since immutable object can't be reused and they are just a use and throw. String being a prime example, which can create lot of garbage and can potentially slow down application due to heavy garbage collection, but again that's extreme case and if used properly Immutable object adds lot of value.
2.

15.2 Wrapper Class and Type Casting Wrapper classes:

1. Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with **java.io.File**), an address of a system can be seen as an object (with **java.util.URL**), an image can be treated as an object (with **java.awt.Image**) and a simple data type can be converted into an object (with **wrapper classes**).
2. **Wrapper classes** are used to convert any data type into an object.
3. The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, up to JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced **wrapper classes**.

What are Wrapper classes?

1. As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used.
2. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to

prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it.

3. Observe the following conversion.

```
int k = 100;
Integer it1 = new Integer(k);
```

The **int** data type **k** is converted into an object, **it1** using **Integer** class. The **it1** object can be used in Java programming wherever **k** is required an object.

4. The following code can be used to unwrap (getting back **int** from **Integer** object) the object **it1**.

```
int m = it1.intValue();
System.out.println(m); // prints 100
System.out.println(m*m); // prints 10000
```

intValue() is a method of **Integer** class that returns an **int** data type.

List of Wrapper classes:

In the above code, **Integer** class is known as a wrapper class (because it wraps around int data type to give it an impression of object). To wrap (or to convert) each primitive data type, there comes a wrapper class. Eight wrapper classes exist in **java.lang** package that represent 8 data types. Following list gives.

Primitive data type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Following is the hierarchy of the above classes.

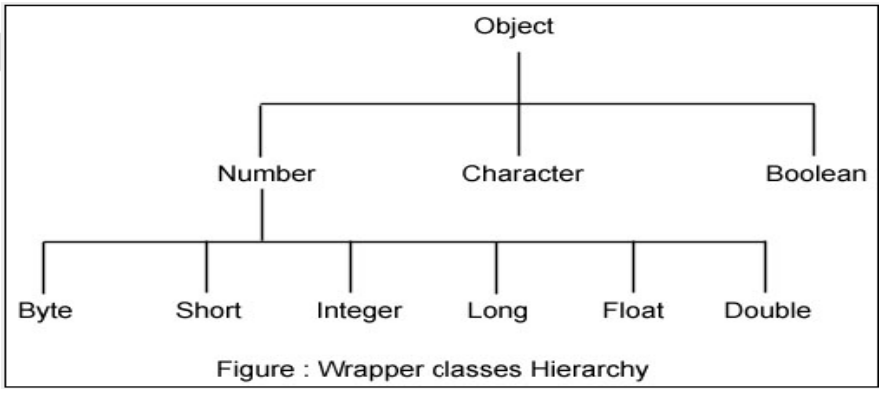


Figure : Wrapper classes Hierarchy

All the 8 wrapper classes are placed in **java.lang** package so that they are implicitly imported and made available to the programmer. As you can observe in the above hierarchy, the super class of all numeric wrapper classes is **Number** and the super class for **Character** and **Boolean** is Object. All the wrapper classes are defined as **final** and thus designers prevented them from inheritance.

Importance of Wrapper classes:

There are mainly two uses with wrapper classes.

1. To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.
2. To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

   This wrapping is taken care of by the compiler, the process is called boxing. So when a primitive is used when an object is required, the compiler boxes the primitive type in its wrapper class. Similarly, the compiler unboxes the object to a primitive as well. The **Number** is part of the java.lang package.

   Here is an example of boxing and unboxing:

```
public class Test{
  public static void main(String args[]){
    Integer x = 5; // boxes int to an Integer object
    x =  x + 10;   // unboxes the Integer to a int
    System.out.println(x);
  }
}
```

   This would produce the following result: 15

Wrapper Class and Type Casting

3. Used to convert primitive values into object type and vice-versa.
4. And most of the utilities classes like ArrayList, HashMap work only on objects, not work with primitive type, but know Jdk1.5 introduce auto boxing so it's not needed to convert primitive type to reference type for utilities class.
5.

Implicit & Explicit Casting

Type Casting: It is use to convert one type into another type.

Primitive Casting:

- Casting lets you convert primitive values from one type to another.
- Casts can be implicit (Widening Conversion) or explicit (Narrowing Conversion)

Primitive Casting (implicit):

Example1:

int a; //take 4 bytes in memory
byte b=20; //take 1 byte in memory
a=b; //Implicit cast (this cast conversion happens automatically , because bigger type can hold smaller type)

Example2:

float b;
int a=90;
b=a;

Primitive Casting (explicit)

Example3:

int d=90;
byte a=(byte) d;
System.out.println(a); //print 90
int d=130;
byte a=(byte)d; //Print -126

Example4:

float a=90.10f;
int b;
b=(int) a; //explicit cast (convert bigger type into smaller type), and programmer has to do explicitly

Note : Wrapper classes are immutable.

Example:

```
int a=100;
Integer i=new Integer(a);
int b=i.intValue();
```

Example:

```
float a=100.10f;
Float r=new Float (a);
float c=r.floatValue();
```

Using Wrapper Methods:

Methods	Descriptions	Example

xxxValue()	Convert Reference Type into Primitive type. and it is not static method.	**Example1:**
		Integer i=new Integer(42);
		int b=i.intValue();
		Example2:
		byte c=i.byteValue();
		float d=i.floatValue();

parseXXX()	• Takes String as an argument and convert it into primitive type
	• Static Method
	• NumberFormatException throws if argument is improper.
		Example1:
		int a=Integer.parseInt("100");
		Example2
		double d=Double.parseDouble("90.10");

toxxxString()	(Binary, Hexadecimal, Octal)	**Example1:** String s=Integer.toHexString(254); String a=Long.toOctalString(254); String b=Integer.toBinaryString(254);

6.

parseInt() Method:

Description: This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.

Syntax:

All the variant of this method are given below:

static int parseInt(String s)

static int parseInt(String s, int radix)

Parameters:

Here is the detail of parameters:

- **s** -- This is a string representation of decimal.
- **radix**(base) -- This would be used to convert String s into integer.

Note: the **radix** or **base** is the number of unique digits, including zero, that a positional numeral system uses to represent numbers. For example, for the decimal system (the most common system in use today) the radix is ten, because it uses the ten digits from 0 through 9.

Radix is a Latin word for "root". *Root* can be considered a synonym for *base* in the arithmetical sense.

In numeral systems:

In the system with radix 13, for example, a string of digits such as 398 denotes the decimal number $3 \times 13^2 + 9 \times 13^1 + 8 \times 13^0$.

Base	Name	Description

Base	Name	Description
10	decimal system	the most used system of numbers in the world, is used in arithmetic. Its ten digits are "0–9". Used in most mechanical counters.
2	binary numeral system	used internally by nearly all computers, is base two. The two digits are "0" and "1", expressed from switches displaying OFF and ON respectively. Used in most electric counters.
16	hexadecimal system	is often used in computing. The sixteen digits are "0–9" followed by "A–F".
8	octal system	is occasionally used in computing. The eight digits are "0–7".

Return Value:

- **parseInt(String s):** This returns an integer (decimal only).
- **parseInt(int i):** This returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix(base) equals 10(decimal system), 2(binary numerical system), 8(octal system), or 16(hexadecimal system) respectively) numbers as input.

AutoBoxing:

- Java 5 supports automatic conversion of primitive types (int, float, double etc.) to their object equivalents (Integer, Float, Double,...) in assignments and method and constructor invocations. This conversion is known as autoboxing.
- Java 5 also supports automatic unboxing, where wrapper types are automatically converted into their primitive equivalents if needed for assignments or method or constructor invocations.

Example:

```
int i = 0;
i = new Integer(5); // auto-unboxing
Integer i2 = 5; // autoboxing
```

15.3 Nested classes in Java:

1. A **class declared inside a class** is known as nested class.

2. We use nested classes to logically group classes and interfaces in one place so that it can be more readable and maintainable code.
Additionally, it can access all the members of outer class including private data members and methods.

Syntax:

```
class Outer_class_Name{
 ...
        class Nested_class_Name{
 ...
```

```
        }
    ...
}
```

Advantages:

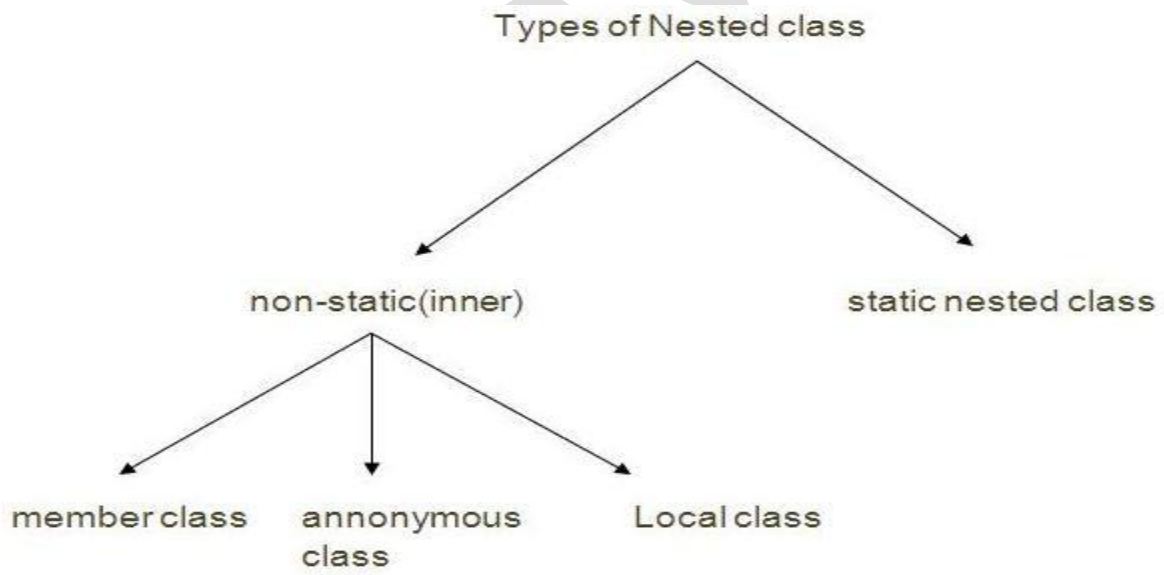
- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

Note: Inner class is a part of nested class.
Non-static nested classes are known as inner classes.

Types of nested classes:

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

1. non-static nested class(inner class)
   a) Member class
   b) Anonymous class
   c) Local class
2. static nested class



Inner class:

Inner class and nested static class in Java both are classes declared inside another class, known as top level class in Java.

Inner Class in java so called nested class is nothing but a class that is being defined inside another class.

inner class in Java:

An normal inner class can be created in any class, with any type of access modifier (public, private, protected & default).

```
[modifier] class OuterClass {
    [modifier] class innerClass {
  }
}
```

1. We can create a long hierarchy of inner classes as long as we want to:

```
private class OuterClass {
   public class InnerClassA {
     public class InnerClassB {
      }
   }
 }
```

2. Outer class can create as many numbers of instances of inner class inside its code.
3. No inner class object is automatically instantiated with creation of outer class's object.
4.

The two different ways to create inner class instances, and also discuss how they are different:

- If you want to create an inner class instance from inside the outer class code, then you can instantiate the class using the normal syntax:

```
InnerClass inClass = new InnerClass();
```

- If you want to create an inner class code outside of the outer class code, or inside a static method that is still a member of the outer class, then you must use a reference to the outer class, like this:
- `OuterClass.InnerClass inner = new OuterClass().new InnerClass();`

  Or you can use the longer syntax like this:

```
OuterClass outClass = new OuterClass();

OuterClass.InnerClass inner = outClass.new InnerClass();
```

Nested Static Class in Java:

Nested static class look exactly similar to member inner classes but has quite a few significant difference with them, e.g. you can access them inside main method because they are static. In order to create instance of nested static class, you don't need instance of enclosing class. You can refer them with class name and you can also import them using static import feature of Java 5.

Member inner class

A class that is declared inside a class but outside a method is known as member inner class.

Invocation of Member Inner class;

1. From within the class

2. From outside the class

Example1: member inner class that is invoked inside a class

In this example, we are invoking the method of member inner class from the display method of Outer class.

```
package com.java.allclasses;
//we are invoking the method of member class from the display method of Outer class.
```

```
package com.java.allclasses;
//we are invoking the method of member class from the display method of Outer class.
public class MemberClass {

    private int data = 30;

    class Inner {
        void msg() {
            System.out.println("data is: " + data);
        }
    }

    void display() {
        Inner in = new Inner();
        in.msg();
    }

    public static void main(String args[]) {
        MemberClass obj = new MemberClass();
        obj.display();
    }

}
// output: data is:30
```

Example2: invoking the msg() method of Inner class from outside the outer class i.e. MemberClass1 class.

```
package com.java.allclasses;
//we are invoking the msg() method of Inner class from outside the outer class i.e. MemberClass1 class.
class Outer {
    private int data = 30;

    class Inner {
        void msg() {
            System.out.println("data is: " + data);
        }
    }
}

class MemberClass1 {
    public static void main(String args[]) {
        Outer obj = new Outer();
        Outer.Inner in = obj.new Inner();
        in.msg();
    }

}
```

Anonymous inner class

A class that have no name is known as anonymous inner class.

Anonymous class can be created by:

1. Class (may be abstract class also).
2. Interface

Note:

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type. As you know well that Parent class reference variable can refer the object of Child class.
3. If we want to define anonymous class then we need to create /declare super class.
4. If we want to use any method in anonymous class then also we must define or we must override that method in super class. i.e without overriding the method in super class, we can't use in anonymous inner class.
5. In general, we will do separately both declaration of the class and instance creation but in we will do both at a time in anonymous inner class.
6.

Local inner class:

A class that is created inside a method is known as local inner class.
If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Topic Name16: Exception Handling:

Exception Handling:

1. An *exception* is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
2. **Exceptions in java** are any abnormal, unexpected events or extraordinary conditions that may occur at runtime. They could be file not found exception, unable to get connection exception and so on. On such conditions java throws an exception object.
3. Java Exceptions are basically Java objects.
4. No Project can ever escape a java error exception.
5. Imp: Exception can arise from different kind of situations such as wrong data entered by user, hardware failure,
   network connection failure,
   Database server down etc.
6. Java exception handling is used to handle error conditions in a program systematically by taking the necessary action. Exception handlers can be written to catch a specific exception such as Number Format exception or an entire group of exceptions by using a generic exception handlers. Any exceptions not specifically handled within a Java program are caught by the Java run time environment.
7. An exception is a subclass of the Exception/Error class, both of which are subclasses of the Throwable class.

8. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
9. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

Java Exception Handling:

Java being an object oriented programming language, whenever an error occurs while executing a statement, creates an **exception object** and then the normal flow of the program halts and JRE tries to find someone that can handle the raised exception. The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception etc. When the exception occurs in a method, the process of creating the exception object and handing it over to runtime environment is called **"throwing the exception"**.

Once runtime receives the exception object, it tries to find the handler for the exception. Exception Handler is the block of code that can process the exception object. The logic to find the exception handler is simple – starting the search in the method where error occurred, if no appropriate handler found, then move to the caller method and so on. So if methods call stack is A->B->C and exception is raised in method C, then the search for appropriate handler will move from C->B->A. If appropriate exception handler is found, exception object is passed to the handler to process it. The handler is said to be **"catching the exception"**. If there are no appropriate exception handler found then program terminates printing information about the exception.
We use specific keywords in java program to create an exception handler block.

Exception Handling 2 Keywords and 3 Blocks:

Java provides specific keywords for exception handling purposes, we will look after them first and then we will write a simple program showing how to use them for exception handling.

1. **throw** – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. **throw** keyword is used to throw exception to the runtime to handle it.
2. **throws** – When we are throwing any exception in a method and not handling it, then we need to use **throws** keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it's caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.
3. **try-catch** – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions.
   We can have multiple catch blocks with a try and

try-catch block can be nested also.

catch block requires a parameter that should be of type Exception.

4. **finally** – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

Imp points:

1. We can't have catch or finally block without a try statement.
2. A try statement should have either catch block or finally block, it can have both blocks.
3. We can't write any code between try-catch-finally block.
4. We can have multiple catch blocks with a single try statement.
5. try-catch blocks can be nested similar to if-else statements.
6. We can have only one finally block with a try-catch statement.

Types of Exception:

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun Microsystems says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

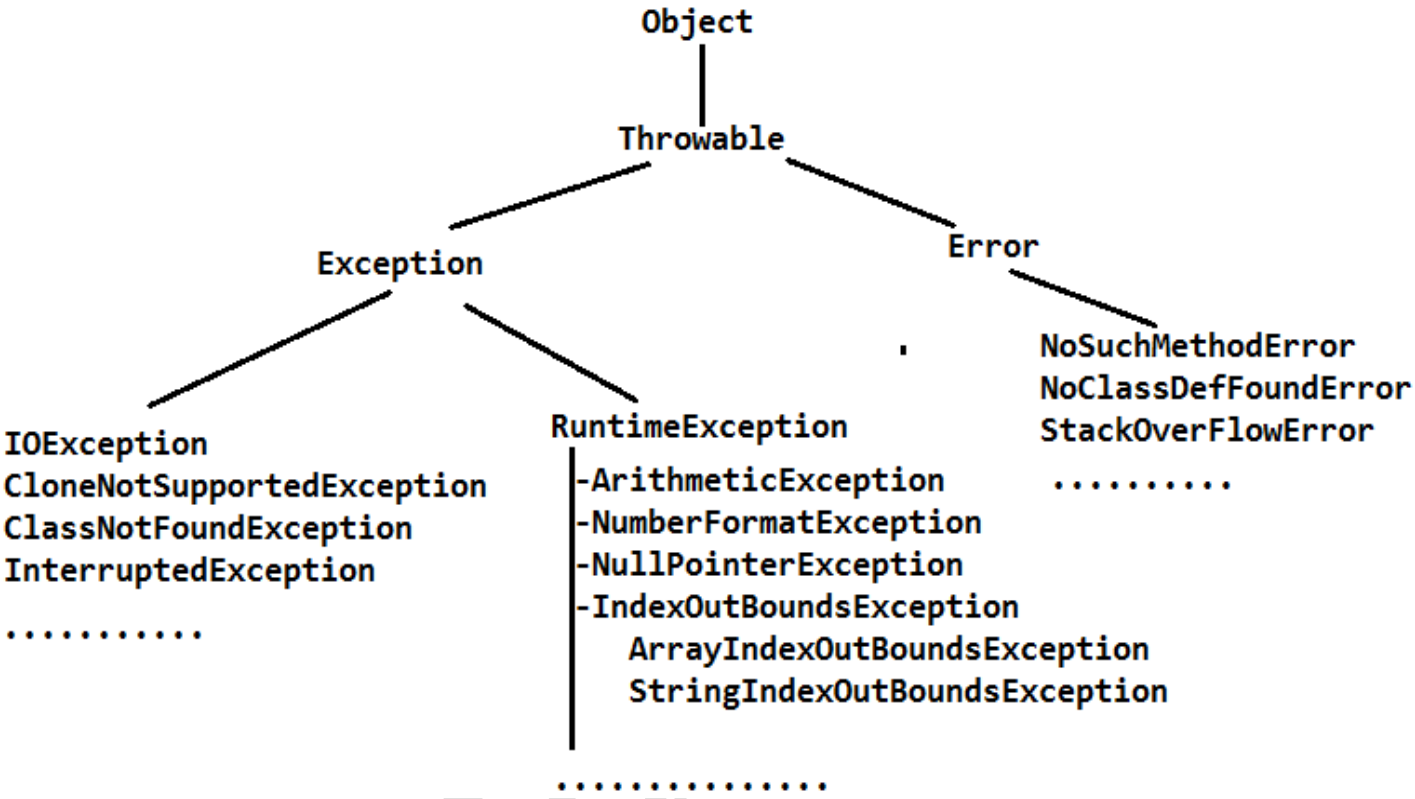
2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, NoSuchMethodError, StackOverFlowError etc.

Exception Hierarchies:



Useful Exception Methods

Exception and all of its subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable. The exception classes are created to specify different kind of exception scenarios so that we can easily identify the root cause and handle the exception according to it's type. Throwable class implements Serializable interface for interoperability.

Some of the useful methods of Throwable class are;

1. **public String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.
2. **public String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use `getMessage()` method to return the exception message.

3. **public synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
4. **public String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
5. **public void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

Summary of Exception Handling:

A program can use exceptions to indicate that an error occurred. To throw an exception, use the `throw` statement and provide it with an exception object — a descendant of `Throwable` — to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a `throws` clause in its declaration.

A program can catch exceptions by using a combination of the `try`, `catch`, and `finally` blocks.

- The `try` block identifies a block of code in which an exception can occur.
- The `catch` block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The `finally` block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the `try` block.

The `try` statement should contain at least one `catch` block or a `finally` block and may have multiple `catch` blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message.

Common scenarios of Exception Handling where exceptions may occur:

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Example program for Exception Handling:

```
package com.bvn.corejava.exceptions;

public class Test {

	public static void main(String args[]){

/*		System.out.println("I am in main method First line");
		System.out.println("a/b");
		System.out.println("10/5");
		System.out.println(10/2);
*/
		try{
		 String s1 = args[0];
		 String s2 = args[1]; // if 10 _(no value) then we will get AIOOBE

	int a = Integer.parseInt(s1);
	int b = Integer.parseInt(s2);//    if we apply 10 two then we get NumberFormatException

		 // only try not exists alone
			//System.out.println(10/0);
			//System.out.println(10/1);
			System.out.println(a/b);// if 10 20 then no exception

		}catch(ArithmeticException ae){
			System.out.println(ae);
			ae.printStackTrace();

		 }catch(ArrayIndexOutOfBoundsException aiobe){// catch not exists without try
		System.out.println(aiobe);

		 }catch(NumberFormatException nfe){// catch not exists without try
		System.out.println(nfe);
		 }catch(Exception e){// catch not exists without try
		System.out.println(e);
		 }
		 finally{ // finally exists without catch block and finally block optional, finally not exists w/o try block
			System.out.println("I am in main method Last line");
		 }

	}
}
```

Topic Name17: Collections Framework:

Introduction to Collections:

1. A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
2. Collections are used to store, retrieve, manipulate and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.
3. Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.

The collections framework was designed to meet several goals.

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic Arrays, LinkedLists, trees and Hashtables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Extending and/or adapting a collection had to be easy.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections.

All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be

polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

Imp: 1. A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.
2. The classes and interfaces of the collections framework are in package java.util.

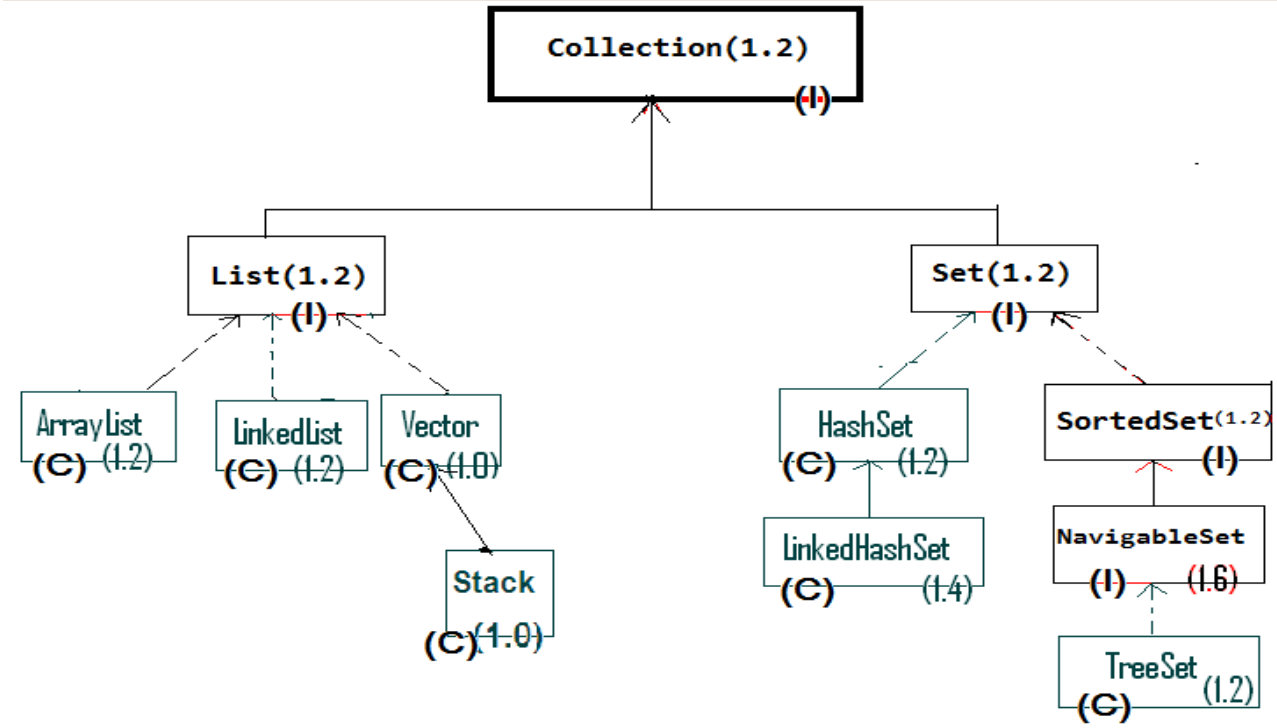
Benefits of the Java Collections Framework:

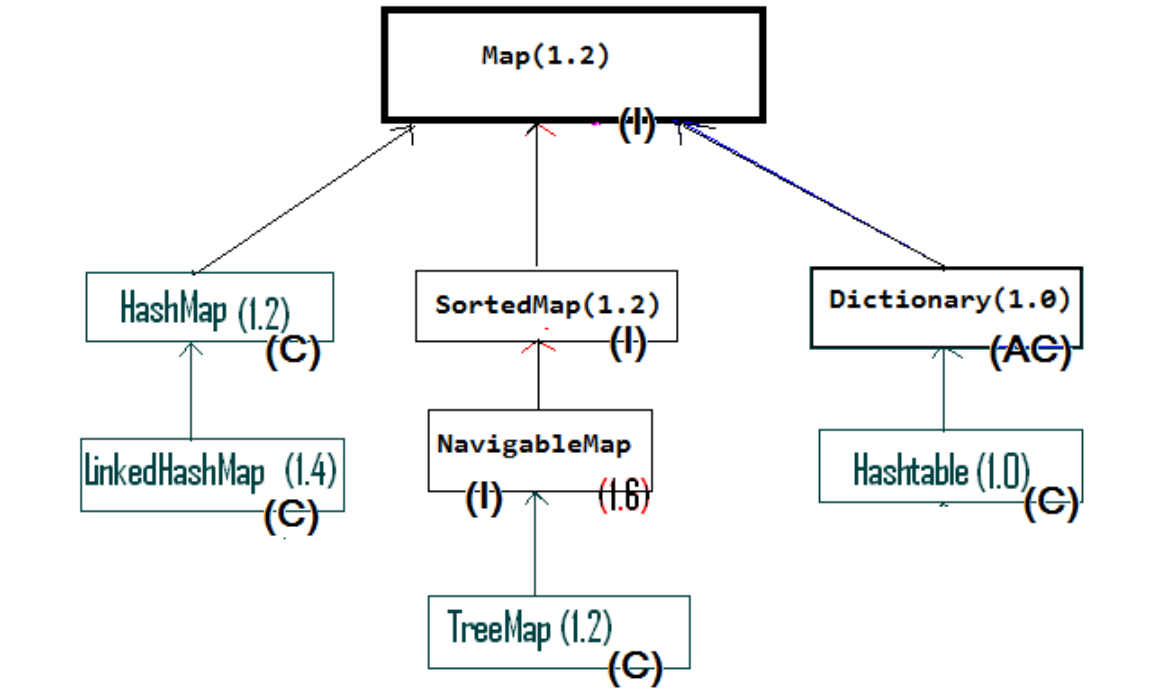
The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection Interfaces:

- The core collection interfaces encapsulate different types of collections and are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. The core collection interfaces are the foundation of the Java Collections Framework.


Collection(1.2)
(I)
List(1.2)
(I)
Set(1.2)
(I)
ArrayList
(C) (1.2)
LinkedList
(C) (1.2)
Vector
(C) (1.0)
Stack
(C) (1.0)
HashSet
(C) (1.2)
LinkedHashSet
(C) (1.4)
SortedSet(1.2)
(I)
NavigableSet
(I) (1.6)
TreeSet
(C) (1.2)


Map(1.2)
(I)
HashMap (1.2)
(C)
SortedMap(1.2)
(I)
Dictionary(1.0)
(AC)
LinkedHashMap (1.4)
(C)
NavigableMap
(I) (1.6)
Hashtable (1.0)
(C)
TreeMap (1.2)
(C)

Collection — the root of the collection hierarchy.

- A collection represents a group of objects known as its *elements*.
- The `Collection` interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired.
- **Some types of collections allow duplicate elements, and others do not**.
- Some are ordered and others are unordered.
- **The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub interfaces, such as `Set` and `List`.**
- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.

Map — an object that maps keys to values.

A `Map` cannot contain duplicate keys; each key can map to at most one value. If you've used `Hashtable`.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

More Info:

The Collection Interfaces:

The collections framework defines several interfaces. This section provides an overview of each interface:

Interface Summary with Description	
Collection	The root interface in the *collection hierarchy*. Collection Interface: The enables you to work with groups of objects; it is at the top of the collections hierarchy.
Comparator	A comparison function, which imposes a *total ordering* on some collection of objects.
Enumeration	The Enumeration: This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. (or) An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Iterator	An iterator over a collection.
List	An ordered collection (also known as a *sequence*). List Interface: This extends **Collection** and an instance of List stores an ordered collection of elements.
ListIterator	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in

	the list.
Map	Map: This maps unique keys to values. An object that maps keys to values.
Map.Entry	Map.Entry: This describes an element (a key/value pair) in a map. This is an inner class of Map. A map entry (key-value pair).
Observer	A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.
RandomAccess	Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access.
Set	A collection that contains no duplicate elements. Set: This extends Collection to handle sets, which must contain unique elements
SortedMap	SortedMap: This extends Map so that the keys are maintained in ascending order. A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time.
SortedSet	SortedSet: This extends Set to handle sorted sets A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

The Collection Classes:

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as it is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.
The standard collection classes are summarized in the following table:

Class Summary	
AbstractCollection	**AbstractCollection:** Implements most of the Collection interface. This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.
AbstractList	**AbstractList:** Extends AbstractCollection and implements most of the List interface. This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
AbstractMap	**AbstractMap** Implements most of the Map interface. This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface.
AbstractSequentialList	**AbstractSequentialList:** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
AbstractSet	**AbstractSet** Extends AbstractCollection and implements most of the Set interface. This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.
ArrayList	ArrayList: Implements a dynamic array by extending AbstractList. Resizable-array implementation of the `List` interface.
Arrays	This class contains various methods for manipulating arrays (such as

	sorting and searching).
BitSet	This class implements a vector of bits that grows as needed.
Calendar	Calendar is an abstract base class for converting between a Date object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on.
Collections	This class consists exclusively of static methods that operate on or return collections.
Currency	Represents a currency.
Date	The class Date represents a specific instant in time, with millisecond precision.
Dictionary	The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
EventListenerProxy	An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
EventObject	The root class from which all event state objects shall be derived.
GregorianCalendar	GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar used by most of the world.
HashMap	HashMap Extends AbstractMap to use a hash table. Hash table based implementation of the Map interface.
HashSet	HashSet Extends AbstractSet for use with a hash table. This class implements the Set interface, backed by a hash table (actually a HashMap instance).
Hashtable	This class implements a hashtable, which maps keys to values.
IdentityHashMap	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents. This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
LinkedHashMap	LinkedHashMap Extends HashMap to allow insertion-order iterations. Hash table and linked list implementation of the Map interface, with predictable iteration order.
LinkedHashSet	LinkedHashSet Extends HashSet to allow insertion-order iterations. Hash table and linked list implementation of the Set interface, with predictable iteration order.
LinkedList	LinkedList: Implements a linked list by extending AbstractSequentialList. Linked list implementation of the List interface.
ListResourceBundle	ListResourceBundle is an abstract subclass of ResourceBundle that manages resources for a locale in a convenient and easy to use list.
Locale	A Locale object represents a specific geographical, political, or cultural region.
Observable	This class represents an observable object, or "data" in the model-view paradigm.
Properties	The Properties class represents a persistent set of properties.

PropertyPermission	This class is for property permissions.
PropertyResourceBundle	`PropertyResourceBundle` is a concrete subclass of `ResourceBundle` that manages resources for a locale using a set of static strings from a property file.
Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
SimpleTimeZone	`SimpleTimeZone` is a concrete subclass of `TimeZone` that represents a time zone for use with a Gregorian calendar.
Stack	The `Stack` class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thread.
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	`TimeZone` represents a time zone offset, and also figures out daylight savings.
TreeMap	TreeMap Extends AbstractMap to use a tree. Red-Black tree based implementation of the `SortedMap` interface.
TreeSet	TreeSet Implements a set stored in a tree. Extends AbstractSet. This class implements the `Set` interface, backed by a `TreeMap` instance.
Vector	The `Vector` class implements a growable array of objects.
WeakHashMap	WeakHashMap Extends AbstractMap to use a hash table with weak keys. A Hashtable-based `Map` implementation with *weak keys*.

The following legacy classes defined by java.util:

SN	Classes with Description
1	Vector This implements a dynamic array. It is similar to ArrayList, but with some differences.
2	Stack Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3	Dictionary Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4	Hashtable Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5	Properties Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6	BitSet A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

All the Synchronized and Non-Synchronized Collection classes:

- Most of the popular collection classes have implementations for both single thread and multiple thread environments. The non-synchronized implementations are always faster. You can use the non-synchronized implementations in multiple thread environments, when you make sure that only one thread updating the collection at any given time.
- A new Java JDK package was introduced at Java 1.5 that is java.util.concurrent. This package supplies a few Collection implementations designed for use in multithreaded environments.

The following table list all the synchronized & non-synchronized collection classes:

Interface	Syncronized	Non-Syncronized
List	java.util.Vector java.util.Stack java.util.concurrent.CopyOnWriteArrayList	java.util.ArrayList java.util.LinkedList
Set	java.util.concurrent.CopyOnWriteArraySet	java.util.TreeSet java.util.HashSet java.util.LinkHashSet
Map	java.util.Hashtable java.util.concurrent.ConcurrentHashMap	java.util.TreeMap java.util.HashMap java.util.LinkedHashMap java.util.IdentityHashMap java.util.EnumMap

17.1 List(I):

List(I):

1) ArrayList(C):
2) Conversion of collection classes and interfaces:
3) Vector(C):
4) LinkedList(C):

I. List Interface (Interface java.util.List):

public interface **List** extends Collection

1. An ordered collection (also known as a *sequence*).
2. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
3. Unlike sets(set is an interface), lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

SN	Methods with Description
1	**void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	**Object get(int index)** Returns the object stored at the specified index within the invoking collection.
4	**int indexOf(Object obj)** Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
5	**int lastIndexOf(Object obj)** Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
6	**ListIterator listIterator( )** Returns an iterator to the start of the invoking list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.
8	**Object remove(int index)** Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	**Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list.
10	**List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also

	referenced by the invoking object.

Imp points:

The List interface extends **Collection** and declares the behaviour of a collection that stores a sequence of elements.

- The `java.util.List` interface is a subtype of the `java.util.Collection` interface.
- It is an growable Array Data Structure.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- It represents an ordered list of objects, meaning you can access the elements of a `List` in a specific order, and by an index too. You can also add the same element more than once to a `List`.
- A list may contain duplicate elements.
- A list accepts null values and also accepts more than one null value because it accepts duplicates.
- In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following below Table.
- The List interface has been implemented in various classes like ArrayList or LinkedList, etc.
- Several of the list methods will throw an UnsupportedOperationException if the collection cannot be modified, and a ClassCastException is generated when one object is incompatible with another.

List Implementations:

Being a `Collection` subtype all methods in the `Collection` interface are also available in the `List` interface.

Since `List` is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following `List` implementations in the Java Collections API:

java.util.ArrayList
java.util.LinkedList
java.util.Vector
java.util.Stack

Generic Lists

By default you can put any `Object` into a `List`, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a `List`. Here is an example:

```
List<MyObject> list = new ArrayList<MyObject>();
```

This `List` can now only have `MyObject` instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

```
MyObject myObject = list.get(0);

for(MyObject anObject : list){
   //do someting to anObject...
}
```

1. ArrayList Class(Class java.util.ArrayList**):**

```
java.lang.Object
        java.util.AbstractCollection
                java.util.AbstractList
                        java.util.ArrayList
```

public class **ArrayList** implements List, Cloneable, Serializable extends AbstractList

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`.

In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added an ArrayList, its capacity grows automatically.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException.

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

The ArrayList class supports three constructors. The first constructor builds an empty array list.

```
ArrayList( )
```

The following constructor builds an array list that is initialized with the elements of the collection c.

```
ArrayList(Collection c)
```

The following constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements.

The capacity grows automatically as elements are added to an array list.

```
ArrayList(int capacity)
```

Apart from the methods inherited from its parent classes, ArrayList defines following methods:

SN	Methods with 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 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 clear()** Removes all of the elements from this list.
6	**Object clone()** Returns a shallow copy of this ArrayList.
7	**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)).
8	**void ensureCapacity(int minCapacity)** Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
9	**Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
10	**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.
11	**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.

12	**Object remove(int index)** Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if index out of range (index < 0 || index >= size()).
13	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
14	**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 is out of range (index < 0 || index >= size()).
15	**int size()** Returns the number of elements in this list.
16	**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.
17	**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.
18	**void trimToSize()** Trims the capacity of this ArrayList instance to be the list's current size.

2. LinkedList Class(Class java.util.LinkedList**):**

```
java.lang.Object
        java.util.AbstractCollection
                java.util.AbstractList
                        java.util.AbstractSequentialList
                                java.util.LinkedList
```

public class **LinkedList**
implements List, Cloneable, Serializable
extends AbstractSequentialList

Linked list implementation of the `List` interface. Implements all optional list operations, and permits all elements (including `null`). In addition to implementing the `List` interface, the `LinkedList` class provides uniformly named methods to `get`, `remove` and `insert` an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque).

All of the stack/queue/deque operations could be easily recast in terms of the standard list operations. They're included here primarily for convenience, though they may run slightly faster than the equivalent List operations.

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the begining or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

The iterators returned by the this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`.

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

The LinkedList class supports two constructors. The first constructor builds an empty linked list:

```
LinkedList( )
```

The following constructor builds a linked list that is initialized with the elements of the collection c.

```
LinkedList(Collection c)
```

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

SN	Methods with 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 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 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 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 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 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.

2. What are differences between LinkedList and ArrayList?

ArrayList is more popular among Java programmer than LinkedList as there are few scenarios on which LinkedList is a suitable collection than ArrayList.

All the differences between LinkedList and ArrayList have there root on difference between Array and LinkedList data-structure. If you are familiar with Array and LinkedList data structure you will most likely derive following differences between them:

LinkedList	ArrayList
LinkedList and ArrayList both implement List Interface	LinkedList and ArrayList both implement List Interface
ArrayList is implemented using re sizable array while LinkedList is implemented using doubly LinkedList.	ArrayList is implemented using re sizable array while LinkedList is implemented using doubly LinkedList.
1) Since Array is an index based data-structure searching or getting element from Array with index is pretty fast. Array provides O(1) performance forget(index) method but remove is costly in ArrayList as you need to rearrange all elements. On the Other hand LinkedList doesn't provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n).	1) Since Array is an index based data-structure searching or getting element from Array with index is pretty fast. Array provides O(1) performance forget(index) method but remove is costly in ArrayList as you need to rearrange all elements. On the Other hand LinkedList doesn't provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n).
2) Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java. ArrayList also needs to update its index if you insert something anywhere except at the end of array.	2) Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java. ArrayList also needs to update its index if you insert something anywhere except at the end of array.
3) Removal is like insertions better in LinkedList than ArrayList.	3) Removal is like insertions better in LinkedList than ArrayList.
4) LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.	4) LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

3. When to use LinkedList and ArrayList in Java?

As I said LinkedList is not as popular as ArrayList but still there are situation where a LinkedList is better choice than ArrayList in Java.

Use LinkedList in Java if:

1) Your application can live without Random access. Because if you need nth element in LinkedList you need to first traverse up to nth element O(n) and then you get data from that node.

2) Your application is more insertion and deletion driver and you insert or remove more than retrieval. Since insertion or removal doesn't involve resizing it's much faster than ArrayList.

Use ArrayList in Java for their entire situation where you need a **non-synchronized index based access**. ArrayList is fast and easy to use, just try to minimize array resizing by constructing arraylist with proper initial size.

3. Vector Class(Class java.util.Vector):

```
java.lang.Object
        java.util.AbstractCollection
                java.util.AbstractList
                        java.util.Vector
```

public class **Vector**
implements List, Cloneable, Serializable
extends AbstractList

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

As of the Java 2 platform v1.2, this class has been retrofitted to implement List, so that it becomes a part of Java's collection framework. Unlike the new collection implementations, Vector is synchronized.

The Iterators returned by Vector's iterator and listIterator methods are *fail-fast*: if the Vector is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException.

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

The Vector class supports four constructors. The first form creates a default vector, which has an initial size of 10:

```
Vector( )
```

The second form creates a vector whose initial capacity is specified by size:

```
Vector(int size)
```

The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward:

```
Vector(int size, int incr)
```

The fourth form creates a vector that contains the elements of collection c:

```
Vector(Collection c)
```

Apart from the methods inherited from its parent classes, Vector defines the following methods:

SN	Methods with Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this Vector.
8	**Object clone()** Returns a clone of this vector.
9	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)**

	Returns true if this Vector contains all of the elements in the specified Collection.
11	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
12	**Object elementAt(int index)** Returns the component at the specified index.
13	**Enumeration elements()** Returns an enumeration of the components of this vector.
14	**void ensureCapacity(int minCapacity)** 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)** Compares the specified Object with this Vector for equality.
16	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
17	**Object get(int index)** Returns the element at the specified position in this Vector.
18	**int hashCode()** Returns the hash code value for this Vector.
19	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
20	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.
22	**boolean isEmpty()** Tests if this vector has no components.
23	**Object lastElement()** Returns the last component of the vector.
24	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.
25	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	**Object remove(int index)** Removes the element at the specified position in this Vector.
27	**boolean remove(Object o)**

	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)** Removes from this Vector all of its elements that are contained in the specified Collection.
29	**void removeAllElements()** Removes all components from this vector and sets its size to zero.
30	**boolean removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	**void removeElementAt(int index)** removeElementAt(int index)
32	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	**boolean retainAll(Collection c)** Retains only the elements in this Vector that are contained in the specified Collection.
34	**Object set(int index, Object element)** Replaces the element at the specified position in this Vector with the specified element.
35	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
36	**void setSize(int newSize)** Sets the size of this vector.
37	**int size()** Returns the number of components in this vector.
38	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39	**Object[] toArray()** Returns an array containing all of the elements in this Vector in the correct order.
40	**Object[] toArray(Object[] a)** 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()** Returns a string representation of this Vector, containing the String representation of each element.
42	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

Differences between a Vector and an Array

- A vector is a dynamic array, whose size can be increased, where as an array size can't be changed.
- Reserve space can be given for vector, where as for arrays can't.
- A vector is a class where as an array is not.
- Vectors can store any type of objects, where as an array can store only homogeneous values.

- Vector is a growable and shrinkable where as Array is not.
- Vector implements the List interface where as array is a primitive data type
- Vector is synchronized where as array is not.
- The size of the array is established when the array is created. As the Vector is growable, the size changes when it grows.

Advantages and disadvantages of Vector and Array:

Advantages of Arrays:
- Arrays supports efficient random access to the members.
- It is easy to sort an array.
- They are more appropriate for storing fixed number of elements

Disadvantages of Arrays:
- Elements can't be deleted
- Dynamic creation of arrays is not possible
- Multiple data types can't be stored

Advantages of Vector:
- Size of the vector can be changed
- Multiple objects can be stored
- Elements can be deleted from a vector

Disadvantages of Vector:
- A vector is an object , memory consumption is more.

Differences between Vector and Array.

Advantages and Disadvantages of Vector and Array:

Arrays provide efficient access to any element and can not modify or increase the size of the array.

Vector is efficient in insertion, deletion and to increase the size.

Arrays size is fixed where as Vector size can increase.

Elements in the array can not be deleted, where as a Vector can.

---

1. What are the similarities between ArrayList and Vector on certain scenarios?
`ArrayList` and `Vector` are two of most used class on java collection package.
By the way Java 5 adds another implementation of `List` interface which is similar to Vector and ArrayList but provides better concurrency access than Vector, its called CopyOnWriteArrayList.
Let's see some similarities between these two and why we can use ArrayList in place of Vector on certain scenario.
1) `ArrayList` and `Vector` are index based and backed up by an array internally.
2) Both ArrayList and Vector maintains the insertion order of element. Means you can assume that you will get the object in the order you have inserted if you iterate over ArrayList or Vector.
3) Both Iterator and ListIterator returned by *ArrayList and Vector* are fail-fast.
4) `ArrayList` and `Vector` also allows null and duplicates.

2. Differences between Arraylist and Vector?
Now let's see some key difference between `Vector` and `ArrayList` in Java**,** this will decide when is the right time to use Vector over ArrayList and vice-versa. Differences are based upon properties like synchronization, thread safety, speed, performance, navigation and Iteration over List etc.

ArrayList	**Vector**
1) Synchronization and thread-safety: First and foremost difference between Vector and ArrayList is that **Vector is synchronized** and **ArrayList is not**, What it means is that all the method which structurally	**1) Synchronization and thread-safety:** First and foremost difference between Vector and ArrayList is that **Vector is synchronized** and **ArrayList is not**, What it means is that all the method which structurally

modifies Vector e.g. add () or remove () are synchronized which makes it thread-safe and allows it to be used safely in a multi-threaded and concurrent environment. On the other hand ArrayList methods are not synchronized thus not suitable for use in multi-threaded environment. This is also a popular interview question on thread, where people ask why ArrayList can't be shared between multiple threads.	modifies Vector e.g. add () or remove () are synchronized which makes it thread-safe and allows it to be used safely in a multi-threaded and concurrent environment. On the other hand ArrayList methods are not synchronized thus not suitable for use in multi-threaded environment. This is also a popular interview question on thread, where people ask why ArrayList can't be shared between multiple threads.
2) Speed and Performance `ArrayList` is way faster than `Vector`. Since Vector is synchronized and thread-safe it pays price of synchronization which makes it little slow. On the other hand `ArrayList` is not synchronized and fast which makes it obvious choice in a single-threaded access environment. You can also use `ArrayList` in a multi-threaded environment if multiple threads are only reading values from ArrayList or you can create read only ArrayList as well.	**2) Speed and Performance** `ArrayList` is way faster than `Vector`. Since Vector is synchronized and thread-safe it pays price of synchronization which makes it little slow. On the other hand `ArrayList` is not synchronized and fast which makes it obvious choice in a single-threaded access environment. You can also use `ArrayList` in a multi-threaded environment if multiple threads are only reading values from ArrayList or you can create read only ArrayList as well.
3) Capacity Whenever Vector crossed the threshold specified it increases itself by value specified in `capacityIncrement` field while you can increase size of ArrayList by calling `ensureCapacity ()` method.	**3) Capacity** Whenever Vector crossed the threshold specified it increases itself by value specified in `capacityIncrement` field while you can increase size of ArrayList by calling `ensureCapacity ()` method.
4) Enumeration and Iterator Vector can return enumeration of items it hold by calling elements () method which is not *fail-fast* as opposed to Iterator and ListIterator returned by ArrayList.	**4) Enumeration and Iterator** Vector can return enumeration of items it hold by calling elements () method which is not *fail-fast* as opposed to Iterator and ListIterator returned by ArrayList.
5) Legacy Another point worth to remember is Vector is one of those classes which comes with JDK 1.0 and initially not part of Collection framework but in later version it's been re-factored to implement List interface so that it could become part of collection framework	**5) Legacy** Another point worth to remember is Vector is one of those classes which comes with JDK 1.0 and initially not part of Collection framework but in later version it's been re-factored to implement List interface so that it could become part of collection framework

4. Stack Class (Class java.util.Stack):

```
java.lang.Object
        java.util.AbstractCollection
                java.util.AbstractList
                        java.util.Vector
                                java.util.Stack
```

public class **Stack**
extends Vector

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual `push` and `pop` operations are provided, as well as a method to `peek` at the top item on the stack, a method to test for whether the stack is `empty`, and a method to `search` the stack for an item and discover how far it is from the top.

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines following methods:

SN	Methods with Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes element onto the stack. element is also returned.
5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

****

17.2 Set(I)

Set(I):

1) HashSet(C):
2) TreeSet(C):
3) LinkedHashSet(C):

II. Set Interface (Interface java.util.Set):

public interface **Set** extends Collection

The `java.util.Set` **interface** is a subtype of the `java.util.Collection` interface. It represents set of objects, meaning each element can only exist once in a `Set`.

Example:

Here is first a **simple Java** `Set` example to give you a feel for how sets work:

```
Set s = new HashSet();
String element = "element 1";
```

```
s.add(element);
System.out.println(set.contains(element) );
```

This example creates a `HashSet` which is one of the **classes** in the Java APIs that implement the `Set` interface. Then it adds a string object to the set, and finally it **checks** if the set contains the element just added.

`Set` doesn't allow any duplicate. If you insert **duplicate** in `Set` it will replace the older value. Any implementation of `Set` in Java will only contains unique elements.

`Set` is an unordered Collection.

`Set` in Java doesn't maintain any order. Though `Set` provide another alternative called `SortedSet` which can store `Set` elements in specific Sorting order defined by Comparable and Comparator methods of Objects stored in `Set`.

`Set` uses equals() method to check uniqueness of elements stored in `Set`, while `SortedSet` uses `compareTo()` method to implement natural sorting order of elements. In order for an element to behave properly in `Set` and `SortedSet`, equals and compareTo must be consistent to each other.

Popular implementation of `Set` interface includes `HashSet`, `TreeSet` and **`LinkedHashSet`**.

If you need to maintain insertion order or object and you collection can contain duplicates than `List` is a way to go. On the other hand **if your requirement is to maintain unique collection without any duplicates than `Set` is the way to go.**

A Set is a Collection that cannot contain duplicate elements.
The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
Set also adds a stronger contract on the behaviour of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.
Two `Set` instances are equal if they contain the same elements.

The **Java** platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. `HashSet`, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. `TreeSet`, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. `LinkedHashSet`, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic **ordering** provided by `HashSet` at a cost that is only slightly higher.

Here's a simple but useful `Set` idiom. Suppose you have a `Collection`, `c`, and you want to create another `Collection` containing the same

elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a `Set` (which, by definition, cannot contain duplicates), initially containing all the elements in `c`. It uses the standard conversion constructor described in the The Collection Interface section.

Or, if using JDK 8 or later, you could easily collect into Set using aggregate operations:

```
c.stream()
 .collect(Collectors.toSet()); // no duplicates
```

Here's a slightly longer example that accumulates a `Collection` of names into a `TreeSet`:

```
Set<String> set = people.stream()
.map(Person::getName)
.collect(Collectors.toCollection(TreeSet::new));
```

And the following is a minor variant of the first idiom that preserves the order of the **original collection** while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a `Set` of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {
    return new LinkedHashSet<E>(c);
}
```

Set Interface Basic Operations

The `size` operation returns the number of elements in the `Set` (its *cardinality*). The `isEmpty` method does exactly what you think it would. The `add` method adds the specified element to the `Set` if it is not already present and returns a boolean indicating whether the element was added. Similarly, the `remove` method removes the specified element from the `Set` if it is present and returns a boolean indicating whether the element was present. The `iterator` method returns an `Iterator` over the `Set`.

The methods declared by Set are summarized in the following table:

Method	Description

boolean add(*Object*)	**Adds the specified element to this set if it is not already present (optional operation).**
boolean addAll(*Collection*)	**Adds all of the elements in the specified collection to this set if they're not already present (optional operation).**
void clear()	**Removes all of the elements from this set (optional operation).**
boolean contains(*Object*)	**Returns** true **if this set contains the specified element.**
boolean containsAll(*Collection*)	**Returns** true **if this set contains all of the elements of the specified collection.**
boolean equals(*Object*)	**Compares the specified object with this set for equality.**
int hashCode()	**Returns the hash code value for this set.**
boolean isEmpty()	**Returns** true **if this set contains no elements.**
Iterator iterator()	**Returns an iterator over the elements in this set.**
boolean remove(*Object*)	**Removes the specified element from this set if it is present (optional operation).**
boolean removeAll(*Collection*)	**Removes from this set all of its elements that are contained in the specified collection (optional operation).**
boolean retainAll(*Collection*)	**Retains only the elements in this set that are contained in the specified collection (optional operation).**
int size()	**Returns the number of elements in this set (its cardinality).**
Object[] toArray()	**Returns an array containing all of the elements in this set.**
Object[] toArray(*Object[]*)	**Returns an array containing all of the elements in this set whose runtime type is that of the specified array.**

Differences between List and Set interfaces.

List(I)	Set(I)
`List` in Java allows duplicates	**`Set` doesn't allow any duplicate**. If you insert **duplicate** in `Set` it will replace the older value. Any implementation of `Set` in Java will only contains unique elements.
`List` is an Ordered Collection. `List` maintains **insertion order of elements**, means any element which is inserted before will go on lower index than any element which is inserted after.	`Set` is an unordered Collection. `Set` in Java doesn't maintain any order. Though `Set` provide another alternative called `SortedSet` which can store `Set` elements in specific Sorting order defined by Comparable and Comparator methods of Objects stored in `Set`.

	`Set` uses equals() method to check uniqueness of elements stored in `Set`, while `SortedSet` uses `compareTo()` method to implement natural sorting order of elements. In order for an element to behave properly in `Set` and `SortedSet`, equals and compareTo must be consistent to each other.
`Set` uses equals() method to check uniqueness of elements stored in `Set`, while `SortedSet` uses `compareTo()` method to implement natural sorting order of elements. In order for an element to behave properly in `Set` and `SortedSet`, equals and compareTo must be consistent to each other.	popular implementation of `Set` interface includes `HashSet`, `TreeSet` and **`LinkedHashSet`**.
if you need to maintain insertion order or object and you collection can contain duplicates than `List` is a way to go. On the other hand if your requirement is to maintain unique collection without any duplicates than `Set` is the way to go.	if you need to maintain insertion order or object and you collection can contain duplicates than `List` is a way to go. On the other hand if your requirement is to maintain unique collection without any duplicates than `Set` is the way to go.
`List` is an ordered collection which allows duplicates.	`Set` is an unordered collection which doesn't allow duplicates.

Imp points of Set:

1. It is an interface extended from collection.
2. the Set interface provides methods for accessing the elements of a finite mathematical set.
3. It doesn't follow insertion order.
4. It accepts null only once.
5. It doesn't accept any duplicate objects (It has no duplicates).

How to convert HashMap to Set:

Let us assume HashMap object as 'hm'

Key	Value
501	*BVN*
502	*Namish*
503	*Nani*
503	*Narayana*

If we call Hm.keySet()

Set s = hm.keySet()----------we are adding keys into Set(I).

Note: we use Set(I) to store unique keys and unique values. So Set doesn't allow duplicate keys.

Set Implementations

Being a `Collection` subtype all methods in the `Collection` interface are also available in the `Set` interface.

Since `Set` is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following `Set` implementations in the Java Collections API:

java.util.EnumSet
java.util.HashSet
java.util.LinkedHashSet
java.util.TreeSet

Generic Sets

By default you can put any `Object` into a `Set`, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a `Set`. Here is an example:

```
Set<MyObject> set = new HashSet<MyObject>();
```

This `Set` can now only have `MyObject` instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

```
for(MyObject anObject : set){
   //do someting to anObject...
}
```

Imp points:

Set Interface Bulk Operations:

Bulk operations are particularly well suited to `Set`s; when applied, they perform standard set-algebraic operations. Suppose `s1` and `s2` are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if `s2` is a **subset** of `s1`. (`s2` is a subset of `s1` if set `s1` contains all of the elements in `s2`.)
- `s1.addAll(s2)` — transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)
- `s1.retainAll(s2)` — transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

---

1. HashSet Class (Class java.util.HashSet):

java.lang.Object

java.util.AbstractCollection

java.util.AbstractSet
java.util.HashSet

public class **HashSet**
implements Set, Cloneable, Serializable
extends AbstractSet

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.

This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the `HashSet` instance:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

The iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the Iterator throws a `ConcurrentModificationException`.

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

A Hsahtable stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

The HashSet class supports four constructors. The first form constructs a default hash set:

```
HashSet( )
```

The following constructor form initializes the hash set by using the elements of c.

```
HashSet(Collection c)
```

The following constructor form initializes the capacity of the hash set to capacity.

The capacity grows automatically as elements are added to the Hash.

```
HashSet(int capacity)
```

The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments:

```
HashSet(int capacity, float fillRatio)
```

Here the fill ratio must be between 0.0 and 1.0, and it determines how full the HashSet can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

Apart from the methods inherited from its parent classes, HashSet defines following methods:

SN	Methods with Description
1	**boolean add(Object o)** Adds the specified element to this set if it is not already present.
2	**void clear()** Removes all of the elements from this set.
3	**Object clone()** Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
4	**boolean contains(Object o)** Returns true if this set contains the specified element
5	**boolean isEmpty()** Returns true if this set contains no elements.
6	**Iterator iterator()** Returns an iterator over the elements in this set.
7	**boolean remove(Object o)** Removes the specified element from this set if it is present.
8	**int size()** Returns the number of elements in this set (its cardinality).

Imp points:

1) `HashSet` implements `java.util.Set` interface which means they follow contract of `Set` interface and doesn't allow any duplicates.

2) HashSet and is not thread-safe and not synchronized. Though you can make them synchronized by using Collections.synchronizedSet() method.
3) Iterator of both classes are fail-fast in nature. They will throw ConcurrentModificationException if Iterator is modified once Iterator is created. this is not guaranteed and application code should not rely on this code but Java makes best effort to fail as soon as it detects structural change in underlying Set.
4) In performance, HashSet is faster than TreeSet and should be preferred choice if sorting of element is not required.
5) HashSet allows null object,
6) HashSet is faster than TreeSet which means if you need performance use HashSet but HashSet doesn't provide any kind of ordering so if you need ordering then you need to switch to TreeSet which provides sorting of keys. Sorting can be natural order defined by Comparable interface or any particular order defined by Comparator interface in Java.

2. LinkedHashSet Class:

The **java.util.LinkedHashSet** class is a Hashtable and LinkedList implementation of the Set interface, with predictable iteration order. Following are the important points about LinkedHashSet:

- This class provides all of the optional Set operations, and permits null elements.

- This class extends HashSet, but adds no members of its own.
- LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.
- That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
- It contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
- It maintains insertion order.
- HashSet is fastest, LinkedHashSet is second on performance.
- LinkedHashSet maintains insertion order of elements.
- LinkedHashSet is implemented using HashSet.
- Both HashSet and LinkedHashSet allow null.

Class declaration:

Following is the declaration for **java.util.LinkedHashSet** class:

```
public class LinkedHashSet<E>
   extends HashSet<E>
     implements Set<E>, Cloneable, Serializable
```

Parameters:

Following is the parameter for **java.util.LinkedHashSet** class:

- **E** -- This is the type of elements maintained by this set.

Class constructors

S.N.	Constructor & Description
1	**LinkedHashSet()** This constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
2	**LinkedHashSet(Collection<? extends E> c)** This constructs a new linked hash set with the same elements as the specified collection.
3	**LinkedHashSet(int initialCapacity)** This constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
4	**LinkedHashSet(int initialCapacity, float loadFactor)** This constructs a new, empty linked hash set with the specified initial capacity and load factor.

(or)

The LinkedHashSet class supports four constructors. The first form constructs a default hash set:

```
LinkedHashSet( )
```

The following constructor form initializes the hash set by using the elements of c.

```
LinkedHashSet(Collection c)
```

The following constructor form initializes the capacity of the hash set to capacity.
The capacity grows automatically as elements are added to the Hash.

```
LinkedHashSet(int capacity)
```

The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments:

```
LinkedHashSet(int capacity, float fillRatio)
```

Class methods:

This class inherits methods from the following classes:

- java.util.HashSet
- java.util.AbstractSet
- java.util.AbstractCollection
- java.util.Object
- java.util.Set

3. TreeSet Class(Class java.util.TreeSet):

java.lang.Object
  java.util.AbstractCollection
    java.util.AbstractSet
      java.util.TreeSet

public class **TreeSet**
implements SortedSet, Cloneable, Serializable
extends AbstractSet

This class implements the `Set` interface, backed by a `TreeMap` instance.

This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements (see `Comparable`), or by the comparator provided at set creation time, depending on which constructor is used.
This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).
Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the `Set` interface. (See `Comparable` or `Comparator` for a precise definition of *consistent with equals*.) This is so because the `Set` interface is defined in terms of the `equals` operation, but a `TreeSet` instance performs all key comparisons using its `compareTo` (or `compare`) method, so two keys that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Set` interface.

Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

The Iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`.

More info:

The **java.util.TreeSet** class implements the **Set** interface. Following are the important points about TreeSet:

- The TreeSet class guarantees that the Map will be in ascending key order and backed by a TreeMap.
- The Map is sorted according to the natural sort method for the key Class, or by the Comparator provided at set creation time, that will depend on which constructor used.
- The ordering must be total in order for the Tree to function properly.
- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Class declaration

Following is the declaration for **java.util.TreeSet** class:

```
public class TreeSet<E>
```

```
   extends AbstractSet<E>
      implements NavigableSet<E>, Cloneable, Serializable
```

Parameters

Following is the parameter for **java.util.TreeSet** class:

- **E** -- This is the type of elements maintained by this set.

Class constructors

S.N.	Constructor & Description
1	**TreeSet()** This constructor constructs a new, empty tree set, sorted according to the natural ordering of its elements.
2	**TreeSet(Collection<? extends E> c)** This constructor constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.
3	**TreeSet(Comparator<? super E> comparator)** This constructor constructs a new, empty tree set, sorted according to the specified comparator.
4	**TreeSet(SortedSet<E> s)** This constructor constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

Difference between HashSet and TreeSet in Java

What is common in HashSet and TreeSet in Java?

1)Both `HashSet` and `TreeSet` implements `java.util.Set` interface which means they follow contract of `Set` interface and doesn't allow any duplicates.
2) Both `HashSet` and `TreeSet` are not thread-safe and not synchronized. Though you can make them synchronized by using `Collections.synchronizedSet()` method.
3) Third similarity between `TreeSet` and `HashSet` is that, Iterator of both classes are fail-fast in nature. They will throw `ConcurrentModificationException` if Iterator is modified once Iterator is created. this is not guaranteed and application code should not rely on this code but Java makes best effort to fail as soon as it detects structural change in underlying `Set`.

HashSet vs TreeSet in Java:

1) First major difference between `HashSet` and `TreeSet` is performance. `HashSet` is faster than `TreeSet` and should be preferred choice if sorting of element is not required.

2) Second difference between `HashSet` and `TreeSet` is that `HashSet` allows null object but `TreeSet` doesn't allow null Object and throw NullPointerException, Why, because `TreeSet` uses compareTo() method to compare keys and `compareTo()` will throw `java.lang.NullPointerException` as shown in below example :

3) Probably most important difference between HashSet and `TreeSet` is the performance. `HashSet` is faster than `TreeSet` which means if you need performance use `HashSet` but `HashSet` doesn't provide any kind of ordering so if you need ordering then you need to switch to `TreeSet` which provides sorting of keys. Sorting can be natural order defined by Comparable interface or any particular order defined by Comparator interface in Java.

****

17.3 Map(I)

Map(I):

1) HashMap(C):
2) HashTable(C):
3) SortedMap(C):
4) IdentityHashMap(C):
5) ConcurrentHashMap(C):
6) TreeMap(C):

III.Map Interface (Interface java.util.Map):

public interface **Map**

An object that maps keys to values.

A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

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.

The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behaviour of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors:

1. void (no arguments) constructor which creates an empty map, and
2. constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class.

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

SN	Methods with Description

1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains k as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)** Returns true if the map contains v as a value. Otherwise, returns false.
4	**Set entrySet( )** Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key k.
7	**int hashCode( )** Returns the hash code for the invoking map.
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**Set keySet( )** Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from m into this map.
12	**Object remove(Object k)** Removes the entry whose key equals k.
13	**int size( )** Returns the number of key/value pairs in the map.
14	**Collection values( )** Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Map Implementations

Since `Map` is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following `Map` implementations in the Java Collections API:

java.util.HashMap
java.util.Hashtable
java.util.EnumMap
java.util.IdentityHashMap
java.util.LinkedHashMap
java.util.Properties
java.util.TreeMap
java.util.WeakHashMap

Adding and Accessing Elements

To add elements to a `Map` you call it's `put()` method. Here are a few examples:

```
Map m = new HashMap();

m.put("key1", "element 1");
m.put("key2", "element 2");
m.put("key3", "element 3");
```

The three `put()` calls maps a string value to a string key. You can then obtain the value using the key. To do that you use the `get()` method like this:

```
String element1 = (String) m.get("key1");
```

You can iterate either the keys or the values of a `Map`. Here is how you do that:

```
// key iterator
Iterator iterator = m.keySet().iterator();

// value iterator
Iterator iterator = m.values();
```

Most often you iterate the keys of the `Map` and then get the corresponding values during the iteration. Here is how it looks:

```
Iterator iterator = m.keySet().iterator();
while(iterator.hasNext(){
  Object key   = iterator.next();
  Object value = m.get(key);
}

//access via new for-loop
for(Object key : m.keySet()) {
    Object value = m.get(key);
```

```
}
```

Removing Elements

You remove elements by calling the `remove(Object key)` method. You thus remove the (key, value) pair matching the key.

Generic Maps

By default you can put any `Object` into a `Map`, but from Java 5, Java Generics makes it possible to limit the types of object you can use for both keys and values in a `Map`. Here is an example:

```
Map<String, MyObject> map = new HashMap<String, MyObject>();
```

This `Map` can now only accept `String` objects for keys, and `MyObject` instances for values. You can then access and iterate keys and values without casting them. Here is how it looks:

```
for(MyObject anObject : map.values()){
   //do someting to anObject...
}

for(String key : map.keySet()){
   MyObject value = map.get(key);
   //do something to value
}
```

1. HashMap Class(Class java.util.HashMap):

```
java.lang.Object
        java.util.AbstractMap
                java.util.HashMap
```

public class **HashMap**
implements Map, Cloneable, Serializable
extends AbstractMap

Hashtable based implementation of the `Map` interface.
This implementation provides all of the optional map operations and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.)
This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of `HashMap` has two parameters that affect its performance: *initial capacity* and *load factor*.
The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.

The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the `rehash` method.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the `HashMap` class, including `get` and `put`). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of `rehash` operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no `rehash` operations will ever occur.

If many mappings are to be stored in a `HashMap` instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Note that this implementation is not synchronized. If multiple threads access this map concurrently,and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`.

Introduction:

The java.util.HashMap class is the Hashtable based implementation of the Map interface.

Following are the important points about HashMap:

1. This class makes no guarantees as to the iteration order of the map; in particular, it does not guarantee that the order will remain constant over time.
2. This class permits null values and the null key.
3.

Class declaration

Following is the declaration for **java.util.HashMap** class:

```
public class HashMap<K,V>
   extends AbstractMap<K,V>
       implements Map<K,V>, Cloneable, Serializable
```

Parameters

Following is the parameter for **java.util.HashMap** class:

- **K** -- This is the type of keys maintained by this map.
- **V** -- This is the type of mapped values.

Class constructors

S.N.	Constructor & Description
1	**HashMap()** This constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
2	**HashMap(Collection<? extends E> c)** This constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
3	**HashMap(int initialCapacity, float loadFactor)** This constructs an empty HashMap with the specified initial capacity and load factor.
4	**HashMap(Map<? extends K,? extends V> m)** This constructs a new HashMap with the same mappings as the specified Map.

Class methods

S.N.	Method & Description
1	void clear() This method removes all of the mappings from this map.
2	Object clone() This method returns a shallow copy of this HashMap instance, the keys and values themselves are not cloned.
3	boolean containsKey(Object key) This method returns true if this map contains a mapping for the specified key.
4	boolean containsValue(Object value) This method returns true if this map maps one or more keys to the specified value.
5	Set<Map.Entry<K,V>> entrySet() This method returns a Set view of the mappings contained in this map.
6	V get(Object key) This method returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
7	boolean isEmpty() This method returns true if this map contains no key-value mapping.
8	Set<K> keySet() This method returns a Set view of the keys contained in this map.
9	V put(K key, V value) This method associates the specified value with the specified key in this map.
10	void putAll(Map<? extends K,? extends V> m) This method copies all of the mappings from the specified map to this map.

11	V remove(Object key) This method removes the mapping for the specified key from this map if present.
12	int size() This method returns the number of key-value mappings in this map.
13	Collection<V> values() This method returns a Collection view of the values contained in this map.

2. Hashtable Class(Class java.util.Hashtable):

```
java.lang.Object
        java.util.Dictionary
                java.util.Hashtable
```

public class **Hashtable** implements Map, Cloneable, Serializable extends Dictionary

This class maps keys to values. Any non-`null` object can be used as a key or as a value.

To successfully store and retrieve objects from a Hashtable, the objects used as keys must implement the `hashCode` method and the `equals()` method.

An instance of `Hashtable` has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the Hashtable, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the Hashtable is *open*: in the case a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the Hashtable is allowed to get before its capacity is automatically increased. When the number of entries in the Hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the `rehash` method.

Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most `Hashtable` operations, including `get` and `put`).

The initial capacity controls a tradeoff between wasted space and the need for `rehash` operations, which are time-consuming. No `rehash` operations will *ever* occur if the initial capacity is greater than the maximum number of entries the `Hashtable` will contain divided by its load factor. However, setting the initial capacity too high can waste space.

If many entries are to be made into a `Hashtable`, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a Hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable ht = new Hashtable();
      ht.put("one", new Integer(1));
      ht.put("two", new Integer(2));
      ht.put("three", new Integer(3));
```

To retrieve a number, use the following code:

```
Integer n = (Integer)ht.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```

The Iterators returned by the iterator and listIterator methods of the Collections returned by all of Hashtable's "collection view methods" are *fail-fast*: if the Hashtable is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException.

some more imp points:

1. Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but **Hashtable is synchronized.**
2. Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key and the value that you want linked to that key. The key is then hashed and the resulting hashcode is used as the index at which the value is stored within the table.
3. The **java.util.Hashtable** class implements a hashtable, which maps keys to values. Following are the important points about Hashtable:
   - In this any non-null object can be used as a key or as a value.
   - If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

Class declaration:

Following is the declaration for **java.util.Hashtable** class:

```
public class Hashtable<K,V>
   extends Dictionary<K,V>
      implements Map<K,V>, Cloneable, Serializable
```

The Hashtable defines four constructors. The first version is the default constructor:

```
Hashtable( )
```

The second version creates a hash table that has an initial size specified by size:

```
Hashtable(int size)
```

The third version creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.
This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.

```
Hashtable(int size, float fillRatio)
```

The fourth version creates a hash table that is initialized with the elements in m.
The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

```
Hashtable(Map m)
```

Apart from the methods defined by Map interface, Hashtable defines the following methods:

SN	Methods with Description
1	**void clear( )** Resets and empties the hash table.
2	**Object clone( )** Returns a duplicate of the invoking object.
3	**boolean contains(Object value)** Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
4	**boolean containsKey(Object key)** Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found.
5	**boolean containsValue(Object value)** Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
6	**Enumeration elements( )** Returns an enumeration of the values contained in the hash table.
7	**Object get(Object key)** Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.
8	**boolean isEmpty( )** Returns true if the hash table is empty; returns false if it contains at least one key.
9	**Enumeration keys( )** Returns an enumeration of the keys contained in the hash table.
10	**Object put(Object key, Object value)** Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table.
11	**void rehash( )** Increases the size of the hash table and rehashes all of its keys.
12	**Object remove(Object key)** Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned.
13	**int size( )** Returns the number of entries in the hash table.
14	**String toString( )** Returns the string equivalent of a hash table.

Map.Entry Interface:

The Map.Entry interface enables you to work with a map entry.

The **entrySet( )** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is

Following table summarizes the methods declared by this interface:

SN	Methods with Description
1	**boolean equals(Object obj)** Returns true if obj is a Map.Entry whose key and value are equal to that of the invoking object.
2	**Object getKey( )** Returns the key for this map entry.
3	**Object getValue( )** Returns the value for this map entry.
4	**int hashCode( )** Returns the hash code for this map entry.
5	**Object setValue(Object v)** Sets the value for this map entry to v. A ClassCastException is thrown if v is not the correct type for the map. A NullPointerException is thrown if v is null and the map does not permit null keys. An UnsupportedOperationException is thrown if the map cannot be changed.

Conversions of Collection interfaces:

How to convert HashMap to Set:

Let us assume HashMap object as 'hm'

Key	Value
501	*BVN*
502	*Namish*
503	*Nani*
503	*Narayana*

If we call Hm.keySet()

Set s = hm.keySet()----------we are adding keys into Set(I).

Note: we use Set(I) to store unique keys and unique values. So Set doesn't allow duplicate keys.

****

Topic Name18: Cursers in Collections Framework: Enumeration(I), Iterator(I), ListIterator(I):

Topic Name19: collections class, comparable & comparator interfaces:

1) **Collections Class ---- java.util.Collections**
2) **Comparable interface ----------java.lang.Comparable**
3) **Comparator interface ----------java.util.Comparator**

Class java.util.Collections

```
java.lang.Object
        java.util.Collections
```

public class **Collections**
extends Object

- The class `java.util.Collections` is a utility class that resides in `java.util` package, it consists entirely of static methods which are used to operate on collections like `List`, `Set`. Common operations like sorting a `List` or finding an element from a `List` can easily be done using the `Collections` class.
- This class consists exclusively of static methods that operate on or return collections.
- It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection.
- The methods of this class all throw a NullPointerException if the collections or class objects provided to them are null.

Some of the Imp methods in collections class:

`Collections.sort(List<T extends Comparable<? super T>> list)` is used to sort the given list in ascending order according to the natural ordering.

`Collections.binarySearch(List<T extends Comparable<? super T>> list, T key)` is used to search the element T in the given list. It returns an index of the searched element if found; otherwise (-) insertion point. Please note that the list must be in sorted into ascending order before calling this method otherwise result would not be as expected.

`Collections.shuffle(List<?> list)` is used to randomly shuffle the elements in the list. It will give different results in different call.

`Collections.fill(List<? super T> list, T obj)` is used to replace the elements of the given list with the specified element. As shown in this example we have replaced all temperature of the list with 0.0.

`Collections.max(Collection<? extends T> coll)` returns the maximum element of the given collection, according to the natural ordering of its elements. We used it to get the maximum temperature in the given list.

`Collections.min(Collection<? extends T> coll)` returns the minimum element of the given collection, according to the natural ordering of its elements. We used it to get the minimum temperature in the given list.

`Collections.reverse(List<?> list)`, reverses the order of the elements in the specified list.

`Collections.copy(List<? super T> dest, List<? extends T> src)` it copies all the elements from one list into another. After this method the index of each copied element in the destination list will be identical to its index in the source list. Destination list needs to be equal or bigger in the size.

Example for static method: java.util.Collections.sort() Method

Description:
The **sort(List<T>)** method is used to sort the specified list into ascending order, according to the natural ordering of its element.
Declaration:
Following is the declaration for **java.util.Collections.sort()** method.

```
public static void sort(List<T> list)
```

Parameters:

- **list--**this is the list to be sorted.

Return Value:

- **NA**

Exception:

- **ClassCastException--**Throws if the list contains elements that are not mutually comparable (for example, strings and integers).
- **UnsupportedOperationException--**Throws if the specified list's list-iterator does not support the set operation.

Sort ArrayList using Comparator:

To sort an ArrayList using a Comparator you should:

- Create a new ArrayList.
- Populate the arrayList with elements, using `add(E e)` API method of ArrayList.
- Invoke `reverseOrder()` API method of Collections to get a Comparator that imposes the reverse of the natural ordering on the list's elements.
- Invoke `sort(List list, Comparator c)` API method of Collections in order to sort the arrayList elements using the comparator. The arrayList's elements will be sorted according to the comparator.

  We can get the arrayList's elements before and after sorting to check how they are sorted. Before sorting the arrayList elements are sorted by insertion order, and after sorting the elements are in reverse than the natural ordering.

What is the Difference between Collection and Collections?

Collection	Collections
A collection represents a group of objects, known as its elements. The `Collection` interface is the least common denominator that all collections implement, and is used to pass collections around and to manipulate them when maximum generality is desired.	A collections framework is a unified architecture for representing and manipulating collections.
Collection is an interface....(it is the root interface of java collection framework)	Collections is a utility class having static methods. Collections has methods for finding maximum number of elements in collection.
`Collection`, as its javadoc says is "*The root interface in the collection hierarchy.*" This means that every single class implementing `Collection` in any form is part of the Java Collections Framework.	The Collections Framework is Java's native implementation of data structure classes (*with implementation specific properties*) which represent a group of objects which are somehow related to each other and thus can be called a collection.
The main difference in my opinion is that `Collection` is base interface which you may use in your code as a type for object (*although I wouldn't directly recommend that*) **while `Collections` just provides useful operations for handling the collections.**	`Collections` is merely an utility method class for doing certain operations, for example adding thread safety to your ArrayList instance by doing this: List<MyObj> list = Collections.synchronizedList(new Arraylist<MyObj>());

Comparable & Comparator Interfaces:

1. Comparable interface:

Fully qualified name: java.lang.Comparable

java.lang Interface Comparable<T>

Type Parameters:

`T` - the type of objects that this object may be compared to

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a`NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

```
{(x, y) such that x.compareTo(y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that x.compareTo(y) == 0}.
```

It follows immediately from the contract for `compareTo` that the quotient is an *equivalence relation* on `C`, and that the natural ordering is a *total order* on `C`. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

```
{(x, y) such that x.equals(y)}.
```

This interface is a member of the Java Collections Framework.

Method Detail:

int compareTo(T o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters: `o` - the object to be compared.

Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws: `NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

How to use comparable interface for Natural sorting order:

Java allows you to sort your object in natural order by implementing Comparable interface.

It's one of the fundamental interfaces of Java API and defined in java.lang package, which means you don't need to implement this unlike its counterpart Comparator, which is defined in java.util package.

Comparable is used to provide natural order of sorting to objects e.g. numeric order is natural order for numbers, alphabetic order is natural order for String and chronological order is natural for dates. Similarly when you define your own objects e.g. Person, sorting it on name sounds natural. Similarly for teams, ranking seems their natural orders.

It all depends how object is looked in their domain. By the way, you are not limited to just one order, you can sort objects in any order by using java.util.Comparator.

Comparable interface defines abstract method compareTo(), you need to override this method to implement natural order sorting of objects in Java. This method returns positive if the object, on which you are calling this method is greater than other object, return negative, if this object is less than other and returns zero if both object are equal.

Several or other classes from Java API relies on this interface for their behaviour, for example Arrays.sort(), Collections.sort() uses this method to sort objects contained in Array and Collection in their natural order. Similarly SortedSet and SortedMap implementation e.g. TreeSet and TreeMap also uses compareTo() method to keep their elements sorted. I have earlier shared some tips to override compareTo() method, which is also worth looking if you are implementing Comparable interface.

Comparable has only one method `compareTo()`, which is where we define logic to compare objects. As I said in previous paragraph that this method returns positive, negative and zero depending upon result of comparison, but most important thing, there is no restriction on return 1, 0 or -1 for greater, equal and lesser results, you can return any positive or negative number, this property is utilized in this example, but it comes with caveat that difference between them should not exceed, `Integer.MAX_VALUE`.

There are few more things, you need to consider while implementing Comparable interface. Before Java 5, its `compareTo()` method accepts `java.lang.Object`, but after introduction of Generics, this class can be written in standard type `T`. This makes it type-safe with compiler helping you by flagging error, when you pass different object to `compareTo()`. So always use generic version. Second thing, make sure your `compareTo()` is consistent with equals() method. Though this is not a requirement mandated by compiler or Java API, and there are examples of violating this principle in Java API itself e.g. `BigDecimal`, you should make them consistent if you are going to store object in `SortedSet` or `SortedMap`.

Object Ordering

A `List l` may be sorted as follows.

```
Collections.sort(l);
```

If the `List` consists of `String` elements, it will be sorted into alphabetical order. If it consists of `Date` elements, it will be sorted into chronological order. How does this happen? `String` and `Date` both implement the `Comparable` interface. `Comparable` implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement `Comparable`.

Classes Implementing Comparable

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic

Date	Chronological
CollationKey	Locale-specific lexicographic

If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a ClassCastException. Similarly, Collections.sort(list, comparator) will throw a ClassCastException if you try to sort a list whose elements cannot be compared to one another using the comparator. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

Summary: Java provides **Comparable** interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has **compareTo(T obj)** method which is used by sorting methods, you can check any Wrapper, String or Date class to confirm this. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if "this" object is less than, equal to, or greater than the object passed as argument.

2. Comparator interface:

Fully qualified name: **java.util.Comparator**

What if you want to sort some objects in an order other than their natural ordering for example for a list of employees in order of seniority.? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a Comparator — an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

The compare method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Much of what was said about Comparable applies to Comparator as well. Writing a compare method is nearly identical to writing a compareTo method, except that the former gets both objects passed in as arguments. The compare method has to obey the same four technical restrictions as Comparable's compareTo method for the same reason — a Comparator must induce a total order on the objects it compares.

Suppose you have a class called Employee, as follows.

```
public class Employee implements Comparable<Employee> {
    public Name name()     { ... }
    public int number()    { ... }
    public Date hireDate() { ... }
       ...
}
```

How to use Comparator?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order:

The compare Method:

```
int compare(Object obj1, Object obj2)
```

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals Method:

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(Object obj)
```

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Overriding equals( ) is unnecessary, and most simple comparators will not do so.

Examples for use of comparator:

1. In most real life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where we need to use Comparator interface because *Comparable.compareTo(Object o)* method implementation can sort based on one field only and we can't chose the field on which we want to sort the Object.
2. Comparator interface *compare(Object o1, Object o2)* method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.
3. Both Comparable and Comparator interfaces uses Generics for compile time type checking.

Comparator vs Comparable:

Parameter	Comparable	Comparator
Sorting logic	Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects	Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc.
Implementation	Class whose objects to be sorted must implement this interface. e.g Country class needs to implement comparable to collection of country object by id	Class whose objects to be sorted do not need to implement this interface. Some other class can implement this interface. E.g.- CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id
Sorting method	int compareTo(Object o1) This method compares this object with o1 object and returns a integer.Its value has following meaning 1. positive – this object is greater than o1 2. zero – this object equals to o1 3. negative – this object is less than o1	int compare(Object o1,Object o2) This method compares o1 and o2 objects. and returns a integer.Its value has following meaning. 1. positive – o1 is greater than o2 2. zero – o1 equals to o2 3. negative – o1 is less than o1
Calling method	Collections.sort(List) Here objects will be sorted on the basis of CompareTo method	Collections.sort(List, Comparator) Here objects will be sorted on the basis of Compare method in Comparator
Package	Java.lang.Comparable	Java.util.Comparator

When to use Comparator and Comparable in Java

At last let's see some best practices and recommendation on when to use Comparator or Comparable in Java:

1) If there is a natural or default way of sorting `Object` already exist during development of `Class` than use `Comparable`. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, `Employee` can be sorted by their Id etc. On the other hand if an `Object` can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use `Comparator` interface. For example Employee can again be sorted on name, salary or department and clients needs an API to do that. `Comparator` implementation can sort out this problem.

2) Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can't implement `Comparable` and `Comparator` is only way to sort those objects.

3) Beware with the fact that how those object will behave if stored in `SorteSet` or `SortedMap` like TreeSet and TreeMap. If an object doesn't implement Comparable than while putting them into `SortedMap`, always provided corresponding `Comparator` which can provide sorting logic.

4) Order of comparison is very important while implementing `Comparable` or `Comparator` interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously.

5) Comparator has a distinct advantage of being self descriptive for example if you are writing Comparator to compare two Employees based upon their salary than name that comparator as `SalaryComparator`, on the other hand compareTo()

Topic Name20: Multithreaded Programming:

Topic Name21: IO Streams:

Topic Name22: Serialization:

Serialization in Java

Serialization in java is a mechanism of *writing the state of an object into a byte stream*.

It is mainly used in Hibernate, RMI, JPA, EJB, JMS technologies.

The reverse operation of serialization is called *deserialization*.

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

Advantage of Java Serialization

It is mainly used to travel object's state on the network (known as marshaling).

---

java.io.Serializable interface

Serializable is a marker interface (has no body). It is just used to "mark" java classes which support a certain capability.

It must be implemented by the class whose object you want to persist. Let's see the example given below:

```
1. import java.io.Serializable;
2. public class Student implements Serializable{
3.  int id;
4.  String name;
5.  public Student(int id, String name) {
6.   this.id = id;
7.   this.name = name;
8.  }
9. }
```

Marker interface:

Marker interface in Java is interfaces with no field or methods or in simple word **empty interface in java is called marker interface**. Example of market interface is Serializable, Clonnable and Remote interface. Now if marker interface doesn't have any field or method or behavior they why would Java needs it? Marker interface are also called **tag interface in Java**.

What is Marker interfaces in Java and why required

Why Marker or Tag interface do in Java

1) Looking carefully on marker interface in Java e.g. **Serializable, Clonnable and Remote** it looks they are **used to indicate something to compiler or JVM**. So if JVM sees a Class is Serializable it done some special operation on it, similar way if JVM sees one Class is implement Clonnable it performs some operation to support cloning. Same is true for **RMI and Remote interface**. So in short Marker interface indicate, signal or a command to Compiler or JVM.

This is pretty standard answer of question about *marker interface* and once you give this answer most of the time interviewee definitely asked "**Why this indication can not be done using a flag inside a class**?" this make sense right? Yes this can be done by using a boolean flag or a String but doesn't marking a class like Serializable or Clonnable makes it more readable and it also allows to take advantage of Polymorphism in Java.

Where Should I use Marker interface in Java

Apart from using built in marker interface for making a class Serializable or Clonnable. One can also develop his own marker interface. Marker interface is a good way to classify code. You can create marker interface to logically divide your code and if you have your own tool than you can perform some pre-processing operation on those classes. Particularly useful for developing API and framework like Spring or Struts.

After introduction of Annotation on Java5, Annotation is better choice than marker interface and JUnit is a perfect example of using Annotation e.g**. @Test for specifying a Test Class**. Same can also be achieved by using Test marker interface.

Another use of marker interface in Java

One more use of marker interface in Java can be commenting. a marker interface called Thread Safe can be used to communicate other developers that classes implementing this marker interface gives thread-safe guarantee and any modification should not violate that. Marker interface can also help code coverage or code review tool to find bugs based on specified behavior of marker interfaces. Again Annotations are better choice `@ThreadSafe` looks lot better than implementing ThraedSafe marker interface.

In summary **marker interface in Java is used to indicate something to compiler, JVM** or any other tool but **Annotation** is better way of doing same thing.

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.

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

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

Important Methods

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

Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

ObjectInputStream class

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

Constructor

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

Important Methods

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

Java Serialization with Inheritance (IS-A Relationship)

If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Java Serialization with Aggregation (HAS-A Relationship)

If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

Java Serialization with static data member

If there is any static data member in a class, it will not be serialized because static is the part of class not object.

Java Serialization with array or collection

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.

Externalizable in java

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- **public void writeExternal(ObjectOutput out) throws IOException**

public void readExternal(ObjectInput in) throws IOException

Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient.

Java Transient Keyword

Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

Example of Java Transient Keyword

In this example, we have created the two classes Student and PersistExample. The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

Let's create a class with transient variable.

```
import java.io.Serializable;
public class Student implements Serializable{
        int id;
        String name;
        transient int age;//Now it will not be serialized
        public Student(int id, String name,int age) {
         this.id = id;
         this.name = name;
         this.age=age;
    }
}
```

Volatile variable in Java:

Volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory. So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable. From Java 5 along with major changes like Autoboxing, Enum, Generics and Variable arguments , Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made by one thread to another also as "happens-before" which solves the problem of memory writes that happen in one

thread can "leak through" and be seen by another thread. Java volatile keyword cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees visibility and ordering , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from synchronization barrier.

Important points on Volatile keyword in Java

1. volatile keyword in Java is only application to variable and using volatile keyword with class and method is illegal.

2. volatile keyword in Java guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.

3. In Java reads and writes are atomic for all variables declared using Java volatile keyword (including long and double variables).

4. Using Volatile keyword in Java on variables reduces the risk of memory consistency errors, because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.

5. From Java 5 changes to a volatile variable are always visible to other threads. What's more it also means that when a thread reads a volatile variable in Java, it sees not just the latest change to the volatile variable but also the side effects of the code that led up the change.

6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without use of volatile keyword in Java.

7. An access to a volatile variable in Java never has chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any lock.

8. Java volatile variable that is an object reference may be null.

9. Java volatile keyword doesn't means atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.

10. If a variable is not shared between multiple threads no need to use volatile keyword with that variable.

Difference between synchronized and volatile keyword in Java

What is difference between volatile and synchronized is another popular core Java question asked in multi-threading and concurrency interviews. Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.

1. Volatile keyword in Java is a field modifier, while synchronized modifies code blocks and methods.

2. Synchronized obtains and releases lock on monitor's Java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting any monitor in case of synchronized, that is not the case with volatile keyword in Java.

4. Synchronized method affects performance more than volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.

6. You can't synchronize on null object but your volatile variable in java could be null.

7. From Java 5 Writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

In **Summary** volatile keyword in Java is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with use of synchronization in Java

Topic Name23: Java 1.5 Features with Programs

All java1.5 features:

Java 2 Platform Standard Edition 5.0 is a major feature release. The features listed below are introduced in 5.0 since the previous major release (1.4.0).

Java Language Features

1. Generics
2. Enhanced for Loop
3. Autoboxing/Unboxing
4. Typesafe Enums
5. Varargs
6. Static Import
7. Metadata (Annotations)

1. Generics:

This long-awaited enhancement to the type system allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the work of casting.

Why Use Generics?

Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  When re-written to use generics, the code does not require casting:

  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types:

A *generic type* is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic `Box` class that operates on objects of any type. It needs only to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) {
           this.object = object;
      }
    public Object get() {
          return object;
    }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get `Integer`s out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

To update the `Box` class to use generics, you create a *generic type declaration* by changing the code "`public class Box`" to "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) {
            this.t = t;
        }
        public T get() {
            return t;
       }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions:

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

Invoking and Instantiating a Generic Type:

To reference the generic `Box` class from within your code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — `Integer` in this case — to the `Box` class itself.

Type Parameter and Type Argument Terminology:
Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "`Box` of `Integer`", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

For more information on diamond notation and type inference, see Type Inference.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;
```

```
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()     { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to autoboxing, it is valid to pass a `String` and an `int` to the class.

As mentioned in The Diamond, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types:

You can also substitute a type parameter (i.e., `K` or `V`) with a parameterized type (i.e., `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Raw Types:

A *raw type* is the name of a generic class or interface without any type arguments.

For example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
   public static void main(String[] args){
      Box<Integer> bi;
      bi = createBox();
   }

   static Box createBox(){
      return new Box();
   }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
```

```
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax,

Generic Methods:

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
   // Generic static method
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
   }
}

public class Pair<K, V> {

   private K key;
   private V value;

   // Generic constructor
   public Pair(K key, V value) {
      this.key = key;
      this.value = value;
   }

   // Generic methods
   public void setKey(K key) { this.key = key; }
   public void setValue(V value) { this.value = value; }
   public K getKey()   { return key; }
   public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

Generics, Inheritance, and Subtypes:

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an "is a" relationship. Since an `Integer` *is a* kind of `Object`, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

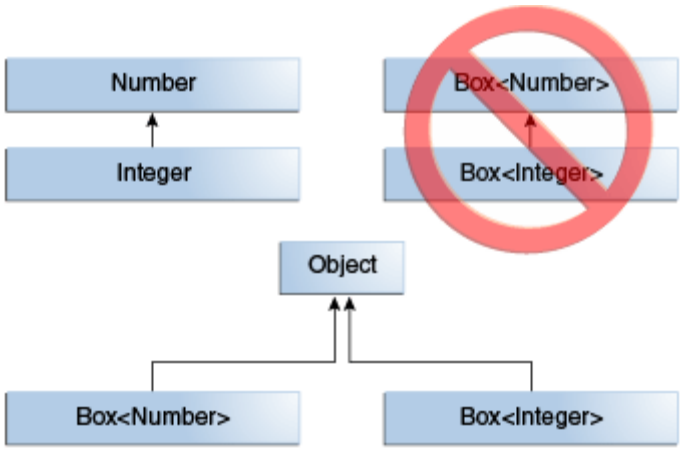
```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.

Note: Given two concrete types A and B (for example, Number and Integer).
MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.

Generic Classes and Subtyping:

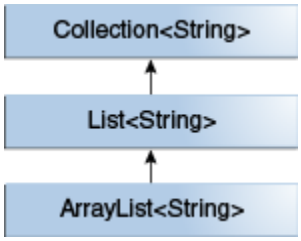
You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the `extends` and `implements` clauses.

Using the `Collections` classes as an example,

ArrayList<E> implements List<E>, and

List<E> extends Collection<E>.

So ArrayList<String> is a subtype of List<String>, which is a subtype of Collection<String>. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



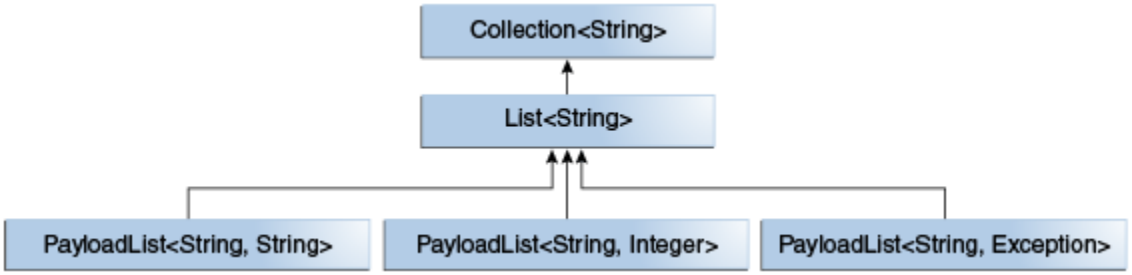
A sample `Collections` hierarchy

Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`



A sample `PayloadList` hierarchy

Wildcards

In generic code, the question mark (`?`), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

When creating a `Pair` object, you cannot substitute a primitive type for the type parameter `K` or `V`:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters `K` and `V`:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes `8` to `Integer.valueOf(8)` and '`a`' to `Character('a')`:

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the `append` method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or `instanceof` with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
```

```
    }
}
```

The set of parameterized types passed to the `rtti` method is:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most you can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the `Throwable` class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
```

```
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
   try {
      for (J job : jobs)
         // ...
   } catch (T e) {   // compile-time error
      // ...
   }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
   public void parse(File file) throws T {     // OK
      // ...
   }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Summary:

1. 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.
2. Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
3. Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

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).

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

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.

Map<String, Bork> myMap = new HashMap<String, Bork>();

That creates a map that uses a String key and a Bork value.

In old Java code, compiler accepts Date() object to be added to names List without any issue. But in runtime while retrieving that Date object, you will get the following exception: see

```
Exception in thread "Main Thread" java.lang.ClassCastException:
java.util.Date
      at
com.java.java5features.WithoutGenerics1.main(WithoutGenerics1.java:17)
```

So, to avoid such kind of unwanted exceptions, we can use Generics to provide compile-time type safety to collections.

Without Generics we will get exception at runtime not at compile time bz Compiler accepts.

We can also **nest generics** as follows:

```
Map<Integer, List<String>> mapStudents = new HashMap<Integer, List<String>>();
```

---

2. Enhanced for Loop:

This new language construct eliminates the drudgery and error-proneness of iterators and index variables when iterating over collections and arrays.

Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
  //Statements
}
```

declaration: The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

expression: This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Enhanced for loop is also referred as *'forEach'* Loop and is specifically designed to iterate through arrays and collections.

Here, we can simply iterate through the list 'names' and retrieve each element into variable 'name'. After that, we can do whatever we want with that variable as usual. Enhanced for loop avoids the need for using temporary index variable and simplifies the process of iterating over arrays and collections.

```
for(Iterator lineup = list.iterator() ; lineup.hasNext() ; ) {
  Object thatThing = lineup.next();
  myMonster.eat(thatThing);
}
```

In a shortened:

```
for(Object thatThing : list) {
  myMonster.eat(thatThing);
}
```

3. Autoboxing/Unboxing:

This facility eliminates the drudgery of manual conversion between primitive types (such as int) and wrapper types (such as Integer).

As any Java programmer knows, you can't put an int (or other primitive value) into a collection. Collections can only hold object references, so you have to *box* primitive values into the appropriate wrapper class (which is Integer in the case of int).

When you take the object out of the collection, you get the Integer that you put in; if you need an int, you must *unbox* the Integer using the **intValue()** method. All of this boxing and unboxing is a pain and clutters up your code. The autoboxing and unboxing feature automates the process, eliminating the pain and the clutter.

The following example illustrates autoboxing and unboxing, along with generics and the for-each loop. In a mere ten lines of code, it computes and prints an alphabetized frequency table of the words appearing on the command line.

```
 import java.util.*;
// Prints a frequency table of the words on the command line
public class Frequency {
  public static void main(String[] args) {
    Map m = new TreeMap();
    for (String word : args) {
       Integer freq = m.get(word);
       m.put(word, (freq == null ? 1 : freq + 1));
    }
    System.out.println(m);
  }
}
```

java Frequency if it is to be it is up to me to do the watusi
{be=1, do=1, if=1, is=2, it=2, me=1, the=1, to=3, up=1, watusi=1}

Before 1.5 you can not put primitive type values into collections. Collections can only hold object references. So Java 1.5 introduce Auto boxing which converts the primitive type into respected Wrapper class object. e.g. int become Integer Object automatically. The autoboxing and unboxing feature automates the process. The performance of the resulting list is likely to be poor, as it boxes and unboxes.

4. Typesafe Enums:

This flexible object-oriented enumerated type facility allows you to create enumerated types with arbitrary methods and fields. It provides all the benefits of the Typesafe Enum pattern ("Effective Java," Item 21) without the verbosity and the error-proneness.

5. Varargs:

Before using Varargs:
Let consider one simple example of finding the multiplication of n number. First we will try to solve this problem using method overloading.

```
/**
 * Java Program which tries to implement variable argument method using
 * method overloading. This started get clumsy once number of parameter exceeds
 * five.
 */
class  VarargsExample{

  public int multiply(int a,int b){ return a*b;}

  public int multiply(int a,int b,int c){ return (a*b)*c;}

  public int multiply(int a,int b,int c,int d{ return (a*b)*(c*d);}

}
```

If we use method overloading same method will be repeated again and again and it's not worth after four or five parameters.

Variable argument or Varargs methods from Java 5

Variable argument or varargs in Java allows you to write more flexible methods which can accept as many argument as you need. variable arguments or varargs were added in Java 1.5

varargs or variable arguments makes it possible for us to call one method with variable number of argument; means define only one method and call that method with zero or more than zero argument.

Syntax:

type ... variable Name.

3 dots is used to denote variable argument in a method and if there are more than one parameter, varargs arguments must be last.

suppose we go to one college and take admission on that college now it's not really decided that admission will be done for how many student may be 50 student will come or 100 or more than that at a time. So college is one class and Admission is one procedure or method that takes no of student as an argument .So in that method we can use varargs or variable arguments.

```
/**
 * Simple real world example of variable argument methods
 */
public class college {

public void admission_method (int... no_of_student) {
  }

}
```

This facility eliminates the need for manually boxing up argument lists into an array when invoking methods that accept variable-length argument lists.

Variable Arguments(var-args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java Variable Length Argument List

Varargs is used to take arbitrary number parameters to the method. Three dots (...) indicates after Object Type is method took number of parameters. Like in below display() method signature, having String type of Varargs list. In First display() method call we pass only one parameter, while in second call we pass two parameters, same way you can pass arbitrary number of parameters while calling the method.

6. Static Import:

This facility lets you avoid qualifying static members with class names without the shortcomings of the "Constant Interface antipattern."

In order to access static members, it is necessary to qualify references with the class they came from. For example, one must say:

```
double r = Math.cos(Math.PI * theta);
```

In order to get around this, people sometimes put static members into an interface and inherit from that interface. This is a bad idea. In fact, it's such a bad idea that there's a name for it: the *Constant Interface Antipattern* (see *Effective Java* Item 17). The problem is that a class's use of the static

members of another class is a mere implementation detail. When a class implements an interface, it becomes part of the class's public API. Implementation details should not leak into public APIs.

The static import construct allows unqualified access to static members *without* inheriting from the type containing the static members. Instead, the program *imports* the members, either individually:

import static java.lang.Math.PI;

or en masse:

import static java.lang.Math.*;

Once the static members have been imported, they may be used without qualification:

double r = cos(PI * theta);

Java 5 added new static import concept. To access the static members, it is necessary to specify from which class it comes from. In below example abs method comes from Math class. So prior Java 5 we need to specify the class name along with static member. But the new approach provides a facility to import it only once and then directly use that static member in code, static member may be method or variable.

7. Metadata (Annotations):

This language feature lets you avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it. Also it eliminates the need for maintaining "side files" that must be kept up to date with changes in source files. Instead the information can be maintained *in* the source file.