Institute for Advanced Computing And Software Development (IACSD)
Akurdi, Pune

Object Oriented Programming with JAVA

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

Java Vs C++

Development wise differences

1. Java is platform independent language but c++ is dependent upon operating system.
At compilation time Java Source code(.java) converts into byte code(.class) .The interpreter translates this byte code at run time into native code and gives output.

2. Java uses both a compiler and interpreter, while C++ only uses a compiler

Syntactical differences

1. There is no final semi-colon at the end of the class definition.
2. Functions are called as methods.
3. main method is a member of class
& has a fixed form
public static void main(String[] args) -- argument is an array of String. This array contains the command-line arguments.
4. main method must be inside some class (there can be more than one main function -- there can even be one in every class)
5. Like the C++ << operator,

To write to standard output, you can use either of the following:

```
System.out.println( ... )
System.out.print( ... )
```

The former prints the given expression followed by a newline, while the latter just prints the given expression.

These functions can be used to print values of any type. eg :

```
System.out.print("hello");   // print a String
System.out.print(16);        // print an integer
System.out.print(5.5 * .2);  // print a floating-point number
```

The + operator can be useful when printing. It is overloaded to work on Strings as follows:
If either operand is a String, it
converts the other operand to a String (if necessary)
creates a new String by concatenating both operands .

Features wise differences.

1. C++ supports pointers whereas Java does not support pointer arithmetic. It supports Restricted pointers.
Java references (Restricted pointers) can't be arithmatically modified.

2. C++ supports operator overloading , multiple inheritance but java does not.

3. C++ is nearer to hardware than Java.

4. Everything (except fundamental or primitive types) is an object in Java (Single root hierarchy as everything gets derived from java.lang.Object).

5.Java is similar to C++ but it doesn't have the complicated aspects of C++, such as pointers, templates, unions, operator overloading, structures, etc. Java also does not support conditional compilation (#ifdef/#ifndef type).

6.Thread support is built into Java but not in C++. C++11, the most recent iteration of the C++ programming language, does have Thread support though.

7.Internet support is built into Java, but not in C++. On the other hand, C++ has support for socket programming which can be used.

8.Java does not support header files and library files. Java uses import to include different classes and methods.

9.Java does not support default arguments.

10.There is no scope resolution operator :: in Java. It has . using which we can qualify classes with the namespace they came from.

11.There is no goto statement in Java.

12.Because of the lack of destructors in Java, exception and auto garbage collector handling is different than C++.

13.Java has method overloading, but no operator overloading unlike C++.

14.The String class does use the + and += operators to concatenate strings and String expressions use automatic type conversion,

16.Java is pass-by-value.

17.Java does not support unsigned integers.

Why java doesn't support c++ copy constructor?
Java does. They're just not called implicitly like they are in C++ .

Firstly, a copy constructor is nothing more than:

```
public class Blah {
  private int foo;

  public Blah() { } // public no-args constructor
  public Blah(Blah b) { foo = b.foo; }  // copy constructor
}
```

Now C++ will implicitly call the copy constructor with a statement like this:

```
Blah b2 = b1;
```

Note:-Cloning/copying in that instance simply makes no sense in Java because all b1 and b2 are references and not value objects like they are in C++. In C++ that statement makes a copy of the object's state.

In Java it simply copies the reference. The object's state is not copied so implicitly calling the copy constructor makes no sense.

All stand-alone C++ programs require a function named main and can have numerous other functions. Java does not have stand alone functions, all functions (called methods) are members of a class. All classes in Java ultimately inherit from the Object class, while it is possible to create inheritance trees that are completely unrelated to one another in C++. In this sense , Java is a pure Object oriented language, while C++ is a mixture of Object oriented and structure language.

The interface keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed(true till Java 8) . C++ does not support interface concept. Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.(death of a diamond)

Java is running on a Virtual Machine, which can recollect unused memory to the operating system, so Java does not destructor. Unlike C++, Java cannot access pointers to do memory operation directly. This leads to a whole host of subtle and extremely important differences between Java and C++.

Furthermore, the C++ compiler does not check whether all local variables are initialized before

they are read. It is quite easy to forget initializing a variable in C++. The value of the variable is then the random bit pattern that happened to be in the memory location that the local variable occupies.

Java does not have global functions and global data. Static in Java is just like global in C++, can be accessed through class name directly, and shared by all instances of the class. For C++, static data members must be defined out side of class definition, because they don't belong
to any specific instance of the class.

Generally Java is more robust than C++ because:

Object handles (references) are automatically initialized to null.
Handles are checked before accessing, and exceptions are thrown in the event of problems.
You cannot access an array out of bounds.
Memory leaks are prevented by automatic garbage collection.
While C++ programmer clearly has more flexibility to create high efficient program, also more chance to encounter error.
About Java Features , development environment

Features of Java

Object Oriented

Everything in Java is coded using OO principles. This facilitates code modularization, reusability, testability, and performance.

Interpreted/Portable

Java source is compiled into platform-independent bytecode, which is then interpreted (compiled into native-code) at runtime. Java code is "Write Once, Run Everywhere"

Simple

Java has a familiar syntax, automatic memory management, exception handling, single inheritance, standardized documentation, and a very rich set of libraries .

Secure/Robust

Due to its support for strong type checking, exception handling, and memory management, Java is immune to buffer- overruns, leaked memory, illegal data access. Additionally, Java comes with a Security Manager too.

Scalable

Java is scalable both in terms of performance/throughput, and as a development environment. A single user can play a Java game on a mobile phone, or millions of users can shop though a Java-based e- commerce enterprise application.

High-performance/Multi-threaded

With its Just-in-Time compiler, Java can achieve (or exceed) performance of native applications. Java supports multi-threaded development out-of-the-box.

Dynamic

Java can load application components at run-time even if it knows nothing about them. Each class has a run-time representation.

Distributed

Java comes with support for networking, as well as for invoking methods on remote (distributed) objects through RMI.

About JVM,JRE,JDK

Java Development Kit [JDK] is the core component of Java Environment and provides all the tools, executable and binaries required to compile, debug and execute a Java Program. JDK is a platform specific software and that's why we have separate installers for Windows, Mac and Unix systems.

Java Virtual Machine[JVM] is the heart of java programming l anguage. When we run a program, JVM is responsible to converting Byte code to the machine specific code. JVM is also platform dependent and provides core java functions like memory management, garbage collection, security etc.

Java Runtime Environment [JRE] is the implementation of JVM, it provides platform to execute java programs. JRE consists of JVM and java binaries and other class libraries to execute any program successfully. To execute any java program, JRE is required.

JVM architecture with journey of java program from source code to execution stage

As shown in the below architecture diagram, JVM subsystems are :

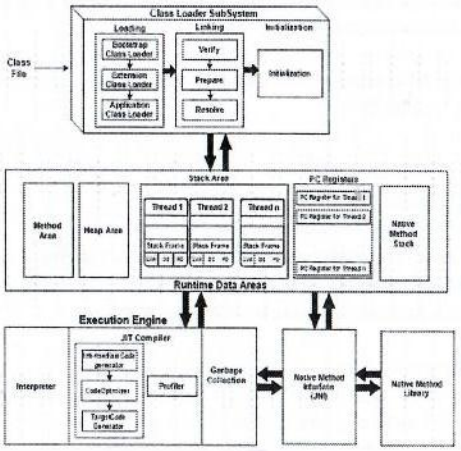
- Class Loader Subsystem
- Runtime Data Area
- Execution Engine

Class Loader Subsystem

Loading : Class loader dynamically loads java classes. It loads, links and initializes the class file when it refers to a class for the first time at runtime, not compile time. Class Loaders follow Delegation Hierarchy Algorithm while loading the class files. 3 types are,

1. Boot Strap Class Loader – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.
2. Extension Class Loader – Responsible for loading classes which are inside the ext folder (jre\lib).
3. Application Class Loader –Responsible for loading Application Level Class path, path

mentioned Environment Variable etc.



Linking : Linking stage involves,

- Verify – Byte code verifier will verify byte code using checksum.
- Prepare – For all static variables memory will be allocated and assigned with default values.
- Resolve – All symbolic memory references are replaced with the original references from Method Area.

Initialization: Here all static variables will be assigned with the original values, and

Runtime Data Area

The Runtime Data Area is divided into 5 major components:

- Method Area – All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- Heap Area – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

- Stack Area – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three sub entities:
  1. Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.
  2. Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
  3. Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.
- PC Registers – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
- Native Method stacks – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

Execution Engine

The byte code which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the byte code and executes it piece by piece.

- Interpreter – The interpreter interprets the byte code faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
- JIT Compiler – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
- Intermediate Code generator – Produces intermediate code
- Code Optimizer – Responsible for optimizing the intermediate code generated above
- Target Code Generator – Responsible for Generating Machine Code or Native Code
- Profiler – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

Garbage Collector: Collects and removes unreferenced objects.

Java Native Interface (JNI): JNI will be interacting with the Native Method

Libraries and provides the Native Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries which is required for the Execution Engine.

Bytecode

Bytecode is in a compiled Java programming language [by javac command] format and has the .class extension executed by Java Virtual Machine (JVM). The Java bytecode gets processed by the Java virtual machine (JVM) instead of the processor. The JVM transforms program code into readable machine language for the CPU because platforms utilize different code interpretation techniques. A JVM converts bytecode for platform interoperability, but bytecode is not platform-specific. JVM is responsible for processing & running the bytecode.

JIT

The magic of java "Write once, run everywhere" is bytecode. JIT improves the performance of Java applications by compiling bytecode to native machine code at run time. JIT is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.
Typical compilers take source code and completely convert it into machine code, JITs take the same source code and convert it into an intermediary "assembly language," which can then be pulled from when it's needed. And that's the key. Assembly code is interpreted into machine code on call—resulting in a faster translation of only the code that you need.
JIT have access to dynamic runtime information and are able to optimize code. JITs monitor and optimize while they run by finding code more often called to make them run better in the future.
JITs reduce the CPU's workload by not compiling everything all at once, but also because the resulting compiled code is optimized for that particular CPU. It's why languages with JIT compilers are able to be so "portable" and run on any platform or OS.

Platform independence

Java is a platform independent programming language, because your source code can be executed on any platform [e.g. Windows, Mac or Linux etc..]. When you install JDK software on your system, JVM

is automatically installed on your system. When we compile Java code then .class file or bytecode is generated by javac compiler. For every operating system separate JVM is available which is capable to read the . class file or byte code and execute it by converting to native code for that specific machine. We compile code once and run everywhere.

Difference between JDK, JRE, and JVM:-

Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.
Now we need an environment to make a run of our program. Henceforth, **JRE** stands for **"Java Runtime Environment"** and may also be written as **"Java RTE."** The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the *Java Virtual Machine (JVM), core classes*, and *supporting files*.
Now let us discuss **JVM**, which stands out for java virtual machines. It is as follows:

- A **specification** where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- An **implementation** is a computer program that meets the requirements of the JVM specification.
- **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Before proceeding to the differences between JDK, JRE, and JVM, let us discuss them in brief first and interrelate them with the image below proposed.

1. JDK (Java Development Kit) is a Kit that provides the environment to **develop and execute(run)** the Java program. JDK is a kit(or package) that includes two things

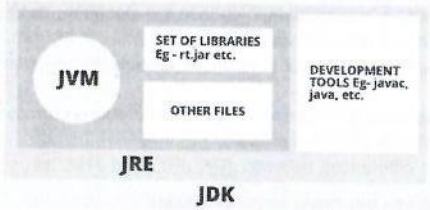
- Development Tools(to provide an environment to develop your java programs)
- JRE (to execute your java program).

2. JRE (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.
3. JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an ***interpreter.***
Now let us discuss the components of JRE in order to understand its importance of it and perceive how it actually works. For this let us discuss components.

The components of JRE are as follows:

1. **Deployment technologies**, including deployment, Java Web Start, and Java Plug-in.
2. **User interface toolkits**, including *Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag, and drop (DnD)*, and *input methods.*

3. **Integration libraries**, including *Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP),* and *scripting.*
4. **Other base libraries**, including *international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization*, and *Java for XML Processing (XML JAXP).*
5. **Lang and util base libraries**, including *lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects*, and *Regular Expressions.*
6. **Java Virtual Machine (JVM)**, including *Java HotSpot Client* and *Server Virtual Machines.*



Language Fundamentals

Sample Java Program

```
public class HelloWorld {
  public static void main(String[]
    args) {System.out.println("Hello
    World!");
```

- All code is contained within a class, in this case HelloWorld.
- The file name must match the class name and have a .java
- extension, for example: HelloWorld.java
- All executable statements are contained within a method, in this
- case named main().
- Use System.out.println() to print text to the terminal.
- Classes and methods (including other flow-control structures) are always defined in

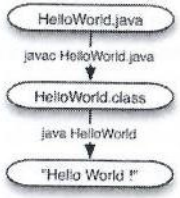
blocks of code enclosed by curly braces ({ }).

- All other statements are terminated with a semi-colon (;).
- Java language is case-sensitive.

Compiling Java Programs

- The JDK comes with a command-line compiler: javac.
- It compiles source code into Java bytecode, which is low-level instruction set similar to binary machine code.
- The bytecode is executed by a Java virtual machine (JVM), rather than a specific physical processor.
- To compile our HelloWorld.java, you could go to the directory containing the source file and execute: javac HelloWorld.java
- This produces the file HelloWorld.class, which contains the Java bytecode.

You can view the generated byte-code, using the -c option to javap, the Java class disassembler. For example: javap -c HelloWorld



To run the bytecode, execute:

java HelloWorld

The main() Method

- A Java application is a public Java class with a main() method.
- The main() method is the entry point into the application.

- The signature of the method is always: public static void main(String[] args)
- Command-line arguments are passed through the args parameter, which is an array of Strings

Basic rules For Writing Java Programs

1. Java compiler doesn't allow accessing of un initialized data members.

2. Files with no public classes(default scoped) can have a name that does not match with any of theclasses in the file .
3. A file can have more than one non public class.
4. There can be only one public class per source code file.
5. If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Example { } must be in a source code file named Example.java.
6. Java compiler doesn't allow accessing of un-inited vars.eg : int n; sop(n);//error

Data Types in Java

Primitive Data Types

Type	Size	Range	Default Value
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	[-263, 263-1]	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Variables

Variable is nothing but identification of memory location.

Types of variables Local

Variables declared inside method are local variable they have scope only within that methods.

Instance

Instance variables are declared inside class but used outside method

Static

Static variable are same as that of instance variable but having keyword static.

They are also called as class variable because they are specified and can be access either class nameor object name.

Type Casting in Java

long And float conversion:-

The range of values that can be represented by a float or double is much larger than the range that can be represented by a long. Although one might lose significant digits when converting from a long to a float, it is still a "widening" operation because the range is wider.

From the Java Language Specification, §5.1.2:

A widening conversion of an int or a long value to float, or of a long value to double, may result in loss of precision - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).

Note that a double can exactly represent every possible int value.

Conversions regarding primitive types

Automatic conversions(also called as widening) ---here it goes for Automatic promotions

byte--->short--->int---> long--->float--->double

char ---> int

eg : char ch='a';

long --->float ---is considered automatic type of conversion(since float data type can hold larger rangeof values than long data type)

Narrowing conversion --- forced conversion(type-casting)eg

double ---> int float --> long double ---> float

Rules for Casting

src & dest - must be compatible, typically dest data type must be able to store larger magnitude ofvalues than that of src data type.

1. Any arithmetic operation involving byte,short --- automatically promoted to --int
2. int & long ---> long
3. long & float ---> float
4. byte,short.....& float & double > double

What is Unicode ?

The Unicode-characters are universal characters encoding standard. It represents way different characters can be represented in different documents like text file, web pages etc.
It is the industry standard designed to consistently and uniquely encode characters used in written languages throughout the world.

The Unicode standard uses hexadecimal to express a character.
For example the value 0x0041 represents A.

The ASCII character set contained limited number of characters. It doesn't have Japanese characters , can't support Devnagari scripts.

The idea behind Unicode was to create a single character set that included every reasonable character in all writing systems in the world.

The Unicode standard was initially designed using 16 bits to encode characters because the primary machines were 16-bit PCs. When the specification for the Java language was created, the Unicode standard was accepted and the char primitive was defined as a 16-bit data type, with characters in the hexadecimal range from 0x0000 to 0xFFFF.

Operators in Java

1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;

*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

3. Java Relational Operators

Relational operators are used to check the relationship between two operands.It returns either trueor false.

Operator	Description	Example
==	Is Equal To	2 == 8 returns false
!=	Not Equal To	2 != 8 returns true
>	Greater Than	2 > 8 returns false
<	Less Than	2 < 8 returns true
>=	Greater Than or Equal To	2 >= 8 returns false
<=	Less Than or Equal To	2 <= 8 returns false

4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decisionmaking.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

5. Java Unary Operators

Unary operators are used with only one operand.

Operator	Meaning
+	Unary plus: not necessary to use since numbers are positive without using it
-	Unary minus: inverts the sign of an expression
++	Increment operator: increments value by 1

--	Decrement operator: decrements value by 1
!	Logical complement operator: inverts the value of a boolean

6. **Java Ternary Operator**

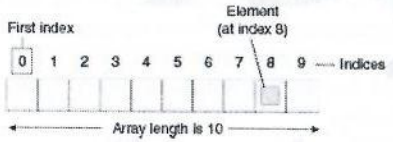
The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example, variable = Expression ? expression1 : expression2

Here,

If the Expression is true, expression1 is assigned to the variable. If the Expression is false, expression2 is assigned to the variable.

Java Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.



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.In Java, array is an object of a dynamically generated class. Java array inherits the Object class.

Types of Array in java

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java Syntax to Declare an Array in Java dataType[] arr; (or)

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

Instantiation of an Array in Java arrVar=new datatype[size];

Multidimensional Array in Java

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

Syntax to Declare Multidimensional Array in Java dataType[][] arrVar; (or)
dataType [][]arrVar; (or)

dataType arrVar [][]; (or)dataType []arrVar [];

Instantiate Multidimensional Array in Java
int[][] arr=new int[3][3];//3 row and 3 column

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

int arr[][] = new int[3][];arr[0] = new int[3]; arr[1] = new int[4]; arr[2] = new int[2];

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos,
Object dest, int destPos, int length)

The two Object arguments specify the array to copy from and the array to copy to. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

for-each Loop for Java Array [Enhanced For Loop]

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.

- It starts with the keyword **for** like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.
- It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop

The syntax of the for-each loop is given below: for(data_type variable:array){
//body of the loop
}

e.g.

```
import java.io.*;
class Demo
{

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

```
{
  // array declaration
   int arr[] = { 10, 50, 60, 80, 90 };
   for (int element : arr)
      System.out.print(element + " ");
  }
}
```

Limitations of for-each loop

decision-making

1.For-each loops are **not appropriate when you want to modify the array**:

```
for (int num : marks)
{
  // only changes num, not the array element
  num = num*2;
}
```

2.For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop

```
for (int num : numbers)
{
  if (num == target)
  {
     return ???;  // do not know the index of num
  }
}
```

3.For-each **only iterates forward over the array in single steps**

```
    // cannot be converted to a for-each loop
for (int i=numbers.length-1; i>0; i--)
{
   System.out.println(numbers[i]);
```



```
}
```

4.For-each **cannot process two decision making statements** at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{
  if (numbers[i] == arr[i])
  { ...
  }
}
```

static Keyword :-

The **static keyword** in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

The *static* keyword is a non-access modifier in Java that is applicable for the following:

1. Blocks
2. Variables
3. Methods
4. Classes

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object

Static initializer block:-

In simpler language whenever we use a **static keyword** and associate it to a block then that block is referred to as a static block. Unlike C++, Java supports a special block, called a static block (also called static clause) that can be used for static initialization of a class. This code inside the static block is executed only once: the first time the class is loaded into memory.

Calling of static block in java?

Now comes the point of how to call this static block. So in order to call any static block, there is no specified way as static block executes automatically when the class is loaded in memory.

syntax --

```
static {
// block gets called only once @ class loading time , by JVM's classlaoder
```

// usage --1. to init all static data members

//& can add functionality -which HAS to be called precisely once.

Use case : singleton pattern , J2EE for loading hibernate/spring... frmwork.

}

They appear -- within class definition & can access only static members directly.(w/o instance)

A class can have multiple static initializer blocks(legal BUT not recommended)

Note:-Static blocks can also be executed before constructors.

e.g.

```
// Java Program to Illustrate Execution of Static
Block
// Before Constructors

Test.java

public class Test {

  // Case 1: Static variable
  static int i;
  // Case 2: Non-static variable
  int j;

  // Case 3: Static blocks
  static
  {
    i = 10;
    System.out.println("static block called ");
  }

  // Constructor calling
  Test() { System.out.println("Constructor
called"); }
}

Tester.java
// Main class
class Tester {

  // Main driver method
  public static void main(String args[])
  {

    // Although we have two objects, static block
is
```



```
    // executed only once.
    Test t1 = new Test();
    Test t2 = new Test();
  }
}
```

Note:-A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

Static variables

When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables:

- We can create static variables at the class level only.
  Unlike C/C++, static local variables are not allowed in Java.
  e.g.

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

 static int fun()
 {
  static int x= 10;  //Error: Static local variables are not allowed
  return x--;
 }
}
```

static block and static variables are executed in the order they are present in a program.

Static methods

When a method is declared with the *static* keyword, it is known as the static method. The most common example of a static method is the *main( )* method. As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

- They can only directly call other static methods.

- They can only directly access static data.
- They cannot refer to this or super in any way.

```
// Java program to demonstrate restriction on static methods

class Test
{
   // static variable
   static int a = 10;

   // instance variable
   int b = 20;

   // static method
   static void m1()
   {
      a = 20;
      System.out.println("from m1");

      // Cannot make a static reference to the non-static field b
      b = 10; // compilation error

      // Cannot make a static reference to the
            // non-static method m2() from the type Test
      m2();  // compilation error

      //  Cannot use super in a static context
      System.out.println(super.a); // compiler error
   }

   // instance method
   void m2()
   {
      System.out.println("from m2");
   }
```

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


Reference Data Types in Java:-

Java provides two types of data types **primitive** and **reference** data type.
The primitive data types are predefined in Java that serves as a fundamental building block while the **reference** data type refers to where data is stored.

what is a reference data type in Java:-

When we create an object (instance) of class then space is reserved in heap memory. Let's understand with the help of an example.

Demo D1 = new Demo();

Now, The space in the heap Memory is created but the question is **how to access that space?**.

Then, We create a Pointing element or simply called **Reference variable** which simply points out the Object(the created space in a Heap Memory).

Understanding Reference variable

1. Reference variable is used to point object/values.

2. Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java. Reference variables hold the objects/values of reference types in Java.

3.Reference variable can also store **null** value.Bydefault, if no object is passed to a reference variable then it will store a null value.

. You can access object members using a reference variable using **dot** syntax.

<reference variable name >.<instance variable_name / method_name>

Constructors in Java:-

Java constructors or constructors in Java is a terminology used to construct something in our programs. A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
In Java, a constructor is a block of codes similar to the method. It is

called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Note: *It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor*
(constructor with no-arguments) if your class doesn't have any.

How Constructors are Different From Methods in Java?

- Constructors must have the same name as the class within which
- it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the
- return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Need of Constructor

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer's memory), then can a box be there with no value defined for its dimensions? The answer is **No**. So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or byJava itself (default constructor).

When Constructor is called?

Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members** of the same class. Rules for writing constructors are as follows:

- The constructor(s) of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static, or Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Constructor Chaining :-

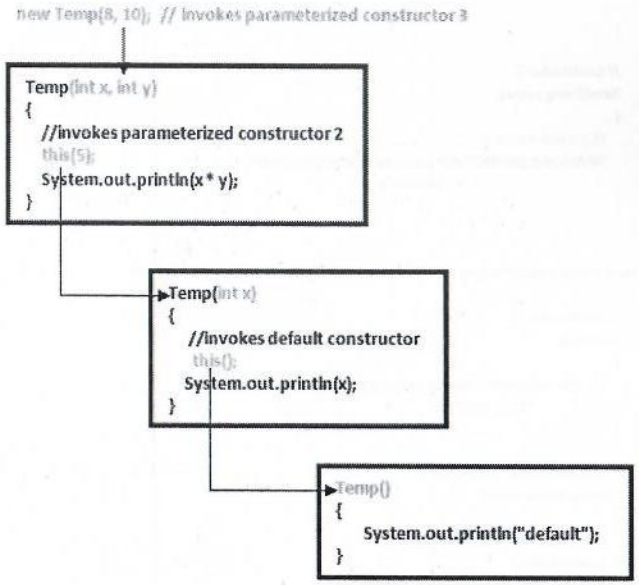
- Constructor chaining is the process of calling one constructor from another constructor with respect to current object.
- One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

Constructor chaining can be done in two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in the same class
- **From base class:** by using **super()** keyword to call the constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub-class constructor's task is to call super class's constructor first. This ensures that the creation of sub class's object starts with the initialization of the data members of the superclass. There could be any number of classes in the inheritance chain. Every constructor calls up the chain till the class at the top is reached.

Why do we need constructor chaining?
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable



Rules of constructor chaining :

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

What happens if we change the order of constructors?
Nothing, Constructor chaining can be achieved in any order

Constructor Chaining to other class using super() keyword :

```
class Base
{
   String name;

   // constructor 1
   Base()
   {
      this("");
      System.out.println("No-argument constructor of" +
                                 " base class");
   }

   // constructor 2
   Base(String name)
   {
      this.name = name;
      System.out.println("Calling parameterized constructor"
                                 + " of base");
   }
}

class Derived extends Base
{
   // constructor 3
   Derived()
   {
      System.out.println("No-argument constructor " +
                  "of derived");
   }

   // parameterized constructor 4
   Derived(String name)
   {
      // invokes base class constructor 2
      super(name);
      System.out.println("Calling parameterized " +
                  "constructor of derived");
   }

   public static void main(String args[])
   {
      // calls parameterized constructor 4
      Derived obj = new Derived("test");

      // Calls No-argument constructor
```



```
      // Derived obj = new Derived();
   }
}
```

Alternative method : using Initializer block : . Inititalizer block is always executed before any constructor, whenever a constructor is used for creating a new object.

The Initializer Block in Java:-

In order to perform any operations while assigning values to an instance data member, an initializer block is used. In simpler terms, the initializer block is used to declare/initialize the common part of various constructors of a class. It runs every time whenever the object is created.

The initializer block contains the code that is always executed whenever an instance is created and it runs each time when an object of the class is created. There are 3 areas where we can use the initializer blocks:

- Constructors
- Methods
- Blocks

```
// Java Program to Illustrate Initializer Block

// Importing required classes
import java.io.*;

// Main class
public class Demo {

    // Initializer block starts..
    {
        // This code is executed before every
constructor.
        System.out.println(
            "Common part of constructors invoked !!");
    }
    // Initializer block ends

    // Constructor 1
    // Default constructor
    public Demo()
    {

        // Print statement
        System.out.println("Default Constructor
invoked");
    }

    // Constructor 2
    // Parameterized constructor
    public Demo(int x)
    {

        // Print statement
        System.out.println(
            "Parameterized constructor invoked");
    }

    // Main driver method
    public static void main(String arr[])
    {

        // Creating variables of class type
        Demo obj1, obj2;

        // Now these variables are
        // made to object and interpreted by compiler
        obj1 = new Demo();

        obj2 = new Demo(0);
    }
}
```


Output:-

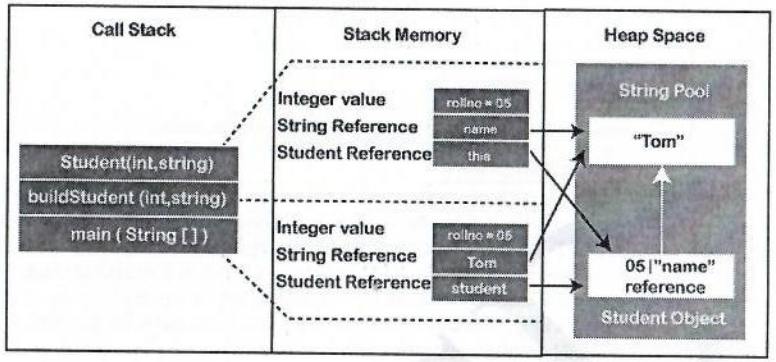
```
Common part of constructors invoked !!

Default Constructor invoked

Common part of constructors invoked !!Parameterized constructor invoked
```

Stack Memory

The stack memory is a physical space (in RAM) allocated to each thread at runtime. It is created when a thread creates. Memory management inthe stack follows LIFO (Last-In-First-Out) order because it is accessible globally. It stores the variables, references to objects, and partial results. Memory allocated to stack lives until the function returns. If there is no space for creating the new objects, it throws the java.lang.StackOverFlowError. The scope of the elements is limited to their threads. The JVM creates a separate stack for each thread

Heap Memory

- It is created when the JVM starts up and used by the application as long
- as the application runs. It stores objects and JRE classes. Whenever we create objects it occupies space in the heap memory while the reference of that object creates in the stack. It does not follow any order like the stack.
- It dynamically handles the memory blocks. It means, we need not to handle the memory manually. For managing the memory automatically, Java provides the garbage collector that deletes the objects which are no longer being used.
- .Memory allocated to heap lives until any one event, either program terminated or memory free does not occur. The elements are globally accessible in the application.
- It is a common memory space shared with all the threads. If the heap space is full, it throws the java.lang.OutOfMemoryError.
- The heap memory is further divided into the following memory areas:
  Young generation
  Survivor space
  Old generation
  Permanent generation
  Code Cache



Object Oriented Concepts in Java

Classes in Java

It is the logical construct upon which the entire Java language is built because it defines the shapeand nature of an object.

As such, the class forms the basis for object-oriented programming in Java.

Any concept you wish to implement in a Java program must be encapsulated within a class.

a class isthat it defines a new data type.

Once defined, this new type can be used to create objects of that type.

Thus, a class is a template for an object, and an object is an instance of a class. Everything in Java isdefined in a class.

When you define a class, you declare its exact form and nature. You do this by specifying the

data that it contains and the code that operates on that data.

In its simplest form, a class just defines a collection of datae.g.

```
class Box {
        //data members
        double height,width,depth;

        double calVolume(); //methods
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class

Objects in Java

Because an object is an instance of a class, you will often see the two words object and instance usedinterchangeably.

To create an object (instance) of a particular class, use the new operator, followed by an

invocationof a constructor for that class, such as:

```
new Box();
```

The constructor method initializes the state of the new object.

The new operator returns a reference to the newly created object.

As with primitives, the variable type must be compatible with the value type when using objectreferences, as in:

```
Box b1 = new Box();
```

To access member data or methods of an object, use the dot (.) notation: variable.field or variable.method().

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

Capsule

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Inheritance

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

- Inheritance allows you to define a class based on the definition of another class.
- The class it inherits from is called a base class or a parent class.
- The derived class is called a subclass or child class.
- Subject to any access modifiers, which we'll discuss later, the subclass gets access to the fields and methods defined by the base class.
- The subclass can add its own set of fields and methods to the set it inherits from its parent.
- Inheritance simplifies modeling of real-world hierarchies through generalization of common features.
- Common features and functionality is implemented in the base classes, facilitating code reuse.
- Subclasses can extended, specialize, and override base class functionality.

Inheritance provides a means to create specializations of existing classes. This is referred to as sub- typing. Each subclass provides specialized state and/or behavior in addition to the state and behavior inherited by the parent class. For example, a manager is also an employee but it has a responsibility over a department, whereas a generic employee does not.

- You define a subclass in Java using the extends keyword followed by the base class name. For example:

```
class Manager extends Employee
{
// mgr class members
}
```

- In Java, a class can extend at most one base class. That is, multiple inheritance is not supported.

- If you don't explicitly extend a base class, the class inherits from Java's Object class, discussed later in this module.
- Java supports multiple levels of inheritance.

For example, Child can extend Parent, which in turn extends Grandparent, and so on.

polymorphism:-

Difference between Compile-time and Run-time Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. In this article, we will see the difference between two types of polymorphisms, compile time and run time.

Compile Time Polymorphism: Whenever an object is bound with its functionality at the compile time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by
checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading. Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology and which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time. The following is an example where compile-time polymorphism can be observed.

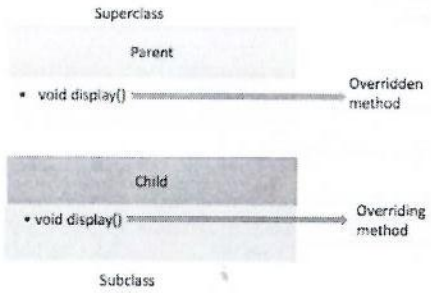
```
public class Demo {

    // First addition function
    public static int add(int a, int b)
    {
        return a + b;
    }

    // Second addition function
    public static double add(
        double a, double b)
    {
        return a + b;
    }

    // Driver code
    public static void main(String args[])
    {
        // Here, the first addition
        // function is called
        System.out.println(add(2, 3));

        // Here, the second addition
        // function is called
        System.out.println(add(2.0, 3.0));
    }
}
```

Run-Time Polymorphism: Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. Java virtual machine determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says the child class has the same method as declared in the parent class. It means if the child class provides the specific implementation of the method that has been provided by one of its parent classes then it is known as method overriding. The following is an example where runtime polymorphism can be observed.


```
class Test {

    // Implementing a method
    public void method()
    {
        System.out.println("Method 1");
    }
}

// Defining a child class
public class Demo extends Test {

    // Overriding the parent method
    public void method()
    {
        System.out.println("Method 2");
    }

    // Driver code
    public static void main(String args[])
    {
        Test test = new Demo();

        test.method();
    }
}
```

Rules of overriding and overloading of methods:-

1. **In java, only inherited methods can be overridden.** This is because method overriding happens only when an instance method of the parent class is re-implemented in the child class with the same signature. For example:




```
class Parent
{
  int a = 15, b = 30;
  void display()
  {
    int c = a+b;
    System.out.println("Sum = "+c);
  }
}

public class Child extends Parent
{
  void display()
  {
    int d = a*b;
    System.out.println("Product = " + d);
  }

  public static void main(String args[])
  {
    Child child = new Child();
    child.display();
  }
}
```

Output:
Product=450

In the above example, the display() method of the parent class is re-implemented in the child class with the same name and signature. Therefore, the display()method of the parent class is overridden by the display() method of the child class.

2. **final and static methods cannot be overridden in java.** This is because

final methods can't be re-implemented in the sub-class of a super-class. Also, only instance methods can be overridden.

```
class Parent
{
  int a = 15, b = 30;
  final void display()
  {
    int c = a+b;
    System.out.println("Sum = "+c);
  }
}

class Child extends Parent
{
  void display()
  {
    super.display();
    int d = a*b;
    System.out.println("Product = " + d);
  }
}

public class JavaHungry
{
  public static void main(String args[])
  {
    Child ob = new Child();
    ob.display();
  }
}
```

Output:

```
/JavaHungry.java:13: error: display() in Child cannot override display() in Parent
  void display()
       ^
 overridden method is final
1 error
```

In this example, the **final method** display() is re-implemented in child class which is not allowed in java. Therefore the compiler will throw an error.

3.The overriding method must have the same argument list. If there is any change in the argument list of the overriding method then it is called method overloading and not method overriding. For example:

```
class Parent
{
```

```
    void display()
    {
      System.out.println("Display method with no arguments");
    }
}

class Child extends Parent
{
    void display(int value)
    {
      System.out.println("Display method with one argument: "+value);
    }
}

public class JavaHungry
{
    public static void main(String args[])
    {
      Child ob = new Child();
      ob.display();
      ob.display(1);
    }
}
```

Output:
Display method with no arguments
Display method with one argument: 1

As the display() method of child class has one argument whereas the display () method of parent class has no arguments, this is called method overloading and not method overriding.

4.The overriding method must have the same return type (or subtype) as that of the overridden method. If the return type(or subtype) of the overriding method is not the same as that of the overridden method, the program will not compile successfully. For example:

```
class Parent
{
    int a = 15;
    int display()
    {
      int b = a*a;
      return b;
    }
}

class Child extends Parent
{
    float display()
    {
```



```
      float d = a/2;
      return d;
    }
}

public class JavaHungry
{
    public static void main(String args[])
    {
      Child ob = new Child();
      float num = ob.display();
      System.out.println(num);
    }
}
```

This code will not compile successfully.

Output:

```
/JavaHungry.java:13: error: display() in Child cannot override display() in Parent
    float display()
          ^
  return type float is not compatible with int
1 error
```

5. **In Java, the overriding method must not have a more restrictive access modifier**. It means that the overriding method must have equal or lesser restrictive access. For better understanding, have a look over the following rules:
a. If the overridden method has default access then the overriding method must be the default, protected or public.
b. If the overridden method is protected then the overriding method must be public or protected.
c. If the overridden method is public then the overriding method must be public.

```
class Parent
{
    public void display()
    {
      System.out.println("Inside Parent Class.");
    }
}

class Child extends Parent
{
    protected void display()
    {
      System.out.println("Inside Child Class");
    }
}

public class JavaHungry
{
```

```
  public static void main(String args[])
  {
    Child obj = new Child();
    obj.display();
  }
}
```

Output:

```
/JavaHungry.java:11: error: display() in Child cannot override display() in Parent
  protected void display()
            ^
 attempting to assign weaker access privileges; was public
1 error
```

This type of overriding is not allowed in Java as the overriding method has more restrictive access than the overridden method.

6. **The overriding method must not throw new or broader checked exceptions.** It means if the overriding method throws a narrower exception or **unchecked exception** then the function overriding becomes legal.

```
import java.io.*;

class Parent
{
  public void display() throws IOException
  {
    System.out.println("Inside Parent Class.");
  }
}

class Child extends Parent
{
  public void display() throws FileNotFoundException
  {
    System.out.println("Inside Child Class");
  }
}

public class JavaHungry
{
  public static void main(String args[])
  {
    Child obj = new Child();
    try
    {
      obj.display();
    }
    catch(IOException e)
    {
```



```
      e.printStackTrace();
    }
  }
}
```

Output:
Inside Child Class

This is a legal overriding in Java as the overriding method throws FileNotFoundException which is a subclass of IOException.

7. **The overridden method can be called from a subclass using the super keyword.**

```
class Parent
{
  public void display()
  {
    System.out.println("Inside Parent Class");
  }
}

class Child extends Parent
{
  public void display()
  {
    super.display();
    System.out.println("Inside Child Class");
  }
}

public class JavaHungry
{
  public static void main(String args[])
  {
    Child c = new Child();
    c.display();
  }
}
```

Output:
Inside Parent Class
Inside Child Class

8. **Overriding of constructors are not allowed.** This is because constructors are not methods and constructors of sub-class and super-class can't be the same.

9. **Abstract methods must be overridden by the first non-abstract(concrete) class.**

```
abstract class Parent
{
  abstract void display();
}

class Child extends Parent
{
  public void display()
  {
    System.out.println("Inside Child Class");
  }
}

public class JavaHungry
{
  public static void main(String args[])
  {
    Child c = new Child();
    c.display();
  }
}
```

Output:
Inside Child Class

10. synchronized and strictfp modifier does not have any effect on the rules of method overriding.

Rules for Method Overloading in Java

1. In Java, method overloading can only be achieved if and only if two or more methods share the same method name but have different method signatures.
2. If more than one method has the same name as well as signature within the same class, the program would not compile successfully. For example:

```
class Overload
{
  void disp()
  {
    System.out.println("No argument");
  }

  void disp(int x)
  {
    System.out.println(x);
  }
  void disp(int x)
  {
    System.out.println("Overloaded Method");
  }
```



```
}

public class JavaHungry
{
  public static void main(String a[])
  {
    Overload object = new Overload();
    object.disp();
    object.disp(1);
    object.disp(2);
  }
}
```

Output:

```
/JavaHungry.java:12: error: method disp(int) is already defined in class Overload
  void disp(int x)
       ^
1 error
```

2. **Method return type has no contribution in method overloading.** If our program consists of more than one method with the same name, same method signature but different return types, the compiler will consider it as the duplicate declarations for the same method. Hence, the program will not compile successfully. For example:

```
class Overload
{
  int calc(int x)
  {
    int y = x*x;
    return y;
  }

  float calc(int x)
  {
    float y = x/2;
    return y;
  }
}

public class JavaHungry
{
  public static void main(String args[])
  {
    Overload object = new Overload();
    int a = object.calc(1);
    float b = object.calc(3);
    System.out.println(a);
    System.out.println(b);
```

```
}
}
```

Output:

```
/JavaHungry.java:9: error: method calc(int) is already defined in class Overload
   float calc(int x)
         ^
1 error
```

That's all for today, please mention in comments in case you have any questions related to rules for method overriding and method overloading in java with code examples.

super and this keywords in Java

In java, **super** keyword is used to access methods of the **parent class** while **this** is used to access methods of the **current class**.

this keyword is a reserved keyword in java i.e, we can't use it as an identifier. It is used to refer current class's instance as well as static members. It can be used in various contexts as given below:

- to refer instance variable of current class
- to invoke or initiate current class constructor
- can be passed as an argument in the method call
- can be passed as argument in the constructor call
- can be used to return the current class instance

super keyword

1. **super** is a reserved keyword in java i.e, we can't use it as an identifier.
2. **super** is used to refer **super-class's instance as well as static members**.
3. **super** is also used to invoke **super-class's method or constructor**.
4. **super** keyword in java programming language refers to the superclass of the class where the super keyword is currently being used.
5. The most common use of **super** keyword is that it eliminates the confusion between the superclasses and subclasses that have methods with same name.

super can be used in various contexts as given below:

- it can be used to refer immediate parent class instance variable
- it can be used to refer immediate parent class method
- it can be used to refer immediate parent class constructor.

Similarities in this and super

- We can use *this* as well as *super* **anywhere except static area**. Example of this is already shown above where we use this as well as super inside public static void main(String[]args) hence we get Compile Time Error since cannot use them inside static area.
- We can use *this* as well as *super* **any number of times in a program**.
- Both are **non-static** keyword.

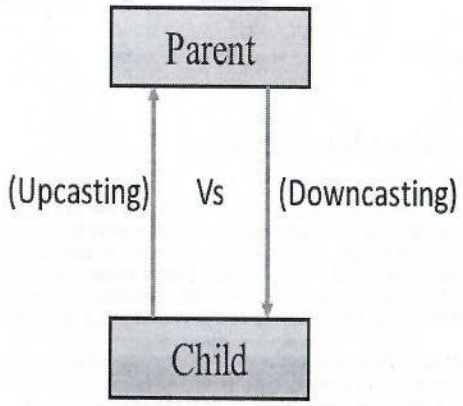
***Note:** We can use 'this' as well as 'super' any number of times but main thing is that we **cannot** use them inside static context.*

Upcasting Vs Downcasting in Java:-

Typecasting is one of the most important concepts which basically deals with the conversion of one data type to another datatype implicitly or explicitly.

In this article, the concept of typecasting for objects is discussed.
Just like the data types, the objects can also be typecasted. However, in objects, there are only two types of objects, i.e. parent object and child object. Therefore, typecasting of objects basically means that one type of object (i.e.) child or parent to another. There are two types of typecasting. They are:

1. **Upcasting:** Upcasting is **the typecasting of a child object to a parent object**. Upcasting can be done implicitly. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.
2. **Downcasting:** Similarly, downcasting means the typecasting of a **parent object to a child object**. Downcasting cannot be implicit.



Example: Let there be a parent class. There can be many children of a parent. Let's take one of the children into consideration. The child inherits the properties of the parent. Therefore, there is an "is-a" relationship between the child and parent. Therefore, the child can be implicitly **upcasted** to the parent. However, a parent may or may not inherits the child's properties. However, we can forcefully cast a parent to a child which is known as **downcasting**. After we define this type of casting explicitly, the compiler checks in the background if this type of casting is possible or not. If it's not possible, the compiler throws a ClassCastException.

Syntax of Upcasting:

```
Parent p = new Child();
```

Upcasting will be done internally and due to upcasting the object is allowed to access only parent class members and child class specified members (overridden methods, etc.) but not all members.

```
// This variable is not
// accessible
p.id = 1;
```

Syntax of Downcasting:

```
Child c = (Child)p;
```

Downcasting has to be done externally and due to downcasting a child object can acquire the properties of the parent object.

```
c.name = p.name;
```

i.e., c.name = "IACSD"

Covariant Return Types in Java:-

As the ear hit eardrums "overriding" we quickly get to know that it can be done either virtue of different datatypes or arguments passed to a function what a programmer learned initially while learning polymorphism in java. Before JDK 5.0, it was not possible to override a method by changing the return type. When we override a parent class method, the name, argument types, and return type of the overriding method in child class has to be exactly the same as that of the parent class method. The overriding method was said to be **invariant** with respect to return type.
Java version 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child's return type should be a subtype of the parent's return type. The overriding method becomes **variant** with respect to return type.

- It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
- We get the liberty to have more specific return types when overriding methods.
- Help in preventing run-time ClassCastExceptions on returns

Note: *If we swap return types of Base and Derived, then above program would not work. Please see this program for example.*

```
// Java Program to Demonstrate Different Return
Types
// if Return Type in Overridden method is Sub-type

// Class 1
class A {
}

// Class 2
class B extends A {
}

// Class 3
// Helper class (Base class)
class Base {

  // Method of this class of class1 return type
  A fun()
  {
    // Display message only
    System.out.println("Base fun()");

    return new A();
  }
}

// Class 4
// Helper class extending above class
class Derived extends Base {

  // Method of this class of class1 return type
  B fun()
  {
    // Display message only
    System.out.println("Derived fun()");

    return new B();
  }
}

// Class 5
// Main class
public class Demo {

  // Main driver method
  public static void main(String args[])
  {

    // Creating object of class3 type
    Base base = new Base();

    // Calling method fun() over this object
```

```
// inside main() method
    base.fun();

    // Creating object of class4 type
    Derived derived = new Derived();

    // Again calling method fun() over this object
    // inside main() method
    derived.fun();
  }
}
```

Note:-In object-oriented programming, downcasting refers to the act of casting a reference of a base class to one of its derived classes. This operation can lead to runtime errors if the object being casted is not of the expected derived class type.

The issue with downcasting can be resolved with the help of covariant return types. Covariant return types allow a method in a subclass to return a more specific type than the method in the parent class. This means that a method that returns a type "T" in the parent class can be overridden to return a more specific type "S" in the subclass, as long as "S" is a subclass of "T".

With covariant return types, you can ensure that the downcasted object is of the expected derived class type. By allowing the overridden method to return a more specific type, you can access the properties and methods of the derived class, without having to perform an explicit downcast operation.

In summary, using covariant return types can help to resolve issues with downcasting in object-oriented programming. By allowing a method in a subclass to return a more specific type, you can avoid runtime errors and ensure that the downcasted object is of the expected derived class type.

Abstract Class in Java

Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the "abstract" keyword in its class definition.

```
abstract class Shape
{
    int color;
    // An abstract function
    abstract void draw();
}
```


In Java, the following some *important observations* about abstract classes are as follows:

An instance of an abstract class can not be created.

Constructors are allowed.

We can have an abstract class without any abstract method.

There can be a **final method** in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final"

We can define static methods in an abstract class

We can use the **abstract keyword** for declaring ***top-level classes (Outer class) as well as inner classes*** as abstract

If a **class** contains at least **one abstract method** then compulsory should declare a class as abstract

If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method

```
// Abstract class
abstract class Sunstar {
    abstract void printInfo();
}

// Abstraction performed using extends
class Employee extends Sunstar {
    void printInfo() {
        String name = "avinash";
        int age = 21;
        float salary = 222.2F;

        System.out.println(name);
        System.out.println(age);
        System.out.println(salary);

    }

}

// Base class
class Base {
    public static void main(String args[]) {
        Sunstar s = new Employee();
        s.printInfo();
    }
}
```

Properties of Abstract class

1.In Java, just like in C++ an instance of an abstract class cannot be created, we can have references to abstract class type though. It is as shown below via the clean java program.

```
abstract class Base {
   abstract void fun();
}

// Class 2
class Derived extends Base {
   void fun()
   {
      System.out.println("Derived fun() called");
   }
}

// Class 3
// Main class
class Main {

   // Main driver method
   public static void main(String args[])
   {

      // Uncommenting the following line will cause
      // compiler error as the line tries to create an
      // instance of abstract class. Base b = new Base();
      // We can have references of Base type.
      Base b = new Derived();
      b.fun();
   }
}
```


2.Like C++, an **abstract class** can contain **constructors** in Java. And a constructor of an abstract class is called when an instance of an inherited class is created. It is as shown in the program below as follows:

```
abstract class Base {

   // Constructor of class 1
   Base()
   {
      // Print statement
      System.out.println("Base Constructor Called");
   }

   // Abstract method inside class1
   abstract void fun();
}

// Class 2
class Derived extends Base {

   // Constructor of class2
   Derived()
   {
      System.out.println("Derived Constructor Called");
   }

   // Method of class2
   void fun()
   {
      System.out.println("Derived fun() called");
   }
}

// Class 3
// Main class
class Demo {

   // Main driver method
   public static void main(String args[])
   {
      // Creating object of class 2
      // inside main() method
      Derived d = new Derived();
      d.fun();
   }
}
```

3. In Java, we can have ***an abstract class without any abstract method***. This allows us to ***create classes that cannot be instantiated but can only be inherited***.

```
// Java Program to illustrate Abstract class
// Without any abstract method

// Class 1
// An abstract class without any abstract method
abstract class Base {

    // Demo method. This is not an abstract method.
    void fun()
    {
        // Print message if class 1 function is called
        System.out.println("Function of Base class is called");
    }
}

// Class 2
class Derived extends Base {
  //This class only inherits the Base class methods and
properties
}

// Class 3
class Main {

    // Main driver method
    public static void main(String args[])
    {
        // Creating object of class 2
        Derived d = new Derived();

        // Calling function defined in class 1 inside main()
        // with object of class 2 inside main() method
        d.fun();
    }
}
```


*4.Abstract classes can also have **final** methods* (methods that cannot be overridden)

Example:

```
//Java Program to Illustrate Abstract classes
// Can also have Final Methods

// Class 1
// Abstract class
abstract class Base {

    final void fun()
    {
        System.out.println("Base fun() called");
    }
}

// Class 2
class Derived extends Base {

}

// Class 3
// Main class
class Demo {

    // Main driver method
    public static void main(String args[])
    {
        {
            // Creating object of abstract class

            Base b = new Derived();
            // Calling method on object created above
            // inside main method

            b.fun();
        }
    }
}
```

5.For any abstract java class we are not allowed to create an object i.e., for an abstract class instantiation is not possible.

```
// Java Program to Illustrate Abstract Class

// Main class
// An abstract class
abstract class Demo {

    // Main driver method
    public static void main(String args[])
    {

        // Trying to create an object
        Demo d = new Demo();
    }}
```

6.Similar to the interface ***we can define static methods in an abstract class*** that ***can be called independently without an object.***

```
// Java Program to Illustrate
// Static Methods in Abstract
// Class Can be called Independently

// Class 1
// Abstract class
abstract class Helper {

    // Abstract method
    static void demofun()
    {

        // Print statement
        System.out.println("Welcome in IACSD");
    }
}

// Class 2
// Main class extending Helper class
public class Demoextends Helper {

    // Main driver method
    public static void main(String[] args)
    {

        // Calling method inside main()
        // as defined in above class
        Helper.demofun();
    }
}
```

7.We can use the **abstract keyword** for declaring top-level classes (Outer class) as well as inner classes as abstract.

```
import java.io.*;

abstract class B {
    // declaring inner class as abstract with abstract
    // method
    abstract class C {
        abstract void myAbstractMethod();
    }
}
class D extends B {
    class E extends C {
        // implementing the abstract method
        void myAbstractMethod()
        {
            System.out.println(
                "Inside abstract method implementation");
        }
    }
```



```
}
public class Main {

    public static void main(String args[])
    {
        // Instantiating the outer class
        D outer = new D();

        // Instantiating the inner class
        D.E inner = outer.new E();
        inner.myAbstractMethod();
    }
}
```

8.If a **class contains at least one abstract method** then **compulsory that we should declare the class as abstract** otherwise we will get a compile-time error ,If a class contains at least one abstract method then, implementation is not complete for that class, and hence it is not recommended to create an object so in order to restrict object creation for such partial classes we use **abstract keyword.**

```
import java.io.*;

// here if we remove the abstract
// keyword then we will get compile
// time error due to abstract method
abstract class Demo {
    abstract void m1();
}

class Child extends Demo {
    public void m1()
    {
        System.out.print("Hello");
    }
}
class Test {
    public static void main(String[] args)
    {
        Child c = new Child();
        c.m1();
    }
}
```

9.If the **Child** class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

```
// Java Program to demonstrate
// Observation
import java.io.*;

abstract class Demo {
   abstract void m1();
   abstract void m2();
   abstract void m3();
}

abstract class FirstChild extends Demo {
   public void m1() {
     System.out.println("Inside m1");
   }
}

class SecondChild extends FirstChild {
   public void m2() {
     System.out.println("Inside m2");
   }
   public void m3() {
     System.out.println("Inside m3");
   }
}

class Demo {
   public static void main(String[] args)
   {
      // if we remove the abstract keyword from FirstChild
      // Class and uncommented below obj creation for
      // FirstChild then it will throw
      // compile time error as did't override all the
      // abstract methods

      // FirstChild f=new FirstChild();
      // f.m1();

      SecondChild s = new SecondChild();
      s.m1();
      s.m2();
      s.m3();
   }
}
```


Interfaces in Java:-

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.
The interface in Java is *a* mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.When we decide a type of entity by its behaviour and not via attribute we should define it as an interface.
Like a class, an interface can have methods and variables, but the methods declared

in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the behaviour.
- Interface do not have constructor.
- Represent behaviour as interface unless every sub-type of the class is guarantee to have that behaviour.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax:

```
interface {

   // declare constant fields

   // declare methods that abstract

   // by default.

}
```

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

Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement infinite number of interface.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

// A simple interface

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

Difference Between Class and Interface

The major differences between a class and an interface are:

S. No.	Class	Interface
1.	In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
2.	Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
3.	The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Implementation: **To implement an interface we use the keyword** implements


```
// Java program to demonstrate working of
// interface

import java.io.*;

// A simple interface
interface In1 {

   // public, static and final
   final int a = 10;

   // public and abstract
   void display();
}

// A class that implements the interface.
class TestClass implements In1 {

   // Implementing the capabilities of
   // interface.
   public void display(){
    System.out.println("IACSD");
   }

   // Driver Code
   public static void main(String[] args)
   {
      TestClass t = new TestClass();
      t.display();
      System.out.println(a);
   }
}
```

Important Points About Interface:-

- We can't create an instance(interface can't be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extend to another interface or interface (more than one interface).
- A class that implements the interface must implement all the methods in the interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritances.
- It is used to achieve loose coupling.
- Inside the Interface not possible to declare instance variables because by default variables are **public static final.**
- Inside the Interface constructors are not allowed.
- Inside the interface main method is not allowed.
- Inside the interface static ,final,private methods declaration are not possible.

Nested Interface in Java:-

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface. Interface in a class Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer this for details). This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as c_name.i_name where c_name is the name of the class in which it is nested and i_name is the name of the interface itself. Let us have a look at the following code:-

```
// Java program to demonstrate working of
// interface inside a class.
import java.util.*;
class Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

class A
{
   public static void main(String[] args)
   {
      Test.Yes obj;
      Testing t = new Testing();
      obj=t;
      obj.show();
   }
}
```

Note:-

The access specifier in above example is default. We can assign public, protected or private also. Below is an example of protected. In this particular example, if we change access specifier to private, we get compiler error because a derived class tries to access it.


```
// Java program to demonstrate protected
// specifier for nested interface.
import java.util.*;
class Test
{
   protected interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

class A
{
   public static void main(String[] args)
   {
      Test.Yes obj;
      Testing t = new Testing();
      obj=t;
      obj.show();
   }
}
```

Interface in another Interface An interface can be declared inside another interface also. We mention the interface as **i_name1.i_name2** where **i_name1** is the name of the interface in which it is nested and **i_name2** is the name of the interface to be implemented.

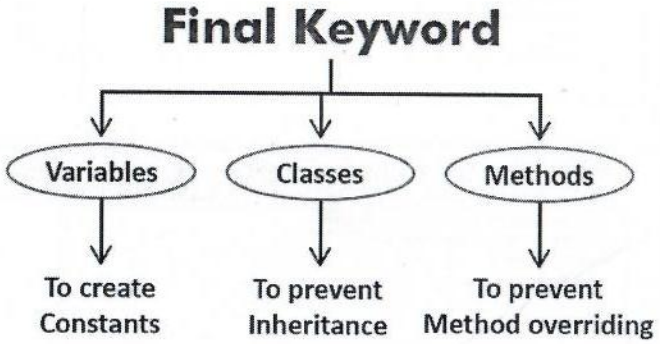
```
// Java program to demonstrate working of
// interface inside another interface.
import java.util.*;
interface Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

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

```
{
    Test.Yes obj;
    Testing t = new Testing();
    obj = t;
    obj.show();
  }
}
```

Note: In the above example, access specifier is public even if we have not written public. If we try to change access specifier of interface to anything other than public, we get compiler error. Remember, interface members can only be public..

final Keyword in Java

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable only to a variable, a method, or a class. The following are different contexts where final is used.



Final Variables

When a variable is declared with the ***final keyword***, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable.

If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

```
final int THRESHOLD = 5;

// Final variable

final int THRESHOLD;

// Blank final variable

static final double PI = 3.141592653589793;

// Final static variable PI

static final double PI;

// Blank final static  variable
```

Initializing a final Variable

We must initialize a final variable, otherwise, the compiler will throw a compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable: 1.You can initialize a final variable when it is declared. This approach is the most common. A final variable is called a **blank final variable** if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable. 2.A blank final variable can be initialized inside an instance-initializer block or inside the constructor.3. If you have more than one constructor in your class then it must be initialized in all of them, otherwise, a compile-time error will be thrown.

A blank final static variable can be initialized inside a static block.

```
class Demo {

   // a final variable
   // direct initialize
   final int THRESHOLD = 5;

   // a blank final variable
   final int CAPACITY;

   // another blank final variable
   final int  MINIMUM;

   // a final static variable PI
   // direct initialize
   static final double PI = 3.141592653589793;

   // a  blank final static  variable
   static final double EULERCONSTANT;

   // instance initializer block for
   // initializing CAPACITY
   {
      CAPACITY = 25;
   }

   // static initializer block for
   // initializing EULERCONSTANT
   static{
      EULERCONSTANT = 2.3;
   }

   // constructor for initializing MINIMUM
   // Note that if there are more than one
   // constructor, you must initialize MINIMUM
   // in them also
   public Demo()
   {
      MINIMUM = -1;
   }

}
```

Note:- ***a final variable cannot be reassign, doing it will throw compile-time error.***


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

Output:Compile Time Error

Final Methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. The Object class does this—a number of its methods are final. We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes.

Illustration: Final keyword with a method

```
class A
{
   final void m1()
   {
      System.out.println("This is a final method.");
   }
}

class B extends A
{
   void m1()
   {
      // Compile-error! We can not override
      System.out.println("Illegal!");
   }
}
```

Final classes

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited).
There are two uses of a final class:
Usage 1: One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float, etc. are final classes. We can not extend them.

```
final class A
{
    // methods and fields
}
// The following class is illegal
class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

Usage 2: The other use of final with classes is to create an immutable class like the predefined String class. One can not make a class immutable without making it final.

enum in Java

Enumerations serve the purpose of representing a group of named constants in a programming language.
A Java enumeration is a class type. Although we don't need to instantiate an enum using **new,** it has the same capabilities as other classes. This fact makes Java enumeration a very powerful tool. Just like classes, you can give them constructor, add instance variables and methods, and even implement interfaces.
One thing to keep in mind is that, unlike classes, enumerations neither inherit other classes nor can get extended(i.e become superclass).

In Java (from 1.5), enums are represented using **enum** data type. Java enums are more powerful than C/C++ enums. In Java, we can also add variables, methods, and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types

Declaration of enum in Java: Enum declaration can be done outside a Class or inside a Class but not inside a Method.


```
//Outside Class
enum Color {
   RED,
   GREEN,
   BLUE;
}

public class Test {
   // Driver method
   public static void main(String[] args)
   {
      Color c1 = Color.RED;
      System.out.println(c1);
   }
}
```

```
//Inside Class
public class Test {
   enum Color {
      RED,
      GREEN,
      BLUE;
   }

   // Driver method
   public static void main(String[] args)
   {
      Color c1 = Color.RED;
      System.out.println(c1);
   }
}
```

Important Points of enum:

- Every enum is internally implemented by using Class.

```
/* internally above enum Color is converted to

class Color
{
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}*/
```

- Every enum constant represents an **object** of type enum.
- enum type can be passed as an argument to **switch** statements.

Enum and Inheritance:

- All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.

- **toString() method** is overridden in **java.lang.Enum class**, which returns enum constant name.
- enum can implement many interfaces.

values(), ordinal() and valueOf() methods:

- These methods are present inside **java.lang.Enum**.
- **values() method** can be used to return all values present inside the enum.
- Order is important in enums.By using the **ordinal() method**, each enum constant index can be found, just like an array index.
- **valueOf() method** returns the enum constant of the specified string value if exists.

Note:-
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable

ie. they are comparable and serializable implicitly.
All enum types in java are singleton by default.
So, you can compare enum types using '==' operator also.

Since enums extends java.lang.Enum, so they can not extend any other class because java does not support multiple inheritance . But, enums can implement any number of interfaces.

enum can be declared within a class or separately

Access Modifiers in Java

Access modifiers in Java help to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- **Default**: When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.
  - The data members, classes, or methods that are not declared using
  - any access modifiers i.e. having default access modifier are accessible **only within the same package**.

```
package p1;

// Class Show is having Default access modifier
class Show
{
   void display()
   {
      System.out.println("Hello World!");
   }
}
```

```
package p2;
import p1.*;

// This class is having default access
modifier
class Demo
{
   public static void main(String args[])
   {
      // Accessing class Show from package
p1
      Show obj = new Show();

      obj.display();
   }
}
```

Output:-Compile time error

- **Private**: The private access modifier is specified using the keyword **private**.
  - The methods or data members declared as private are accessible only **within the class** in which they are declared.
  - Any other **class of the same package will not be able to access** these members.
  - Top-level classes or interfaces can not be declared as private because
    1. private means "only visible within the enclosing class".
    2. protected means "only visible within the enclosing class and any subclasses"

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

```
package p1;

class A
{
private void display()
  {
    System.out.println("Welcome in IACSD");
  }
}

class B
{
public static void main(String args[])
  {
    A obj = new A();
    // Trying to access private method
    // of another class
    obj.display();
  }
}
```

Output:-error: display() has private access in A

obj.display();

- **protected**: The protected access modifier is specified using the keyword **protected**. The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

```
package p1;

// Class A
public class A
{
protected void display()
  {
    System.out.println("Welcome in IACSD");
  }
}

package p2;
import p1.*; // importing all classes in package
p1

// Class B is subclass of A
class B extends A
{
public static void main(String args[])
{
  B obj = new B();
  obj.display();
}

}
```

- **public**: The public access modifier is specified using the keyword **public**.
  - The public access modifier has the **widest scope** among all other access modifiers.

- Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

Packages In Java

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

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

All we need to do is put related classes into packages. After that, we can simply write

an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
We can reuse existing classes from the packages as many time as we need it in our program.

How packages work?

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

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

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

util is a subpackage created inside **java** package.
Example of Subpackage

```
package com.javatpoint.core;
class Simple{
 public static void main(String args[]){
  System.out.println("Hello subpackage");
 }
```

```
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Note:-The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

Accessing classes inside a package

Consider following two statements :

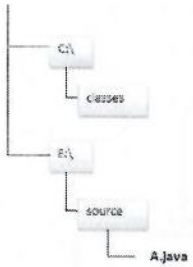
```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;

// Only mentioned class of this package will be accessible.
import package.classname;

// Class name is generally used when two packages have the same
// class name. For example in below code both packages have
// date class so using a fully qualified name to avoid conflict
import java.util.Date;
import my.package.Date;
```

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
//save as Simple.java
package mypack;
public class Simple{
public static void main(String args[]){
    System.out.println("Welcome to package");
    }
}
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

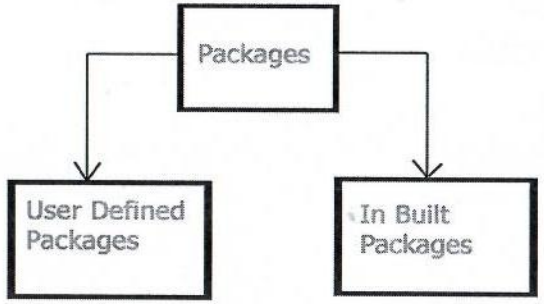
To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

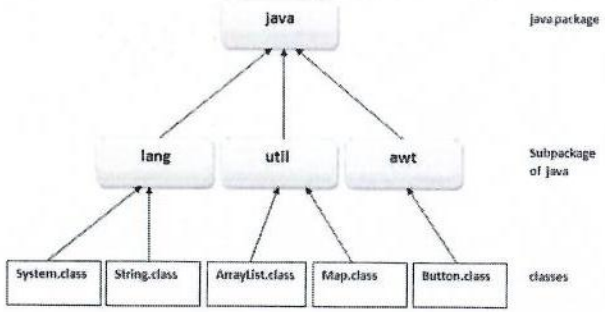
Types of packages:



Built-in Packages

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

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6) **java.net:** Contain classes for supporting networking operations.



User-defined packages

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

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;


public class MyClass
{
   public void getNames(String s)
   {
      System.out.println(s);
   }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;


public class PrintName
{
  public static void main(String args[])
  {
     // Initializing the String variable
     // with a value
     String name = "IACSD";

     // Creating an instance of class MyClass in
     // the package.
     MyClass obj = new MyClass();

     obj.getNames(name);
  }
}
```

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

Static import in Java

In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. Math.sqrt(), but by using static import we can access sqrt() method directly. According to SUN microSystem, it will improve the code readability and enhance coding. But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should not go for static import.

Difference between import and static import:

- With the help of import, we are able to access classes and interfaces which are present in any package. But using static import, we can access all the static members (variables and methods) of a class directly without explicitly calling class name.
- The main difference is Readability, ClassName.dataMember (System.out) is
- less readable when compared to dataMember(out), static import can make
- your program more readable

Garbage Collection in Java

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

What is Garbage Collection?

In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors**.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**. The garbage collector is the best example of the Daemon thread as it is always running in the background.

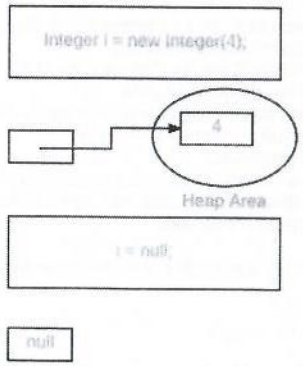
How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Important Concepts Related to Garbage Collection in Java

1. Unreachable objects: An object is said to be unreachable if it doesn't contain any reference to it. Also, note that objects which are part of the island of isolation are also unreachable.

```
Integer i = new Integer(4);

// the new Integer object is reachable  via the reference in 'i'

i = null;

// the Integer object is no longer reachable.
```



2. Eligibility for garbage collection: An object is said to be eligible for GC(garbage collection) if it is unreachable. After *i = null*, integer object 4 in the heap area is suitable for garbage collection in the above image.

Ways to make an object eligible for Garbage Collector

- Even though the programmer is not responsible for destroying useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
- There are generally four ways to make an object eligible for garbage collection.
    1. Nullifying the reference variable
    2. Re-assigning the reference variable
    3. An object created inside the method
    4. Island of Isolation

Ways for requesting JVM to run Garbage Collector

- Once we make an object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage
- Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
- We can also request JVM to run Garbage Collector. There are two ways to do It.
    1. **Using *System.gc()* method:** System class contain static method *gc()* for requesting JVM to run Garbage Collector.

2. **Using *Runtime.getRuntime().gc()* method:** Runtime class allows
3. the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.
4. There is no guarantee that any of the above two methods will run Garbage Collector.
5. The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()*

Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on
- the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.
- *finalize()* method is present in Object class with the following prototype.

protected void finalize() throws Throwable

Based on our requirement, we can override *finalize()* method for performing our cleanup activities like closing connection from the database.

1. The *finalize()* method is called by Garbage Collector, not JVM. However,
2. Garbage Collector is one of the modules of JVM.
3. Object class *finalize()* method has an empty implementation. Thus, it is recommended to override the *finalize()* method to dispose of system resources or perform other cleanups.
4. The *finalize()* method is never invoked more than once for any object.
5. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored, and the finalization of that object terminates.

we have to request gc(garbage collector), and for this, we have to write the following

3 steps before closing brace of sub-block.

1. Set references to null(i.e X = Y = null;)
2. Call, System.gc();
3. Call, System.runFinalization();

For Example:-

```
class Employee {

   private int ID;
   private String name;
   private int age;
   private static int nextId = 1;

   // it is made static because it
   // is keep common among all and
   // shared by all objects
   public Employee(String name, int age)
   {
      this.name = name;
      this.age = age;
      this.ID = nextId++;
   }
   public void show()
   {
      System.out.println("Id=" + ID +
"\nName=" + name
                  + "\nAge=" + age);
   }
   public void showNextId()
   {
      System.out.println("Next employee id
will be="
                  + nextId);
   }
   protected void finalize()
   {
      --nextId;
      // In this case,
      // gc will call finalize()
      // for 2 times for 2 objects.
   }
}

public class UseEmployee {
   public static void main(String[] args)
   {
      Employee E = new Employee("EMP1",
56);
      Employee F = new Employee("EMP2",
45);
      Employee G = new Employee("EMP3",
25);
      E.show();
      F.show();
      G.show();
      E.showNextId();
      F.showNextId();
      G.showNextId();

      {
         // It is sub block to keep
         // all those interns.
         Employee X = new Employee("EMP4",
23);
         Employee Y = new Employee("EMP5",
21);
         X.show();
         Y.show();
         X.showNextId();
         Y.showNextId();
         X = Y = null;
         System.gc();
         System.runFinalization();
      }
      E.showNextId();
   }
}
```

Wrapper Classes in Java

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a

wrapper class object. Let's check on the wrapper classes in java with examples:

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish
2. to modify the arguments passed into a method (because primitive types are passed by value).
3. The classes in java.util package handles only objects and hence wrapper
4. classes help in this case also.
5. Data structures in the Collection framework, such as ArrayList and Vector, s
6. tore only objects (reference types) and not primitive types.
7. An object is needed to support synchronization in multithreading.

Below are given examples of wrapper classes in java with their corresponding Primitive data types in java.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

Autoboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer,

long to Long, double to Double, etc.

Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a

wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

```
// Java program to demonstrate Wrapping and UnWrapping
// in Classes
import java.io.*;

class Demo {
    public static void main(String[] args)
    {
        // byte data type
        byte a = 1;

        // wrapping around Byte object
        Byte byteobj = new Byte(a);

        // int data type
        int b = 10;

        // wrapping around Integer object
        Integer intobj = new Integer(b);

        // float data type
        float c = 18.6f;

        // wrapping around Float object
        Float floatobj = new Float(c);

        // double data type
        double d = 250.5;

        // Wrapping around Double object
        Double doubleobj = new Double(d);

        // char data type
        char e = 'a';

        // wrapping around Character object
        Character charobj = e;

        // printing the values from objects
        System.out.println(
            "Values of Wrapper objects (printing as objects)");
        System.out.println("\nByte object byteobj: "
                           + byteobj);
        System.out.println("\nInteger object intobj: "
                           + intobj);
        System.out.println("\nFloat object floatobj: "
                           + floatobj);
        System.out.println("\nDouble object doubleobj: "
                           + doubleobj);
        System.out.println("\nCharacter object charobj: "
                           + charobj);

        // objects to data types (retrieving data types from
        // objects) unwrapping objects to primitive data
        // types
        byte bv = byteobj;
```

```
    int iv = intobj;
    float fv = floatobj;
    double dv = doubleobj;
    char cv = charobj;

    // printing the values from data types
    System.out.println("\nUnwrapped values (printing as data
types)");
    System.out.println("\nbyte value, bv: " + bv);
    System.out.println("\nint value, iv: " + iv);
    System.out.println("\nfloat value, fv: " + fv);
    System.out.println("\ndouble value, dv: " + dv);
    System.out.println("\nchar value, cv: " + cv);
  }
}
```

Strings in Java

Strings are the type of objects that can store the character of values. A string acts the same as an array of characters in Java.

Ways of Creating a String

There are two ways to create a string in Java:

- String Literal
- Using new Keyword

Syntax:
<String_Type> <string_variable> = "<sequence_of_string>";

1. String literal

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Example:
String s = "Welcome in IACSD";

2. Using new keyword

- String s = new String("Welcome");
- In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in the heap (non-pool)

Example:
String s = new String ("Welcome in IACSD");

Interfaces and Classes in Strings in Java

CharBuffer: This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.
String: It is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

CharSequence Interface

CharSequence Interface is used for representing the sequence of Characters in Java. Classes that are implemented using the CharSequence interface are mentioned below:

1. String
2. StringBuffer
3. StringBuilder

1. StringBuffer

StringBuffer is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
Syntax:
StringBuffer s = new StringBuffer("GeeksforGeeks");

2. StringBuilder

StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.
Syntax:
StringBuilder str = new StringBuilder();

str.append("GFG");

3. StringTokenizer

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

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

Above we saw we can create a string by String Literal.

String s="Welcome";

Here the JVM checks the String Constant Pool. If the string does not exist, then a new string instance is created and placed in a pool. If the string exists, then it will not create a new object. Rather, it will return the reference to the same instance. The cache that stores these string instances is known as the String Constant pool or String Pool. In earlier versions of Java up to JDK 6 String pool was located inside PermGen(Permanent Generation) space. But in JDK 7 it is moved to the main heap area.

Immutable String in Java

In Java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once a string object is created its data or state can't be changed but a new string object is created.

String Constant Pool

String is a sequence of characters. One of the most important characteristics of a string in Java is that they are immutable. In other words, once created, the internal state of a string remains the same throughout the execution of the program. This immutability is achieved through the use of a special string constant pool in the heap. In this article, we will understand about the storage of the strings.

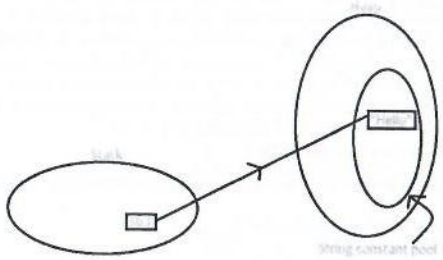
A string constant pool is a separate place in the heap memory where the values of all the strings which are defined in the program are stored. When we declare a string, an object

of type String is created in the stack, while an instance with the value of the string is

created in the heap. On standard assignment of a value to a string variable, the variable is

allocated stack, while the value is stored in the heap in the string constant pool. For example, let's assign some value to a string str1. In java, a string is defined and the value

is assigned as:String str1 = "Hello";The following illustration explains the memory allocation for the



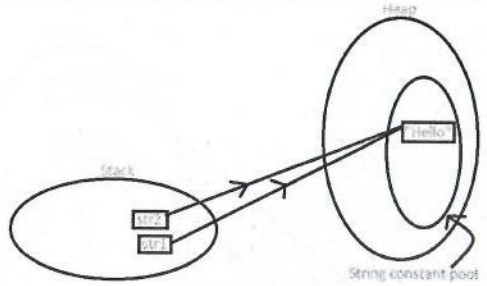
In the above scenario, a string object is created in the stack, and the value "Hello" is created and stored in the heap. Since we have normally assigned the value, it is stored in the constant pool area of the heap. A pointer points towards the value stored in the heap from the object in the stack. Now,

let's take the same example with multiple string variables having the same value as follows:

String str1 = "Hello";

String str2 = "Hello";

The following illustration explains the memory allocation for the above declaration:



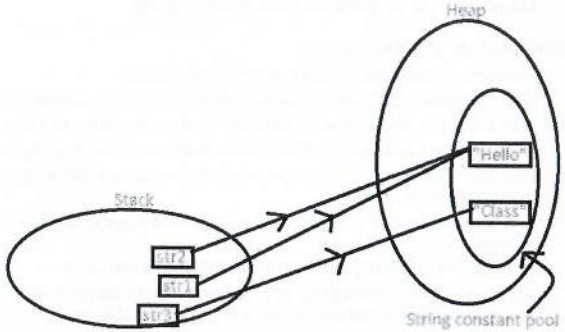
In this case, both the string objects get created in the stack, but another instance of the value "Hello" is not created in the heap. Instead, the previous instance of "Hello" is re-used. The *string constant pool* is a small cache that resides within the heap. Java stores all the *values* inside the string constant pool on direct allocation. This way, if a similar value needs to be accessed again, a new string object created in the stack can reference it directly with the help of a pointer. In other words, the string constant pool exists mainly to reduce memory usage and improve the re-use of existing instances in memory. When a string object is assigned a different value, the new value will be registered in the string constant pool as a separate instance. Lets understand this with the following example:

String str1 = "Hello";

String str2 = "Hello";

String str3 = "Class";

The following illustration explains the memory allocation for the above declaration:

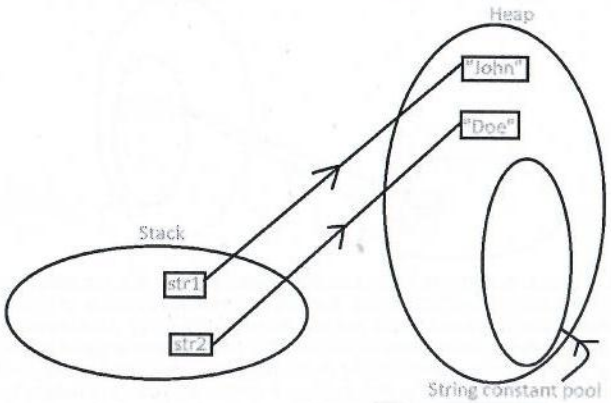


One way to skip this memory allocation is to use the new keyword while creating a new string object. The 'new' keyword forces a new instance to always be created regardless of whether the same value was used previously or not. Using 'new' forces the instance to be created in the heap outside the string constant pool which is clear, since caching and re-using of instances isn't allowed here. Let's understand this with an example:

String str1 = new String("John");

String str2 = new String("Doe");

The following illustration explains the memory allocation for the above declaration:



String and StingBuilder

String

- Automatically created with double quotes
- Pooled (interned) to save memory
- Immutable — once created cannot change
- Concatenated using the expensive + operator
- Many methods for string manipulation/searching

StringBuilder and StringBuffer

- Mutable — can quickly grow and shrink as needed
- Few methods modifying the buffer: append(), insert(), delete(), replace()
- Use toString() if you need to get a String representation of the object's content
- StringBuffer is synchronized for thread-safe access in multithreaded applications
- Use StringBuilder for better performance unless you plan to access the object from multiple threads
- In a single-threaded context (as is generally the case), StringBuilder is faster than StringBuffer.

Difference Between StringBuffer and StringBuilder in Java

Strings in Java are the objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created. However, java provides multiple classes through which strings can be used. Two such classes are StringBuffer and StringBuilder. In this article, we will see the difference between both the classes.

StringBuffer Class: StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically

grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. In order to create a string buffer, an object needs to be created, (i.e.), if we wish to create a new string buffer with name str, then:
StringBuffer str = new StringBuffer();

Example: The following is an example which implements the StringBuffer class

```
/ Java program to demonstrate
// the StringBuffer class

public class Demo {

    // Driver code
    public static void main(String args[])
    {

        // Creating a new StringBuffer
        StringBuffer str
            = new StringBuffer("Hello");

        str.append(" World!");

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

StringBuilder Class: Similar to StringBuffer, the StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters. The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. Similar to StringBuffer, in order to create a new string with the name str, we need to create an object of StringBuilder, (i.e.):
StringBuilder str = new StringBuilder();

Example: The following is an example which implements the StringBuilder class.

```
// Java program to demonstrate
// the StringBuilder class

public class Demo {

    // Driver code
    public static void main(String args[])
    {

        // Creating a new StringBuilder
        StringBuilder str
            = new StringBuilder("Hello");

        str.append(" World!");

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

Conversion from StringBuffer to StringBuilder

The StringBuffer cannot be directly converted to StringBuilder. We first need to convert the StringBuffer to a String object by using the inbuilt method *toString()*. After converting it to a string object, we can simply create a StringBuilder using the constructor of the class. For example:

```
// Java program to demonstrate
// the conversion between the
// StringBuffer and StringBuilder
// class

public class Demo {

    // Driver code
    public static void main(String args[])
    {
        StringBuffer sbr
            = new StringBuffer("IACSD");

        // Conversion from StringBuffer
        // object to the String object
        String str = sbr.toString();

        // Creating a new StringBuilder
        // using the constructor
        StringBuilder sbl
            = new StringBuilder(str);

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

Conversion from StringBuilder to StringBuffer

Similar to the above conversion, the StringBuilder cannot be converted to the StringBuffer directly. We first need to convert the StringBuilder to the String object by using the inbuilt method *toString()*. Now, we can create a StringBuilder using the constructor. For example:

```
// Java program to demonstrate
// the conversion between the
// StringBuilder and StringBuffer
// class

public class Demo {

    // Driver code
    public static void main(String args[])
    {
        StringBuilder sbr
            = new StringBuilder("IACSD");

        // Conversion from StringBuilder
        // object to the String object
        String str = sbr.toString();

        // Creating a new StringBuffer
        // using the constructor
        StringBuffer sbl
```



```
            = new StringBuffer(str);

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

Exceptions in Java

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

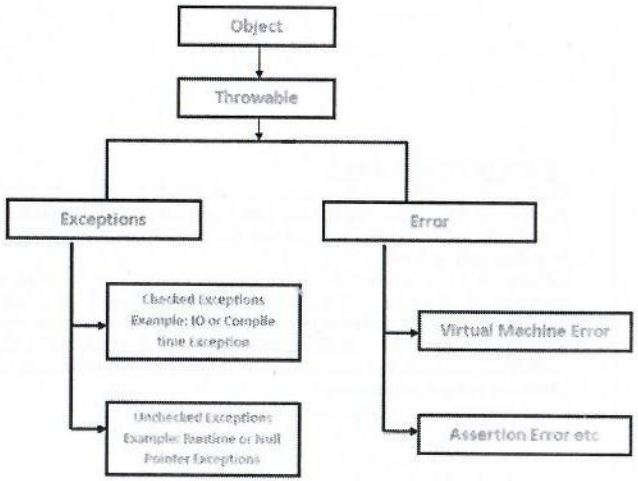
Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

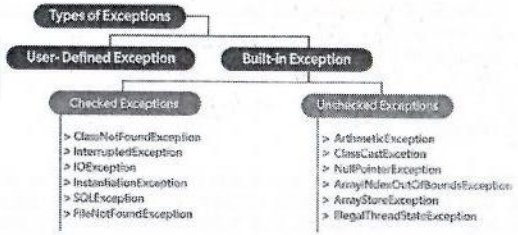
Exception Hierarchy

All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

A. Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see Checked vs Unchecked Exceptions*

B. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

Methods to print the Exception information:

1.printStackTrace()– This method prints exception information in the format of Name of the exception: description of the exception, stack

```
int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        e.printStackTrace();
      }
```

2.toString() – This method prints exception information in the format of Name of the exception: description of the exception

```
int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        System.out.println(e.toString());
      }
```

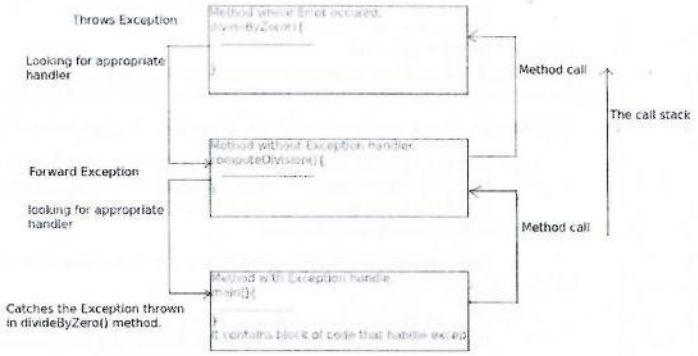
3.getMessage() -This method prints only the description of the exception.

```
int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        System.out.println(e.getMessage());
      }
    }
```

How Does JVM handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred, and proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.



The call stack and searching the call stack for exception handler.

How Programmer Handles an Exception?

Customized Exception Handling: Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.
***Tip:** One must go through control flow in try catch finally block for better understanding.*

Need for try-catch clause(Customized Exception Handling)

Consider the below program in order to get a better understanding of the try-catch clause.

How to Use the try-catch Clause?

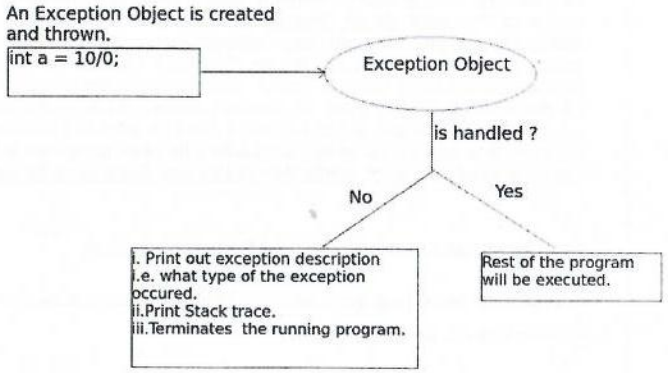
```
try {
   // block of code to monitor for errors
   // the code you think can raise an exception
} catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
} catch (ExceptionType2 exOb) {
   // exception handler for ExceptionType2
}
// optional
finally { // block of code to be executed after try block ends
}
```

Certain below key points are needed to be remembered that are as follows:

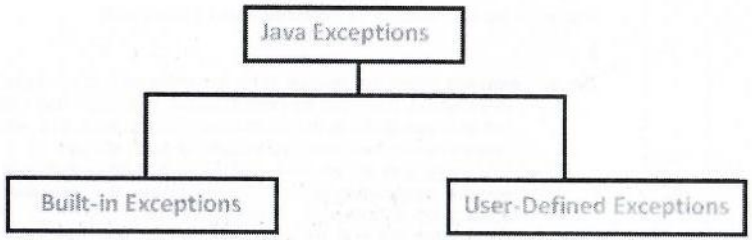
- In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a **catch** block after it. There can be more than one exception handlers. Each **catch** block is an exception handler that handles the exception to the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
- For each try block, there can be zero or more catch blocks, but **only one** final block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after **try and catch blocks.** And if an exception does not occur, then it will be executed after

the **try** block. The finally block in java is used to put important codes such as clean up code e.g., closing the file or closing the connection.

The summary is depicted via visual aid below as follows



Types of Exception in Java with Examples



Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, the user can also create exceptions which are called 'user-defined Exceptions'.

The following steps are followed for the creation of a user-defined Exception.

- The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

```
MyException(String str)
{
  super(str);
}
```

- To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

```
MyException me = new MyException("Exception details");
```

throw me;

- The following program illustrates how to create your own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
- If it is so, then MyException is raised and a message is displayed "Balance amount is less".

```
// Java program to demonstrate user defined exception

// This program throws an exception whenever balance
// amount is below Rs 1000
class MyException extends Exception
{
    //store account information
    private static int accno[] = {1001, 1002, 1003, 1004};

    private static String name[] =
                {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

    private static double bal[] =
         {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

    // default constructor
    MyException() {    }

    // parameterized constructor
    MyException(String str) { super(str); }

    // write main()
    public static void main(String[] args)
    {
        try  {
            // display the heading for the table
            System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                                             "\t" + "BALANCE");

            // display the actual account information
            for (int i = 0; i < 5 ; i++)
            {
                System.out.println(accno[i] + "\t" + name[i] +
                                                  "\t" + bal[i]);

                // display own exception if balance < 1000
                if (bal[i] < 1000)
                {
                    MyException me =
                        new MyException("Balance is less than 1000");
                    throw me;
                }
            }
        } //end of try
```



```
        catch (MyException e) {
            e.printStackTrace();
        }
    }
}
```

throw and throws in Java

throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Syntax:

throw *Instance*

Example:

throw new ArithmeticException("/ by zero");

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class. Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

throws

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax:

type method_name(parameters) throws exception_list

exception_list is a comma separated list of all the exceptions which a method might throw.

In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception XXX must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways:

1. By using try catch
2. By using **throws** keyword

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

Important points to remember about throws keyword:

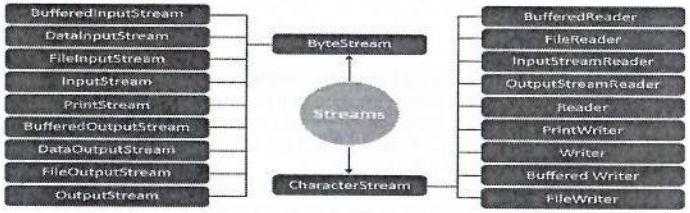
- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

- By the help of throws keyword we can provide information to the caller of the method about the exception.

File Handling in Java

Types of Streams in Java IO

The Streams in Java IO are of the following types:



Why File Handling is Required?

- File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.
- In simple words, file handling means reading and writing data to a file.

Streams in Java

- In Java, a sequence of data is known as a stream.
- This concept is used to perform I/O operations on a file.
- There are two types of streams :

1. Input Stream:

The Java InputStream class is the superclass of all input streams. The input stream is used to read data from numerous input devices like the keyboard, network, etc. InputStream is an abstract class, and because of this, it is not useful by itself. However, its subclasses are used to read data.

There are several subclasses of the InputStream class, which are as follows:

1. AudioInputStream
2. ByteArrayInputStream
3. FileInputStream
4. FilterInputStream
5. StringBufferInputStream
6. ObjectInputStream

Creating an InputStream

// Creating an InputStream

InputStream obj = new FileInputStream();

Here, an input stream is created using FileInputStream.

Note: *We can create an input stream from other subclasses as well as InputStream.*

Methods of InputStream

S No.	Method	Description
1	read()	Reads one byte of data from the input stream.
2	read(byte[] array)()	Reads byte from the stream and stores that byte in the specified array.
3	mark()	It marks the position in the input stream until the data has been read.
4	available()	Returns the number of bytes available in the input stream.
5	markSupported()	It checks if the mark() method and the reset() method is supported in the stream.
6	reset()	Returns the control to the point where the mark was set inside the stream.
7	skips()	Skips and removes a particular number of bytes from the input stream.
8	close()	Closes the input stream.

2. Output Stream:

The output stream is used to write data to numerous output devices like the monitor, file, etc. OutputStream is an abstract superclass that represents an output stream. OutputStream is an abstract class and because of this, it is not useful by itself. However, its subclasses are used to write data.

There are several subclasses of the OutputStream class which are as follows:

1. ByteArrayOutputStream
2. FileOutputStream
3. StringBufferOutputStream
4. ObjectOutputStream
5. DataOutputStream
6. PrintStream

Creating an OutputStream

// Creating an OutputStream

OutputStream obj = new FileOutputStream();

Here, an output stream is created using FileOutputStream.

Note: *We can create an output stream from other subclasses as well as OutputStream.*
Methods of OutputStream

S. No.	Method	Description
1.	write()	Writes the specified byte to the output stream.
2.	write(byte[] array)	Writes the bytes which are inside a specific array to the output stream.
3.	close()	Closes the output stream.
4.	flush()	Forces to write all the data present in an output stream to the destination.

Based on the data type, there are two types of streams :

1. Byte Stream:

This stream is used to read or write byte data. The byte stream is again subdivided into two types which are as follows:

- **Byte Input Stream:** Used to read byte data from different devices.
- **Byte Output Stream:** Used to write byte data to different devices.

2. Character Stream:

This stream is used to read or write character data. Character stream is again subdivided into 2 types which are as follows:

- **Character Input Stream:** Used to read character data from different devices.
- **Character Output Stream:** Used to write character data to different devices.

Owing to the fact that you know what a stream is, let's polish up File Handling in Java by further understanding the various methods that are useful for performing operations on the files like creating, reading, and writing files.

Java File Class Methods

The following table depicts several File Class methods:

Method Name	Description	Return Type
canRead()	It tests whether the file is readable or not.	Boolean
canWrite()	It tests whether the file is writable or not.	Boolean
createNewFile()	It creates an empty file.	Boolean
delete()	It deletes a file.	Boolean

Method Name	Description	Return Type
exists()	It tests whether the file exists or not.	Boolean
length()	Returns the size of the file in bytes.	Long
getName()	Returns the name of the file.	String
list()	Returns an array of the files in the directory.	String[]
mkdir()	Creates a new directory.	Boolean
getAbsolutePath()	Returns the absolute pathname of the file.	String

Let us now get acquainted with the various file operations in Java.

File operations in Java

The following are the several operations that can be performed on a file in Java :

- **Create a File**
- **Read from a File**
- **Write to a File**
- **Delete a File**

Now let us study each of the above operations in detail.

1. Create a File

- In order to create a file in Java, you can use the createNewFile() method.
- If the file is successfully created, it will return a Boolean value true and false if the file already exists.

```
// Import the File class
import java.io.File;

// Import the IOException class to handle errors
import java.io.IOException;

public class Demo {
  public static void main(String[] args)
  {

    try {
      File Obj = new File("myfile.txt");
      if (Obj.createNewFile()) {
        System.out.println("File created: "
                           + Obj.getName());
      }
      else {
        System.out.println("File already exists.");
      }
    }
    catch (IOException e) {
```

```
            System.out.println("An error has occurred.");
            e.printStackTrace();
        }
    }
}
```

2.Read from a File: We will use the Scanner class in order to read contents from a file. Following is a demonstration of how to read contents from a file in Java.

```
try {
        File Obj = new File("myfile.txt");
        Scanner Reader = new Scanner(Obj);
        while (Reader.hasNextLine()) {
            String data = Reader.nextLine();
            System.out.println(data);
        }
        Reader.close();
    }
    catch (FileNotFoundException e) {
        System.out.println("An error has occurred.");
        e.printStackTrace();
    }
```

3.Write to a File: We use the FileWriter class along with its write() method in order to write some text to the file. Following is a demonstration of how to write text to a file in Java :

```
try {
    FileWriter Writer
        = new FileWriter("myfile.txt");
    Writer.write(
        "Files in Java are seriously good!!");
    Writer.close();
    System.out.println("Successfully written.");
}
catch (IOException e) {
    System.out.println("An error has occurred.");
    e.printStackTrace();
}
```

4. **Delete a File:** We use the delete() method in order to delete a file. Following is a demonstration of how to delete a file in Java :

```
File Obj = new File("myfile.txt");
if (Obj.delete()) {
    System.out.println("The deleted file is : "
                + Obj.getName());
}
else {
    System.out.println(
        "Failed in deleting the file.");
}
```

File handling in Java using FileWriter and FileReader

Java FileWriter and FileReader classes are used to write and read data from text files (they are Character Stream classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

FileWriter

FileWriter is useful to create a file writing characters into it.

- This class inherits from the OutputStream class.
- The constructors of this class assume that the default character encoding and
- the default byte-buffer size are acceptable. To specify these values yourself,
- construct an OutputStreamWriter on a FileOutputStream.

- FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
- FileWriter creates the output file if it is not present already.

Constructors:

- **FileWriter(File file) –** Constructs a FileWriter object given a File object.
- **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
- **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
- **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
- **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

Methods:

- **public void write (int c) throws IOException –** Writes a single character.
- **public void write (char [] stir) throws IOException –** Writes an array of characters.
- **public void write(String str)throws IOException –** Writes a string.
- **public void write(String str, int off, int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is the number of characters to write.
- **public void flush() throws IOException** flushes the stream
- **public void close() throws IOException** flushes the stream first and then closes the writer.

Reading and writing take place character by character, which increases the number of I/O operations and affects the performance of the system.**BufferedWriter** can be used along with FileWriter to improve the speed of execution.

The following program depicts how to create a text file using FileWriter

```
/ Creating a text File using FileWriter
import java.io.FileWriter;
import java.io.IOException;
class CreateFile
{
   public static void main(String[] args) throws IOException
   {
      // Accept a string
      String str = "File Handling in Java using "+
            " FileWriter and FileReader";

      // attach a file to FileWriter
      FileWriter fw=new FileWriter("output.txt");
```

```
        // read character wise from string and write
        // into FileWriter
        for (int i = 0; i < str.length(); i++)
            fw.write(str.charAt(i));

        System.out.println("Writing successful");
        //close the file
        fw.close();
    }
}
```

FileReader

FileReader is useful to read data in the form of characters from a 'text' file.

- This class inherited from the InputStreamReader Class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
- FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Constructors:

- **FileReader(File file)** – Creates a FileReader , given the File to read from
- **FileReader(FileDescripter fd)** – Creates a new FileReader , given the FileDescripter to read from
- **FileReader(String fileName)** – Creates a new FileReader , given the name of the file to read from

Methods:

- **public int read () throws IOException** – Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
- **public int read(char[] cbuff) throws IOException** – Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
- **public abstract int read(char[] buff, int off, int len) throws IOException** –Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
  Parameters:
  cbuf – Destination buffer
  off – Offset at which to start storing characters
  len – Maximum number of characters to read
- **public void close() throws IOException** closes the reader.
- **public long skip(long n) throws IOException** –Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.
  Parameters:
  n – The number of characters to skip

The following program depicts how to read from the 'text' file using FileReader

// Reading data from a file using FileReader


```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
class ReadFile
{
    public static void main(String[] args) throws IOException
    {
        // variable declaration
        int ch;

        // check if File exists or not
        FileReader fr=null;
        try
        {
            fr = new FileReader("text");
        }
        catch (FileNotFoundException fe)
        {
            System.out.println("File not found");
        }

        // read from FileReader till the end of file
        while ((ch=fr.read())!=-1)
            System.out.print((char)ch);

        // close the file
        fr.close();
    }
}
```

Java.io.BufferedReader Class in Java

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes. In general, each read request made by a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. Programs that use DataInputStreams for textual input can be localized by replacing each **DataInputStream** with an appropriate **BufferedReader**.

Constructors of BufferedReader Class

Constructor	Action Performed
BufferedReader(Reader in)	Creates a buffering character-input stream that uses a default-sized input buffer
BufferedReader(Reader in, int sz)	Creates a buffering character-input stream that uses an input buffer of the specified size.

Methods of BufferedReader Class

Method Name	Action
close()	Closes the stream and releases any system resources associated with it.Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.
mark()	Marks the present position in the stream. Subsequent calls to reset() will attempt to reposition the stream to this point.
markSupported()	Tells whether this stream supports the mark() operation, which it does.
read()	Reads a single character.
read(char[] cbuf, int off, int len)	Reads characters into a portion of an array. This method implements the general contract of the corresponding read method of the Reader class. As an additional convenience, it attempts to read as many characters as possible by repeatedly invoking the read method of the underlying stream.
readLine()	Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a line feed.
ready()	Tells whether this stream is ready to be read.
reset()	Resets the stream to the most recent mark.
skip(long)	Skips characters.

Java.io.BufferedWriter class methods in Java

Bufferreader class writes text to character-output stream, buffering characters.Thus, providing efficient writing of single array, character and strings. A buffer size needs to be specified, if not it takes Default value.
An output is immediately set to the underlying character or byte stream by the Writer.
Class Declaration
public class BufferedWriter

extends Writer

Constructors

- **BufferedWriter(Writer out)**: Creates a buffered character-output stream that uses a default-sized output buffer.
- **BufferedWriter(Writer out, int size):** Creates a new buffered character-output stream that uses an output buffer of the given size.

Methods:

- **write() : java.io.BufferedWriter.write(int arg)** writes a single character that is specified by an integer argument.
  Syntax :

public void write(int arg)

Parameters :
arg : integer that specifies the character to write
Return :
Doesn't return any value.

- **write() : java.io.BufferedWriter.write(String arg, int offset, int length)** writes String in the file according to its arguments as mentioned in the Java Code.
  Syntax :

public void write(String arg, int offset, int length)

Parameters :
arg : String to be written
offset : From where to start reading the string
length : No. of characters of the string to write
Return :
Doesn't return any value.

- **newLine() : java.io.BufferedWriter.newLine()** breaks/separates line.
  Syntax :

public void newLine()

Return :
Doesn't return any value.

- **flush() : java.io.BufferedWriter.flush()** flushes character from write buffer.
  Syntax :

public void flush()

Return :
Doesn't return any value.

- **close() : java.io.BufferedWriter.close()** flushes character from write buffer and then close it.
  Syntax :

public void close()

Return :
Doesn't return any value.

InputStreamReader class in Java
An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
Declaration :
public class InputStreamReader

extends Reader

Constructors :

- **InputStreamReader(InputStream in_strm) :** Creates an InputStreamReader that uses the default charset.
- **InputStreamReader(InputStream in_strm, Charset cs) :** creates an InputStreamReader that uses the given charset.
- **InputStreamReader(InputStream in_strm, CharsetDecoder dec) :** Creates an InputStreamReader that uses the given charset decoder.
- **InputStreamReader(InputStream in_strm, String charsetName) :** Creates an InputStreamReader that uses the named charset
- **ready() : java.io.InputStreamReader.ready()** tells whether the Character stream is ready to be read or not. An InputStreamReader is ready if its input buffer is not empty, or if bytes are available to be read from the underlying byte stream.
  Syntax :

public boolean ready()

Returns :
True : if the Character stream is ready to be read
False : if the Character stream is not ready to be read

- **close() : java.io.InputStreamReader.close()** closes InputStreamReader and releases all the Streams associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException.
  Syntax :

public void close()

Returns :
No value is returned

- **getEncoding() : java.io.InputStreamReader.getEncoding()** returns the name of the character encoding being used by this stream.
  Syntax :

public String getEncoding()

Parameters :
Returns :
No value is returned

- **read() : java.io.InputStreamReader.read()** Returns single character after reading.
  Syntax :

public int read()

Returns :
Returns single character after reading or -1 if the end of the stream has been reached

Java.io.OutputStreamWriter Class methods

OutputStreamWriter class connects character streams to byte streams. It encodes Characters into bytes using a specified charset.
Declaration :

public class OutputStreamWriter
extends Writer

Constructors :

- **OutputStreamWriter(OutputStream geek_out) :** Creates a "geek_out" OutputStreamWriter that uses a default charset for encoding.
- **OutputStreamWriter(OutputStream geek_out, Charset geek_set) :** Creates a "geek_out" OutputStreamWriterthat uses a "geek_ set" charset for encoding.
- **OutputStreamWriter(OutputStream geek_out, CharsetEncoder encode) :** Creates a "geek_out" OutputStreamWriter that uses a given encoder.
- **OutputStreamWriter(OutputStream geek_out, String setName) :** Creates a "geek_out" OutputStreamWriter that uses a named character set.

Methods:

- **flush() : java.io.OutputStreamWriter.flush()** flushes the stream.
  Syntax :

public void flush()

Parameters :

Return :
void
Exception :
-> **IOException :** if in case anu I/O error occurs.

- **close() : java.io.OutputStreamWriter.close()** closes the flushed stream.
  Syntax :

public void close()

Parameters :

Return :
void
Exception :
-> **IOException :** if in case anu I/O error occurs, e.g writing after closing the stream

- **write(int char) : java.io.OutputStreamWriter.write(int char)** writes a single character.
  Syntax :

public void write(int char)

Parameters :
char : character to be written
Return :
void

- **write(String geek, int offset, int strlen) : java.io.OutputStreamWriter.write(String geek, int offset, int strlen)** writes a portion of "geek" string starting from "offset position" upto "strlen" length.
  Syntax :

public void write(String geek, int offset, int strlen)

Parameters :
geek : string whose portion is to be written
offset : starting position from where to write
strlen : length upto which we need to write
Return :
void

Exception :
-> **IOException :** if in case any I/O error occurs.

- **write(char[] geek, int offset, int strlen) : java.io.OutputStreamWriter.write(char[] geek, int offset, int strlen)** writes a portion of "geek" character array starting from "offset position" upto "strlen" length.

Syntax :
public void write(char[] geek, int offset, int strlen)

Parameters :
geek : character array whose portion is to be written
offset : starting position from where to write
strlen : length upto which we need to write
Return :
void
Exception :
-> **IOException :** if in case anu I/O error occurs.

- **getEncoding() : java.io.OutputStreamWriter.getEncoding()** tells the name of the character encoding being used in the mentioned Stream.
  If there is predefined name exists, then it is returned otherwise canonical name of the encoding is returned.
  Returns Null, if the stream has been already closed.

Syntax :
public String getEncoding()

Parameters :

Return :
Name of the charset encoding used
Exception :
-> **IOException :** if in case anu I/O error occurs.

Serialization and Deserialization :-

Serialization is a mechanism of converting the state of an object into a byte stream. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

Advantages of Serialization
1. To save/persist state of an object.
2. To travel an object across a network. Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface. Serializable is a **marker interface** (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember 1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true. 2. Only non-static data members are saved via Serialization process. 3. Static data members and transient data members are not saved via Serialization process.So, if you don't want to save value of a non-static data member then make it transient. 4. Constructor of object is never called when an object is deserialized. 5. Associated objects must be implementing Serializable interface. Example :

```
class A implements Serializable{


// B also implements Serializable

// interface.

B ob=new B();

}
```

SerialVersionUID The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name. It must be static, final and of type long. i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L; If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable
classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data. It is also recommended to use private modifier for UID since it is not useful as inherited member. **serialver** The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes. You can run the following command to get serialVersionUID serialver [-classpath classpath] [-show] [classname...]

transient variables:- A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0). In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization.

Transient Vs Final:
final variables will be participated into serialization directly by their values.
Hence declaring a final variable as transient there is no use.
//the compiler assign the value to final variable
example:

```
final int x= 10;
int y = 20;
System.out.println(x);// compiler will replace this as System.out.println(10)->10
because x is final.
System.out.println(y);//20
```

```
import java.io.*;

class Emp implements Serializable {
private static final long serialversionUID =
                        129348938L;
   transient int a;
   static int b;
   String name;
   int age;

   // Default constructor
public Emp(String name, int age, int a, int b)
   {
      this.name = name;
      this.age = age;
      this.a = a;
      this.b = b;
   }

}

public class SerialExample {
public static void printdata(Emp object1)
   {

      System.out.println("name = " + object1.name);
      System.out.println("age = " + object1.age);
      System.out.println("a = " + object1.a);
      System.out.println("b = " + object1.b);
   }

public static void main(String[] args)
   {
      Emp object = new Emp("ab", 20, 2, 1000);
      String filename = "shubham.txt";

      // Serialization
      try {

         // Saving of object in a file
         FileOutputStream file = new FileOutputStream
                              (filename);
         ObjectOutputStream out = new ObjectOutputStream
                              (file);

         // Method for serialization of object
```



```
         out.writeObject(object);

         out.close();
         file.close();

         System.out.println("Object has been serialized\n"
                  + "Data before Deserialization.");
         printdata(object);

         // value of static variable changed
         object.b = 2000;
      }

      catch (IOException ex) {
         System.out.println("IOException is caught");
      }

      object = null;

      // Deserialization
      try {

         // Reading the object from a file
         FileInputStream file = new FileInputStream
                              (filename);
         ObjectInputStream in = new ObjectInputStream
                              (file);

         // Method for deserialization of object
         object = (Emp)in.readObject();

         in.close();
         file.close();
         System.out.println("Object has been deserialized\n"
                  + "Data after Deserialization.");
         printdata(object);

         // System.out.println("z = " + object1.z);
      }

      catch (IOException ex) {
         System.out.println("IOException is caught");
      }

      catch (ClassNotFoundException ex) {
         System.out.println("ClassNotFoundException" +
                  " is caught");
      }
   }
}
```

Introduction to Java NIO

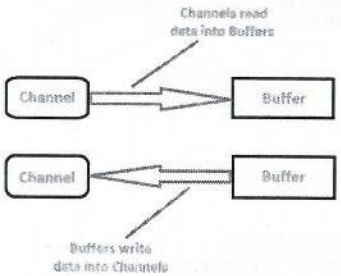
Java IO(Input/Output) is used to perform read and write operations. The java.io package contains all the classes required for input and output operation. Whereas, Java NIO (New IO) was introduced from JDK 4 to implement high-speed IO operations. It is an alternative to the standard IO API's.
In this article, we will understand more about Java NIO.

Java NIO(New Input/Output) is high-performance networking and file handling API and structure which works as an alternative IO API for Java. It is introduced from JDK 4. Java NIO works as the second I/O system after standard Java IO with some added advanced features. It provides a different way of working with I/O than the standard IO. Like **Java.io** package which contains all the classes required for Java input and output operations, the **java.nio package** defines the buffer classes which are used throughout NIO APIs. We use **Java NIO** for the following two main reasons:

1. **Non-blocking IO operation:** Java NIO performs non-blocking IO operations. This means that it reads the data whichever is ready. For instance, a thread can ask a channel to read the data from a buffer and the thread can go for other work during that period and continue again from the previous point where it has left. In the meantime, the reading operation is complete which increases the overall efficiency.
2. **Buffer oriented approach:** Java NIO's buffer oriented approach allows us to move forth and back in the buffer as we need. The data is read into a buffer and cached there. Whenever the data is required, it is further processed from the buffer.

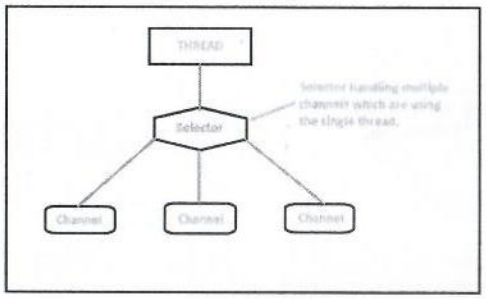
The main working of the Java NIO package is based on some core components. They are:

1. **Buffers:** Buffers are available in this package for the primitive data types. Java NIO is a buffer oriented package. It means that the data can be written/read to/from a buffer which further processed using a channel. Here, the buffers act as a container for the data as it holds the primitive data types and provides an overview of the other NIO packages. These buffers can be filled, drained, flipped, rewind, etc.
2. **Channels:** Channels are the new primitive I/O abstraction. A channel is a bit like stream used for communicating with the outside world. From the channel, we can read the data into a buffer or write from a buffer. Java NIO performs the non-blocking IO operations and the channels are available for these IO operations. The connection to different entities is represented by various channels which are capable of performing non-blocking I/O operation. The channels work as a medium or a gateway. The following image illustrates the channel and buffer interaction:



3. **Selectors:** Selectors are available for non-blocking I/O operations. A selector is an object which monitors multiple channels for the events. As Java NIO performs the non-blocking IO operations, selectors and the selection keys with selectable channels defines the multiplexed IO operations. So, in simple words, we can say that the

selectors are used to select the channels which are ready for the I/O operation. The following image illustrates the selector handling the channels:



Java NIO provides a new I/O model based on channels, buffers and selectors. So, these modules are considered as the core of the API. The following table illustrates the list of Java.nio packages for an NIO system and why they are used:

Package	Purpose
java.nio package	It provides the overview of the other NIO packages.Different types of buffers are encapsulated by this NIO system, which are used throughout the NIO API's.
java.nio.channels package	It supports channels and selectors, which represent connections to entities which are essentially open the I/O connections and selects the channel ready for I/O.
java.nio.channels.spi package	It supports the service provider classes for java.io.channel package.
java.nio.file package	It provides the support for files.
java.nio.file.spi package	It supports the service provider classes for java.io.file package.
java.nio.file.attribute package	It provides the support for file attributes.
java.nio.charset package	It defines character sets and providing encoding and decoding operations for new algorithms.
java.nio.charset.spi	It supports the service provider classes for java.nio.charset

Package	Purpose
package	package.

Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

```
public final void writeObject(Object obj)
                throws IOException
```

The ObjectInputStream class contains **readObject()** method for deserializing an object.

```
public final Object readObject()
            throws IOException,
          ClassNotFoundException
```

Advantages of Serialization 1. To save/persist state of an object. 2. To travel an object across a network only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface. Serializable is a **marker interface** (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote. **Points to remember** 1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true. 2. Only non-static data members are saved via Serialization process. 3. Static data members and transient data members are not saved via Serialization process.So, if you don't want to save value of a non-static data member then make it transient. 4. Constructor of object is never called when an object is deserialized. 5. Associated objects must be implementing Serializable interface. Example :

```
class A implements Serializable{


// B also implements Serializable

// interface.

B ob=new B();

}
```


SerialVersionUID The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name. It must be static, final and of type long. i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L; If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data. It is also recommended to use private modifier for UID since it is not useful as inherited member. **serialver** The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes. You can run the following command to get serialVersionUID serialver [-classpath classpath] [-show] [classname...]

```
// Java code for serialization and deserialization
// of a Java object
import java.io.*;

class Demo implements java.io.Serializable
{
   public int a;
   public String b;

   // Default constructor
   public Demo(int a, String b)
   {
      this.a = a;
      this.b = b;
   }

}

class Test
{
   public static void main(String[] args)
   {
      Demo object = new Demo(1, " IACSD");
      String filename = "file.ser";

      // Serialization
      try
      {
         //Saving of object in a file
         FileOutputStream file = new FileOutputStream(filename);
         ObjectOutputStream out = new ObjectOutputStream(file);

         // Method for serialization of object
```

```
            out.writeObject(object);

            out.close();
            file.close();

            System.out.println("Object has been serialized");

        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }


        Demo object1 = null;

        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object1 = (Demo)in.readObject();

            in.close();
            file.close();

            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }

    }
}
```

----------Example-2----------

```
import java.io.*;

class Emp implements Serializable {
```



```
private static final long serialversionUID =
                        129348938L;
    transient int a;
    static int b;
    String name;
    int age;

    // Default constructor
public Emp(String name, int age, int a, int b)
    {
        this.name = name;
        this.age = age;
        this.a = a;
        this.b = b;
    }

}

public class SerialExample {
public static void printdata(Emp object1)
    {

        System.out.println("name = " + object1.name);
        System.out.println("age = " + object1.age);
        System.out.println("a = " + object1.a);
        System.out.println("b = " + object1.b);
    }

public static void main(String[] args)
    {
        Emp object = new Emp("ab", 20, 2, 1000);
        String filename = "Manjiri.txt";

        // Serialization
        try {

            // Saving of object in a file
            FileOutputStream file = new FileOutputStream
                                (filename);
            ObjectOutputStream out = new ObjectOutputStream
                                (file);

            // Method for serialization of object
            out.writeObject(object);

            out.close();
            file.close();

            System.out.println("Object has been serialized\n"
                    + "Data before Deserialization.");
            printdata(object);
```

```
            // value of static variable changed
            object.b = 2000;
        }

        catch (IOException ex) {
            System.out.println("IOException is caught");
        }

        object = null;

        // Deserialization
        try {

            // Reading the object from a file
            FileInputStream file = new FileInputStream
                                (filename);
            ObjectInputStream in = new ObjectInputStream
                                (file);

            // Method for deserialization of object
            object = (Emp)in.readObject();

            in.close();
            file.close();
            System.out.println("Object has been deserialized\n"
                            + "Data after Deserialization.");
            printdata(object);

            // System.out.println("z = " + object1.z);
        }

        catch (IOException ex) {
            System.out.println("IOException is caught");
        }

        catch (ClassNotFoundException ex) {
            System.out.println("ClassNotFoundException" +
                            " is caught");
        }
    }
}
```

Description for Output: You have seen while deserializing the object the values of a and b has changed. The reason being a was marked as transient and b was static. In case of **transient variables:-** A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0). In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization.

Transient Vs Final:

final variables will be participated into serialization directly by their values.

Hence declaring a final variable as transient there is no use.
//the compiler assign the value to final variable
example:

```
final int x= 10;
int y = 20;
System.out.println(x);// compiler will replace this as System.out.println(10)->10
because x is final.
System.out.println(y);//20
```

Date class in Java

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

Constructors

- **Date()** : Creates date object representing current date and time.
- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.

Important Methods

- **boolean after(Date date) :** Tests if current date is after the given date.
- **boolean before(Date date) :** Tests if current date is before the given date.
- **int compareTo(Date date) :** Compares current date with given date. Returns 0 if the argument Date is equal to the Date; a value less than 0 if the Date is before the Date argument; and a value greater than 0 if the Date is after the Date argument.
- **long getTime()** : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- **void setTime(long time)** : Changes the current date and time to given time.

Calendar Class in Java

Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.

As it is an Abstract class, so we cannot use a constructor to create an instance. Instead, we will have to use the static method Calendar.getInstance() to instantiate and implement a sub-class.

- Calendar.getInstance(): return a Calendar instance based on the current time in the default time zone with the default locale.
- Calendar.getInstance(TimeZone zone)
- Calendar.getInstance(Locale aLocale)
- Calendar.getInstance(TimeZone zone, Locale aLocale)

METHOD	DESCRIPTION
abstract void add(int field, int amount)	It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.

METHOD	DESCRIPTION
int get(int field)	It is used to return the value of the given calendar field.
abstract int getMaximum(int field)	It is used to return the maximum value for the given calendar field of this Calendar instance.
abstract int getMinimum(int field)	It is used to return the minimum value for the given calendar field of this Calendar instance.
Date getTime()	It is used to return a Date object representing this Calendar's time value.</td

demonstrate get() method.

```
import java.util.*;
public class Calendar2 {
   public static void main(String[] args)
   {
      // creating Calendar object
      Calendar calendar = Calendar.getInstance();

      // Demonstrate Calendar's get()method
      System.out.println("Current Calendar's Year: " + calendar.get(Calendar.YEAR));
      System.out.println("Current Calendar's Day: " + calendar.get(Calendar.DATE));
      System.out.println("Current MINUTE: " + calendar.get(Calendar.MINUTE));
      System.out.println("Current SECOND: " + calendar.get(Calendar.SECOND));
   }
}
```

demonstrate getMaximum() method.

```
public class Calendar3 {
   public static void main(String[] args)
   {
      // creating calendar object
      Calendar calendar = Calendar.getInstance();

      int max = calendar.getMaximum(Calendar.DAY_OF_WEEK);
      System.out.println("Maximum number of days in a week: " + max);

      max = calendar.getMaximum(Calendar.WEEK_OF_YEAR);
      System.out.println("Maximum number of weeks in a year: " + max);
   }
}
```

demonstrate the getMinimum() method.

```
public class Calendar4 {
   public static void main(String[] args)
```



```
   {
      // creating calendar object
      Calendar calendar = Calendar.getInstance();

      int min = calendar.getMinimum(Calendar.DAY_OF_WEEK);
      System.out.println("Minimum number of days in week: " + min);

      min = calendar.getMinimum(Calendar.WEEK_OF_YEAR);
      System.out.println("Minimum number of weeks in year: " + min);
   }
}
```

demonstrate add() method.

```
public class Calendar5 {
   public static void main(String[] args)
   {
      // creating calendar object
      Calendar calendar = Calendar.getInstance();
      calendar.add(Calendar.DATE, -15);
      System.out.println("15 days ago: " + calendar.getTime());
      calendar.add(Calendar.MONTH, 4);
      System.out.println("4 months later: " + calendar.getTime());
      calendar.add(Calendar.YEAR, 2);
      System.out.println("2 years later: " + calendar.getTime());
   }
}
```

Java SimpleDateFormat

The java.text.SimpleDateFormat class provides methods to format and parse date and time in java. The SimpleDateFormat is a concrete class for formatting and parsing date which inherits java.text.DateFormat class.

Notice that *formatting means converting date to string* and *parsing means converting string to date*.

Constructors of the Class SimpleDateFormat

SimpleDateFormat(String pattern_args): Instantiates the SimpleDateFormat class using the provided pattern - pattern_args, default date format symbols for the default FORMAT locale.

SimpleDateFormat(String pattern_args, Locale locale_args): Instantiates the SimpleDateFormat class using the provided pattern - pattern_args. For the provided FORMAT Locale, the default date format symbols are - locale_args.

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class SimpleDateFormatExample {
public static void main(String[] args) {
   Date date = new Date();
   SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
   String strDate= formatter.format(date);
```

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

Java Convert Date to String

We can convert **Date to String in java** using *format()* method of java.text.DateFormat class.

format() method of DateFormat

The format() method of DateFormat class is used to convert Date into String. DateFormat is an abstract class. The child class of DateFormat is SimpleDateFormat. It is the implementation of DateFormat class. The **signature** of format() method is given below:

```
String format(Date d)
```

Java Date to String Example

Let's see the simple code to convert Date to String in java.

```
Date date = Calendar.getInstance().getTime();
DateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
String strDate = dateFormat.format(date);
```

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Calendar;
public class DateToStringExample1 {
    public static void main(String args[]){
        Date date = Calendar.getInstance().getTime();
        DateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
        String strDate = dateFormat.format(date);
        System.out.println("Converted String: " + strDate);

    }
}
```

Java String to Date Example

Let's see the simple code to convert String to Date in java.

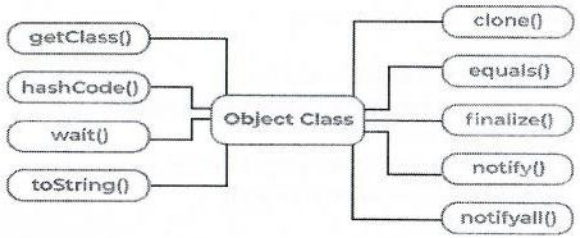

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class StringToDateExample1 {
public static void main(String[] args)throws Exception {
  String sDate1="31/12/1998";
  Date date1=new SimpleDateFormat("dd/MM/yyyy").parse(sDate1);
    System.out.println(sDate1+"\t"+date1);
 }
 }
```

Different Format

```
String sDate1="31/12/1998";
String sDate2 = "31-Dec-1998";
String sDate3 = "12 31, 1998";
String sDate4 = "Thu, Dec 31 1998";
String sDate5 = "Thu, Dec 31 1998 23:37:50";
String sDate6 = "31-Dec-1998 23:37:50";
SimpleDateFormat formatter1=new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat formatter2=new SimpleDateFormat("dd-MMM-yyyy");
SimpleDateFormat formatter3=new SimpleDateFormat("MM dd, yyyy");
SimpleDateFormat formatter4=new SimpleDateFormat("E, MMM dd yyyy");
SimpleDateFormat formatter5=new SimpleDateFormat("E, MMM dd yyyy HH:mm:ss");
SimpleDateFormat formatter6=new SimpleDateFormat("dd-MMM-yyyy HH:mm:ss");
Date date1=formatter1.parse(sDate1);
Date date2=formatter2.parse(sDate2);
Date date3=formatter3.parse(sDate3);
Date date4=formatter4.parse(sDate4);
Date date5=formatter5.parse(sDate5);
Date date6=formatter6.parse(sDate6);
```

Object Class in Java

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a class does not extend any other class then it is a direct child class of **Object** and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.



Using Object Class Methods

The Object class provides multiple methods which are as follows:

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method
- clone() method
- wait(), notify() notifyAll() methods

1. toString() method

The toString() provides a String representation of an object and is used to convert an object to a String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString()
{
   return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

It is always recommended to override the **toString()** method to get our own String representation of Object.

Note: *Whenever we try to print any Object reference, then internally toString() method is called.*

2. hashCode() method

For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm. The hashCode()

method is **native** because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.

Use of hashCode() method

It returns a hash value that is used to search objects in a collection. JVM(Java Virtual Machine) uses the hashcode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc. The main advantage of saving objects based on hash code is that searching becomes easy.

Note: *Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example, for a Student class, we can return the roll no. of a student from the hashCode() method as it is unique.*

```
public class Student {
   static int last_roll = 100;
   int roll_no;

   // Constructor
   Student()
   {
      roll_no = last_roll;
      last_roll++;
   }

   // Overriding hashCode()
   @Override public int hashCode() { return roll_no; }

   // Driver code
   public static void main(String args[])
   {
      Student s = new Student();

      // Below two statements are equivalent
      System.out.println(s);
      System.out.println(s.toString());
   }
}
```

3. equals(Object obj) method

It compares the given object to "this" object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override the **equals(Object obj)** method to get our own equality condition on Objects.

Note: *It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.*

4. getClass() method

It returns the class object of "this" object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don't override it.

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

```
        Object obj = new String("IACSD");
        Class c = obj.getClass();
        System.out.println("Class of Object obj is : "
                           + c.getName());
    }
}
```

Note: *After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in Reflection*

5. finalize() method

This method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object. We should override finalize() method to dispose of system resources, perform clean-up activities and minimize memory leaks. For example, before destroying the Servlet objects web container, always called finalize method to perform clean-up activities of the session.

Note: *The finalize method is called just* ***once*** *on an object even though that object is eligible for garbage collection multiple times.*

```
// Java program to demonstrate working of finalize()

public class Test {
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();

        System.out.println("end");
    }

    @Override protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Clone() method in Java

bject cloning refers to the creation of an exact copy of an object. It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

Using Assignment Operator to create a copy of the reference variable

In Java, there is no operator to create a copy of an object. Unlike C++, in Java, if we use the assignment operator then it will create a copy of the reference variable and not

the object. This can be explained by taking an example. The following program demonstrates the same.

```
// Java program to demonstrate that assignment operator
// only creates a new reference to same object
import java.io.*;

// A test class whose objects are cloned
class Test {
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}

// Driver Class
class Main {
    public static void main(String[] args)
    {
        Test ob1 = new Test();

        System.out.println(ob1.x + " " + ob1.y);

        // Creating a new reference variable ob2
        // pointing to same address as ob1
        Test ob2 = ob1;

        // Any change made in ob2 will
        // be reflected in ob1
        ob2.x = 100;

        System.out.println(ob1.x + " " + ob1.y);
        System.out.println(ob2.x + " " + ob2.y);
    }
}
```

Creating a copy using the clone() method

The class whose object's copy is to be made must have a public clone method in it or in one of its parent class.

- Every class that implements clone() should call super.clone() to obtain the cloned object reference.
- The class must also implement java.lang.Cloneable interface whose object
- clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class's object.
- Syntax:

protected Object clone() throws CloneNotSupportedException

Usage of clone() method -Shallow Copy

Please Note – In the below code example the clone() method does create a completely new object with a different hashCode value, which means its in a separate

memory location. But due to the Test object c being inside Test2, the primitive types have achieved deep copy but this Test object c is still shared between t1 and t2. To overcome that we explicitly do a deep copy for object variable c, which is discussed later.

```
// A Java program to demonstrate
// shallow copy using clone()
import java.util.ArrayList;

// An object reference of this class is
// contained by Test2
class Test {
    int x, y;
}

// Contains a reference of Test and
// implements clone with shallow copy.
class Test2 implements Cloneable {
    int a;
    int b;
    Test c = new Test();
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

// Driver class
public class Main {
    public static void main(String args[])
        throws CloneNotSupportedException
    {
        Test2 t1 = new Test2();
        t1.a = 10;
        t1.b = 20;
        t1.c.x = 30;
        t1.c.y = 40;

        Test2 t2 = (Test2)t1.clone();

        // Creating a copy of object t1
        // and passing it to t2
        t2.a = 100;

        // Change in primitive type of t2 will
        // not be reflected in t1 field
        t2.c.x = 300;

        // Change in object type field will be
        // reflected in both t2 and t1(shallow copy)
        System.out.println(t1.a + " " + t1.b + " " + t1.c.x
                           + " " + t1.c.y);
```



```
        System.out.println(t2.a + " " + t2.b + " " + t2.c.x
                           + " " + t2.c.y);
    }
}
```

In the above example, t1.clone returns the shallow copy of the object t1. To obtain a deep copy of the object certain modifications have to be made in the clone method after obtaining the copy.

Deep Copy vs Shallow Copy

- **Shallow copy** is the method of copying an object and is followed by default in cloning. In this method, the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to the same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
- Therefore, any changes made in referenced objects in object X or Y will be reflected in other objects.

Shallow copies are cheap and simple to make. In the above example, we created a shallow copy of the *object.*

Usage of clone() method – Deep Copy

- If we want to create a deep copy of object X and place it in a new object Y then a new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In the below example, we create a deep copy of the object.
- 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.

```
// A Java program to demonstrate
// deep copy using clone()

// An object reference of this
// class is contained by Test2
class Test {
    int x, y;
}

// Contains a reference of Test and
// implements clone with deep copy.
class Test2 implements Cloneable {
    int a, b;

    Test c = new Test();

    public Object clone() throws CloneNotSupportedException
    {
        // Assign the shallow copy to
        // new reference variable t
        Test2 t = (Test2)super.clone();
```

```
        // Creating a deep copy for c
        t.c = new Test();
        t.c.x = c.x;
        t.c.y = c.y;

        // Create a new object for the field c
        // and assign it to shallow copy obtained,
        // to make it a deep copy
        return t;
    }
}

public class Main {
    public static void main(String args[])
        throws CloneNotSupportedException
    {
        Test2 t1 = new Test2();
        t1.a = 10;
        t1.b = 20;
        t1.c.x = 30;
        t1.c.y = 40;

        Test2 t3 = (Test2)t1.clone();
        t3.a = 100;

        // Change in primitive type of t2 will
        // not be reflected in t1 field
        t3.c.x = 300;

        // Change in object type field of t2 will
        // not be reflected in t1(deep copy)
        System.out.println(t1.a + " " + t1.b + " " + t1.c.x
                           + " " + t1.c.y);
        System.out.println(t3.a + " " + t3.b + " " + t3.c.x
                           + " " + t3.c.y);
    }
}
```

In the above example, we can see that a new object for the Test class has been assigned to copy an object that will be returned to the clone method. Due to this t3 will obtain a deep copy of the object t1. So any changes made in 'c' object fields by t3, will not be reflected in t1.

Advantages of clone method:

- If we use the assignment operator to assign an object reference to another reference variable then it will point to the same address location of the old object and no new copy of the object will be created. Due to this any changes in the reference variable will be reflected in the original object.
- If we use a copy constructor, then we have to copy all the data over explicitly i.e. we have to reassign all the fields of the class in the constructor explicitly.

But in the clone method, this work of creating a new copy is done by the method itself. So to avoid extra processing we use object cloning.

Collections in Java

Any group of individual objects which are represented as a single unit is known as the collection of the objects. In Java, a separate framework named the *"Collection Framework"* has been defined in JDK 1.2 which holds all the collection classes and interface in it.
The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

What is a Framework?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

Need for a Separate Collection Framework

Before the Collection Framework(or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were **Arrays** or **Vectors**, or **Hashtables**. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different **methods**, syntax, and **constructors** present in every collection class.
Let's understand this with an example of adding an element in a hashtable and a vector.

```
// Java program to demonstrate
// why collection framework was needed
import java.io.*;
import java.util.*;

class CollectionDemo {

    public static void main(String[] args)
    {
        // Creating instances of the array,
        // vector and hashtable
        int arr[] = new int[] { 1, 2, 3, 4 };
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();

        // Adding the elements into the
        // vector
        v.addElement(1);
        v.addElement(2);

        // Adding the element into the
        // hashtable
        h.put(1, "IACSD");
        h.put(2, "4IACSD");

        // Array instance creation requires [],
```

```
        // while Vector and hastable require ()
        // Vector element insertion requires addElement(),
        // but hashtable element insertion requires put()

        // Accessing the first element of the
        // array, vector and hashtable
        System.out.println(arr[0]);
        System.out.println(v.elementAt(0));
        System.out.println(h.get(1));

        // Array elements are accessed using [],
        // vector elements using elementAt()
        // and hashtable elements using get()
    }
}
```

As we can observe, none of these collections(Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback is that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection. Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

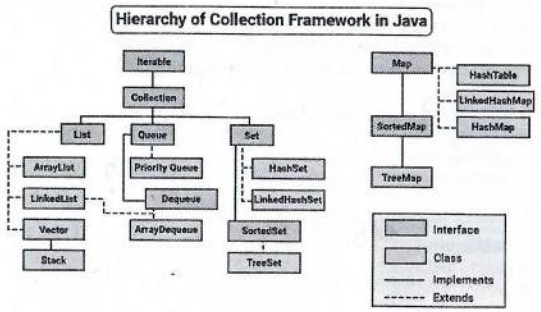
Hierarchy of the Collection Framework

The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework. The collection framework contains an interface named an

iterable interface which provides the iterator to iterate through all the collections. This interface is extended by the main collection interface which acts as a root for the

collection framework. All the collections extend this collection interface thereby

extending the properties of the iterator and the methods of this interface. The following figure illustrates the hierarchy of the collection framework.



Before understanding the different components in the above framework, let's first understand a class and an interface.

- **Class:** A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- **Interface:** Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body). Interfaces specify what a class must do and not how. It is the blueprint of the class.

Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

Method	Description
add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.
equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.

Method	Description
max()	This method is used to return the maximum value present in the collection.
parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to remove all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retain only the elements in this collection that are contained in the specified collection.
size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

Interfaces that extend the Collections Interface

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

1. Iterable Interface: This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the

Iterator iterator();

2. Collection Interface: This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.

3. List Interface: This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

```
List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is the type of the object
```

The classes which implement the List interface are as follows:

A. ArrayList: ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list. ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases. Let's understand the ArrayList with the following example

```
// Java program to demonstrate the
// working of ArrayList
import java.io.*;
import java.util.*;

class Demo {

    // Main Method
    public static void main(String[] args)
    {

        // Declaring the ArrayList with
        // initial size n
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Appending new elements at
        // the end of the list
        for (int i = 1; i <= 5; i++)
            al.add(i);

        // Printing elements
        System.out.println(al);

        // Remove element at index 3
        al.remove(3);
```

```
        // Displaying the ArrayList
        // after deletion
        System.out.println(al);

        // Printing elements one by one
        for (int i = 0; i < al.size(); i++)
            System.out.print(al.get(i) + " ");
    }
}
```

B. LinkedList: LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

Let's understand the LinkedList with the following example:

```
// Java program to demonstrate the
// working of LinkedList
import java.io.*;
import java.util.*;

class Demo {

    // Main Method
    public static void main(String[] args)
    {

        // Declaring the LinkedList
        LinkedList<Integer> ll = new LinkedList<Integer>();

        // Appending new elements at
        // the end of the list
        for (int i = 1; i <= 5; i++)
            ll.add(i);

        // Printing elements
        System.out.println(ll);

        // Remove element at index 3
        ll.remove(3);

        // Displaying the List
        // after deletion
        System.out.println(ll);

        // Printing elements one by one
        for (int i = 0; i < ll.size(); i++)
            System.out.print(ll.get(i) + " ");
    }
}
```

C. Vector: A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

Let's understand the Vector with an example:

```
// Java program to demonstrate the
// working of Vector
import java.io.*;
import java.util.*;

class vectorDemo {

    // Main Method
    public static void main(String[] args)
    {

        // Declaring the Vector
        Vector<Integer> v = new Vector<Integer>();

        // Appending new elements at
        // the end of the list
        for (int i = 1; i <= 5; i++)
            v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the Vector
        // after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)
            System.out.print(v.get(i) + " ");
    }
}
```

D. Stack: Stack class models and implements the Stack data structure. The class is based on the basic principle of *last-in-first-out*. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be referred to as the subclass of Vector. Let's understand the stack with an example:

```
// Java program to demonstrate the
// working of a stack
import java.util.*;
public class Demo {

    // Main Method
```

```
    public static void main(String args[])
    {
        Stack<String> stack = new Stack<String>();
        stack.push("Welcome");
        stack.push("in");
        stack.push("IACSD");
        stack.push("Pune");

        // Iterator for the stack
        Iterator<String> itr = stack.iterator();

        // Printing the stack
        while (itr.hasNext()) {
            System.out.print(itr.next() + " ");
        }

        System.out.println();

        stack.pop();

        // Iterator for the stack
        itr = stack.iterator();

        // Printing the stack
        while (itr.hasNext()) {
            System.out.print(itr.next() + " ");
        }
    }
}
```

Note: Stack is a subclass of Vector and a legacy class. It is thread-safe which might be overhead in an environment where thread safety is not needed. An alternate to Stack is to use ArrayDequeue which is not thread-safe and has faster array implementation.

4. Queue Interface: As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold on a first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example,

Queue <T> pq = new PriorityQueue<> ();
Queue <T> ad = new ArrayDeque<> ();
Where T is the type of the object.

The most frequently used implementation of the queue interface is the PriorityQueue.
Priority Queue: A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used

in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. Let's understand the priority queue with an

example:

```
// Java program to demonstrate the working of
// priority queue in Java
import java.util.*;

class PQueueDemo {

    // Main Method
    public static void main(String args[])
    {
        // Creating empty priority queue
        PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();

        // Adding items to the pQueue using add()
        pQueue.add(10);
        pQueue.add(20);
        pQueue.add(15);

        // Printing the top element of PriorityQueue
        System.out.println(pQueue.peek());

        // Printing the top element and removing it
        // from the PriorityQueue container
        System.out.println(pQueue.poll());

        // Printing the top element again
        System.out.println(pQueue.peek());
    }
}
```

5. **Deque Interface:** This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class. For example,

   The class which implements the deque interface is ArrayDeque.
   ArrayDeque: ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage. Let's understand ArrayDeque with an example:

```
// Java program to demonstrate the
// ArrayDeque class in Java
```

```
import java.util.*;
public class ArrayDequeDemo {
   public static void main(String[] args)
   {
      // Initializing an deque
      ArrayDeque<Integer> de_que = new ArrayDeque<Integer>(10);

      // add() method to insert
      de_que.add(10);
      de_que.add(20);
      de_que.add(30);
      de_que.add(40);
      de_que.add(50);

      System.out.println(de_que);

      // clear() method
      de_que.clear();

      // addFirst() method to insert the
      // elements at the head
      de_que.addFirst(564);
      de_que.addFirst(291);

      // addLast() method to insert the
      // elements at the tail
      de_que.addLast(24);
      de_que.addLast(14);

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

6. Set Interface: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes. For example,

Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
Where T is the type of the object.

The following are the classes that implement the Set interface:

A. HashSet: The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements. Let's understand HashSet with an example:

```
// Java program to demonstrate the
// working of a HashSet
import java.util.*;
```



```
public class HashSetDemo {

    // Main Method
   public static void main(String args[])
   {
      // Creating HashSet and
      // adding elements
      HashSet<String> hs = new HashSet<String>();

      hs.add("Welcome");
      hs.add("In ");
      hs.add("IACSD");
      hs.add("Pune");
      hs.add("CDAC");

      // Traversing elements
      Iterator<String> itr = hs.iterator();
      while (itr.hasNext()) {
         System.out.println(itr.next());
      }
   }
}
```

B. LinkedHashSet: A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements. Let's understand the LinkedHashSet with an example:

```
// Java program to demonstrate the
// working of a LinkedHashSet
import java.util.*;

public class LinkedHashSetDemo {

    // Main Method
   public static void main(String args[])
   {
      // Creating LinkedHashSet and
      // adding elements
      LinkedHashSet<String> lhs = new LinkedHashSet<String>();

      lhs.add("Welcome");
      lhs.add("In");
      lhs.add("IACSD");
      lhs.add("CDAC");
      lhs.add("Pune");

      // Traversing elements
      Iterator<String> itr = lhs.iterator();
      while (itr.hasNext()) {
```

```
            System.out.println(itr.next());
        }
    }
}
```

7. Sorted Set Interface: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class. For example,

SortedSet<T> ts = new TreeSet<> ();
Where T is the type of the object.
The class which implements the sorted set interface is TreeSet.

TreeSet: The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. Let's understand TreeSet with an example:

```
// Java program to demonstrate the
// working of a TreeSet
import java.util.*;

public class TreeSetDemo {

    // Main Method
    public static void main(String args[])
    {
        // Creating TreeSet and
        // adding elements
        TreeSet<String> ts = new TreeSet<String>();

        ts.add("Welcome");
        ts.add("In");
        ts.add("IACSD");
        ts.add("CDAC");
        ts.add("Pune");

        // Traversing elements
        Iterator<String> itr = ts.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

8. **Map Interface:** A map is a data structure that supports the key-value pair for mapping the data. This interface doesn't support duplicate keys because the same key cannot

have multiple mappings, however it allows duplicate values in different keys. A map is useful if there is data and we wish to perform operations on the basis of the key. This map interface
is implemented by various classes like HashMap, TreeMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. For example

Map<T> hm = new HashMap<> ();
Map<T> tm = new TreeMap<> ();

Where T is the type of the object. ,

The frequently used implementation of a Map interface is a HashMap.

HashMap: HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also
uses HashMap internally. Let's understand the HashMap with an example:

```
// Java program to demonstrate the
// working of a HashMap
import java.util.*;

public class HashMapDemo {

    // Main Method
    public static void main(String args[])
    {
        // Creating HashMap and
        // adding elements
        HashMap<Integer, String> hm = new HashMap<Integer, String>();

        hm.put(1, "Welcome");
        hm.put(2, "In");
        hm.put(3, "IACSD");

        // Finding the value for a key
        System.out.println("Value for 1 is " + hm.get(1));

        // Traversing through the HashMap
        for (Map.Entry<Integer, String> e : hm.entrySet())
            System.out.println(e.getKey() + " " + e.getValue());
    }
}
```

Set in Java:-
The set interface is present in java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements. There are two interfaces that extend the set implementation namely SortedSet and NavigableSet.

the navigable set extends the sorted set interface. Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.
Declaration: The Set interface is declared as:
public interface Set extends Collection

Creating Set Objects
Since Set is an interface, objects cannot be created of the typeset. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. This type-safe set can be defined as:
// Obj is the type of the object to be stored in Set

```
Set<Obj> set = new HashSet<Obj> ();
```

Let us discuss methods present in the Set interface provided below in a tabular format below as follows:

Method	Description
add(element)	This method is used to add a specific element to the set. The function adds the element only if the specified element is not already present in the set else the function returns False if the element is already present in the Set.
addAll(collection)	This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.
clear()	This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.
contains(element)	This method is used to check whether a specific element is present in the Set or not.
containsAll(collection)	This method is used to check whether the set contains all the elements present in the given collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.
hashCode()	This method is used to get the hashCode value for this instance of the Set. It returns an integer value which is the hashCode value for this instance of the Set.

Method	Description
isEmpty()	This method is used to check whether the set is empty or not.
iterator()	This method is used to return the iterator of the set. The elements from the set are returned in a random order.
remove(element)	This method is used to remove the given element from the set. This method returns True if the specified element is present in the Set otherwise it returns False.
removeAll(collection)	This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set changed as a result of the call.
retainAll(collection)	This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.
size()	This method is used to get the size of the set. This returns an integer value which signifies the number of elements.
toArray()	This method is used to form an array of the same elements as that of the Set.

```
// Java program Illustrating Set Interface

// Importing utility classes
import java.util.*;

// Main class
public class Demo {

    // Main driver method
    public static void main(String[] args)
    {
        // Demonstrating Set using HashSet
        // Declaring object of type String
        Set<String> hash_Set = new HashSet<String>();

        // Adding elements to the Set
        // using add() method
        hash_Set.add("Welcome");
        hash_Set.add("In");
        hash_Set.add("IACSD");
```

```
        hash_Set.add("Pune");
        hash_Set.add("Set");

        // Printing elements of HashSet object
        System.out.println(hash_Set);
    }
}
```

Operations on the Set Interface

The set interface allows the users to perform the basic mathematical operation on the set.

Let's take two arrays to understand these basic operations. Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]. Then the possible operations on the sets are:

1. **Intersection:** This operation returns all the common elements from the given two sets. For the above two sets, the intersection would be:Intersection = [0, 1, 3, 4]
2. **Union:** This operation adds all the elements in one set with the other. For the above two sets, the union would be: Union = [0, 1, 2, 3, 4, 5, 7, 8, 9]
3. **Difference:** This operation removes all the values present in one set from the other set. For the above two sets, the difference would be: Difference = [2, 8, 9]

```
// Java Program Demonstrating Operations on the Set
// such as Union, Intersection and Difference operations

// Importing all utility classes
import java.util.*;

// Main class
public class SetExample {

    // Main driver method
    public static void main(String args[])
    {
        // Creating an object of Set class
        // Declaring object of Integer type
        Set<Integer> a = new HashSet<Integer>();

        // Adding all elements to List
        a.addAll(Arrays.asList(
            new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));

        // Again declaring object of Set class
        // with reference to HashSet
        Set<Integer> b = new HashSet<Integer>();

        b.addAll(Arrays.asList(
            new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));
```



```
        // To find union
        Set<Integer> union = new HashSet<Integer>(a);
        union.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(union);

        // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);

        // To find the symmetric difference
        Set<Integer> difference = new HashSet<Integer>(a);
        difference.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(difference);
    }
}
```

Performing Various Operations on SortedSet

After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. Since Set is an interface, it can be used only with a class that implements this interface. HashSet is one of the widely used classes which implements the Set interface. Now, let's see how to perform a few frequently used operations on the HashSet. We are going to perform the following operations as follows:

1. Adding elements
2. Accessing elements
3. Removing elements
4. Iterating elements
5. Iterating through Set

Now let us discuss these operations individually as follows:

1. **Adding Elements**

In order to add an element to the Set, we can use the add() method. However, the insertion order is not retained in the Set. Internally, for every element, a hash is generated and the values are stored with respect to the generated hash. the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are accepted by the Set.

```
// Creating an object of Set and
        // declaring object of type String
        Set<String> hs = new HashSet<String>();

        // Adding elements to above object
        // using add() method
        hs.add("B");
        hs.add("B");
        hs.add("C");
```

```
hs.add("A");

// Printing the elements inside the Set object
System.out.println(hs);
```

2:Accessing the Elements

After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains().

```
// Creating an object of Set and
    // declaring object of type String
    Set<String> hs = new HashSet<String>();

    // Elements are added using add() method
    // Later onwards we will show accessing the same

    // Custom input elements
    hs.add("A");
    hs.add("B");
    hs.add("C");
    hs.add("A");

    // Print the Set object elements
    System.out.println("Set is " + hs);

    // Declaring a string
    String check = "D";

    // Check if the above string exists in
    // the SortedSet or not
    // using contains() method
    System.out.println("Contains " + check + " "
                + hs.contains(check));
```

3: **Removing the Values**

The values can be removed from the Set using the remove() method.

```
// Declaring object of Set of type String
    Set<String> hs = new HashSet<String>();

    // Elements are added
    // using add() method

    // Custom input elements
    hs.add("A");
    hs.add("B");
    hs.add("C");
    hs.add("B");
    hs.add("D");
    hs.add("E");

    // Printing initial Set elements
```



```
    System.out.println("Initial HashSet " + hs);

    // Removing custom element
    // using remove() method
    hs.remove("B");

    // Printing Set elements after removing an element
    // and printing updated Set elements
    System.out.println("After removing element " + hs);
```

4: Iterating through the Set

There are various ways to iterate through the Set. The most famous one is to use the enhanced for loop.

```
// Creating object of Set and declaring String type
    Set<String> hs = new HashSet<String>();

    // Adding elements to Set
    // using add() method

    // Custom input elements
    hs.add("A");
    hs.add("B");
    hs.add("C");
    hs.add("B");
    hs.add("D");
    hs.add("E");

    // Iterating through the Set
    // via for-each loop
    for (String value : hs)

      // Printing all the values inside the object
      System.out.print(value + ", ");

    System.out.println();
```

Classes that implement the Set interface in Java Collections can be easily perceived from the image below as follows and are listed as follows:

- *HashSet*
- *EnumSet*
- *LinkedHashSet*
- *TreeSet*

Class 1: HashSet

HashSet class which is implemented in the collection framework is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements. Let's see how to create a set object using this class.

```
// Java program Demonstrating Creation of Set object
// Using the Hashset class

// Importing utility classes
import java.util.*;

// Main class
class DemoHashset {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating object of Set of type String
        Set<String> h = new HashSet<String>();

        // Adding elements into the HashSet
        // using add() method

        // Custom input elements
        h.add("India");
        h.add("Australia");
        h.add("South Africa");

        // Adding the duplicate element
        h.add("India");

        // Displaying the HashSet
        System.out.println(h);

        // Removing items from HashSet
        // using remove() method
        h.remove("Australia");
        System.out.println("Set after removing "
                           + "Australia:" + h);

        // Iterating over hash set items
        System.out.println("Iterating over set:");

        // Iterating through iterators
        Iterator<String> i = h.iterator();

        // It holds true till there is a single element
        // remaining in the object
        while (i.hasNext())

            System.out.println(i.next());
    }
}
```

Class 2: EnumSet

EnumSet class which is implemented in the collections framework is one of the

specialized implementations of the Set interface for use with the enumeration type. It is a high-performance set implementation, much faster than HashSet. All of the elements in an enum set must come from a single enumeration type that is specified when the set is created either explicitly or implicitly. Let's see how to create a set object using this class.

```
// Java program to demonstrate the
// creation of the set object
// using the EnumSet class
import java.util.*;

enum demo { CODE, LEARN, CONTRIBUTE, QUIZ, MCQ }
;

public class DemoEnum {

    public static void main(String[] args)
    {
        // Creating a set
        Set<demo> set1;

        // Adding the elements
        set1 = EnumSet.of(demo.QUIZ, demo.CONTRIBUTE,
                          demo.LEARN, demo.CODE);

        System.out.println("Set 1: " + set1);
    }
}
```

Class 3: LinkedHashSet

LinkedHashSet class which is implemented in the collections framework is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. Let's see how to create a set object using this class.

```
// Java program to demonstrate the
// creation of Set object using
// the LinkedHashset class
import java.util.*;

class DemoLinkedList {

    public static void main(String[] args)
    {
        Set<String> lh = new LinkedHashSet<String>();

        // Adding elements into the LinkedHashSet
        // using add()
        lh.add("India");
        lh.add("Australia");
```

```
        lh.add("South Africa");

        // Adding the duplicate
        // element
        lh.add("India");

        // Displaying the LinkedHashSet
        System.out.println(lh);

        // Removing items from LinkedHashSet
        // using remove()
        lh.remove("Australia");
        System.out.println("Set after removing "
                    + "Australia:" + lh);

        // Iterating over linked hash set items
        System.out.println("Iterating over set:");
        Iterator<String> i = lh.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

Class 4: TreeSet

TreeSet class which is implemented in the collections framework and implementation of the SortedSet Interface and SortedSet extends Set Interface. It behaves like a simple set with the exception that it stores elements in a sorted format. TreeSet uses a tree data structure for storage. Objects are stored in sorted, ascending order. But we can iterate in descending order using the method TreeSet.descendingIterator(). Let's see how to create a set object using this class.

```
Set<String> ts = new TreeSet<String>();

        // Adding elements into the TreeSet
        // using add()
        ts.add("India");
        ts.add("Australia");
        ts.add("South Africa");

        // Adding the duplicate
        // element
        ts.add("India");

        // Displaying the TreeSet
        System.out.println(ts);

        // Removing items from TreeSet
        // using remove()
        ts.remove("Australia");
        System.out.println("Set after removing "
                    + "Australia:" + ts);
```



```
        // Iterating over Tree set items
        System.out.println("Iterating over set:");
        Iterator<String> i = ts.iterator();

        while (i.hasNext())
            System.out.println(i.next());
```

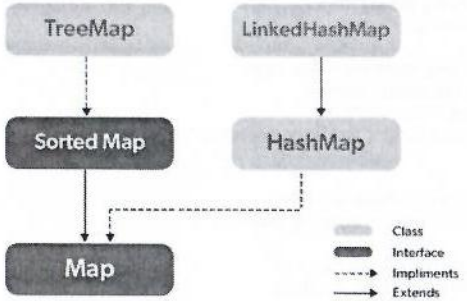
Map Interface in Java

The map interface is present in java.util package represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.

why and when to use Maps?

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some common scenarios are as follows:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).



Creating Map Objects

Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Map.
Syntax: Defining Type-safe Map

Map hm = new HashMap();

// Obj is the type of the object to be stored in Map

Characteristics of a Map Interface

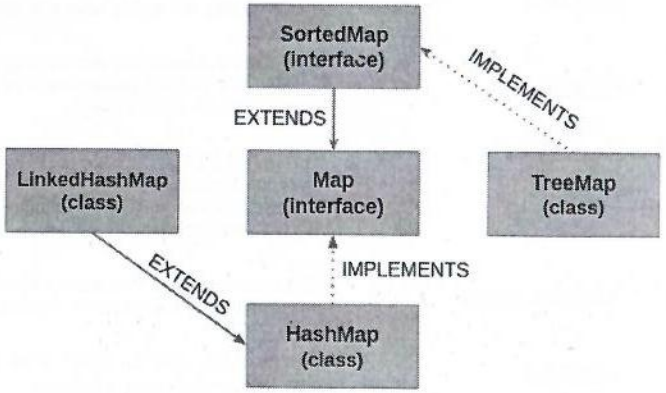
1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on the specificimplementations.For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
3. There are two interfaces for implementing Map in java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

Methods in Map Interface

Method	Action Performed
clear()	This method is used to clear and remove all of the elements or mappings from a specified Map collection.
containsKey(Object)	This method is used to check whether a particular key is being mapped into the Map or not. It takes the key element as a parameter and returns True if that element is mapped in the map.
containsValue(Object)	This method is used to check whether a particular value is being mapped by a single or more than one key in the Map. It takes the value as a parameter and returns True if that value is mapped by any of the key in the map.
entrySet()	This method is used to create a set out of the same elements contained in the map. It basically returns a set view of the map or we can create a new set and store the map elements into them.
equals(Object)	This method is used to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not.

Method	Action Performed
get(Object)	This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key.
hashCode()	This method is used to generate a hashCode for the given map containing keys and values.
isEmpty()	This method is used to check if a map is having any entry for key and value pairs. If no mapping exists, then this returns true.
keySet()	This method is used to return a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
put(Object, Object)	This method is used to associate the specified value with the specified key in this map.
putAll(Map)	This method is used to copy all of the mappings from the specified map to this map.
remove(Object)	This method is used to remove the mapping for a key from this map if it is present in the map.
size()	This method is used to return the number of key/value pairs available in the map.
values()	This method is used to create a collection out of the values of the map. It basically returns a Collection view of the values in the HashMap.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

Method	Action Performed
putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current associate value.



MAP Hierarchy in Java

Class 1: ***HashMap***

HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. This class uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String. A shorter value helps in indexing and faster searches. Let's see how to create a map object using this class.

```
// Creating an empty HashMap
    Map<String, Integer> map = new HashMap<>();

    // Inserting entries in the Map
    // using put() method
    map.put("vishal", 10);
    map.put("sachin", 30);
    map.put("vaibhav", 20);
```



```
    // Iterating over Map
    for (Map.Entry<String, Integer> e : map.entrySet())

        // Printing key-value pairs
        System.out.println(e.getKey() + " "
                + e.getValue());
```

Class 2: ***LinkedHashMap***

LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Let's see how to create a map object using this class.

```
    // Creating an empty LinkedHashMap
    Map<String, Integer> map = new LinkedHashMap<>();

    // Inserting pair entries in above Map
    // using put() method
    map.put("vishal", 10);
    map.put("sachin", 30);
    map.put("vaibhav", 20);

    // Iterating over Map
    for (Map.Entry<String, Integer> e : map.entrySet())

        // Printing key-value pairs
        System.out.println(e.getKey() + " "
                + e.getValue());
```

Class 3: ***TreeMap***

The TreeMap in Java is used to implement the Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. Let's see how to create a map object using this class.

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

    // Creating an empty TreeMap
    Map<String, Integer> map = new TreeMap<>();

    // Inserting custom elements in the Map
    // using put() method
```

```
}
```

Iterating through the Map

There are multiple ways to iterate through the Map. The most famous way is to use a for-each loop and get the keys. The value of the key is found by using the getValue() method.

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

    // Initialization of a Map
    // using Generics
    Map<Integer, String> hm1
       = new HashMap<Integer, String>();

    // Inserting the Elements
    hm1.put(new Integer(1), "Welcome");
    hm1.put(new Integer(2), "In");
    hm1.put(new Integer(3), "IACSD");

    for (Map.Entry mapElement : hm1.entrySet()) {
       int key
          = (int)mapElement.getKey();

       // Finding the value
       String value
          = (String)mapElement.getValue();

       System.out.println(key + " : "
                          + value);
    }
```

Count Occurrence of number using Hashmap

In this code we are using **putIfAbsent()** along with **Collections.frequency()** to count the exact occurence of numbers. In many program you need to count occurrence of particular number or letter. You use following approach to solve those type of problems

```
class HelloWorld {
   public static void main(String[] args) {
      int a[]={1,13,4,1,41,31,31,4,13,2};
        // put all elements in arraylist
      ArrayList<Integer> aa=new ArrayList();
      for(int i=0;i<a.length;i++){
         aa.add(a[i]);
      }
      HashMap<Integer,Integer> h=new HashMap();
        //counting occurrence of numbers
      for(int i=0;i<aa.size();i++){
         h.putIfAbsent(aa.get(i),Collections.frequency(aa,aa.get(i)));
      }
```



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

Hashtable in Java

The **Hashtable** class implements a hash table, which 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. The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface. It was part of the original Java Collections framework and was introduced in Java 1.0.

However, the Hashtable class has since been considered obsolete and its use is generally discouraged. This is because it was designed prior to the introduction of the Collections framework and does not implement the Map interface, which makes it difficult to use in conjunction with other parts of the framework. In addition, the Hashtable class is synchronized, which can result in slower performance compared to other implementations of the Map interface.

In general, it's recommended to use the Map interface or one of its implementations (such as HashMap or ConcurrentHashMap) instead of the Hashtable class.

Features of Hashtable

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn't provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

Declaration:

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

Type Parameters:

- **K** – the type of keys maintained by this map
- **V** – the type of mapped values

```
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        // Iterating over Map using for each loop
        for (Map.Entry<String, Integer> e : map.entrySet())

            // Printing key-value pairs
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }
```

Performing Various Operations using *Map Interface* and *HashMap Class*

Since Map is an interface, it can be used only with a class that implements this interface. Now, let's see how to perform a few frequently used operations on a Map using the widely used HashMap class. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the map

Adding Elements

In order to add an element to the map, we can use the put() method. However, the insertion order is not retained in the hashmap. Internally, for every element, a separate hash is generated and the elements are indexed based on this hash to make it more efficient.

```
// Default Initialization of a
    // Map
    Map<Integer, String> hm1 = new HashMap<>();

    // Initialization of a Map
    // using Generics
    Map<Integer, String> hm2
      = new HashMap<Integer, String>();

    // Inserting the Elements
    hm1.put(1, "Welcome");
    hm1.put(2, "In");
    hm1.put(3, "IACSD");

    hm2.put(new Integer(1), "Welcome");
    hm2.put(new Integer(2), "In");
    hm2.put(new Integer(3), "IACSD");

    System.out.println(hm1);
    System.out.println(hm2);
```

Changing Element

After adding the elements if we wish to change the element, it can be done by again adding the element with the put() method. Since the elements in the map are indexed using the keys, the

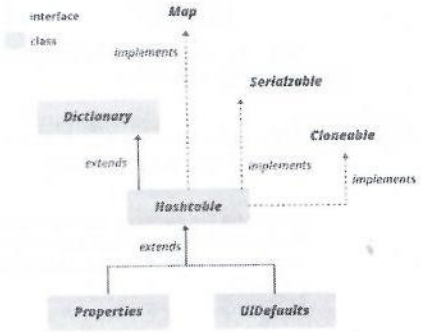
value of the key can be changed by simply inserting the updated value for the key for which we wish to change.

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

    // Initialization of a Map
    // using Generics
    Map<Integer, String> hm1
      = new HashMap<Integer, String>();

    // Inserting the Elements
    hm1.put(new Integer(1), "IACSD");
    hm1.put(new Integer(2), "IACSD");
    hm1.put(new Integer(3), "IACSD");

    System.out.println("Initial Map " + hm1);

    hm1.put(new Integer(2), "For");

    System.out.println("Updated Map " + hm1);
  }
```

Removing Elements

In order to remove an element from the Map, we can use the remove() method. This method takes the key value and removes the mapping for a key from this map if it is present in the map.

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

    // Initialization of a Map
    // using Generics
    Map<Integer, String> hm1
      = new HashMap<Integer, String>();

    // Inserting the Elements
    hm1.put(new Integer(1), "Welcome ");
    hm1.put(new Integer(2), "In");
    hm1.put(new Integer(3), "IACSD");
    hm1.put(new Integer(4), "Pune");

    // Initial Map
    System.out.println(hm1);

    hm1.remove(new Integer(4));

    // Final Map
    System.out.println(hm1);
```



Constructors:

In order to create a Hashtable, we need to import it from **java.util.Hashtable**.
There are various ways in which we can create a Hashtable.

1. **Hashtable():** This creates an empty hashtable with the default load factor of 0.75 and an initial capacity is 11
   Hashtable<K, V> ht = new Hashtable<K, V>();

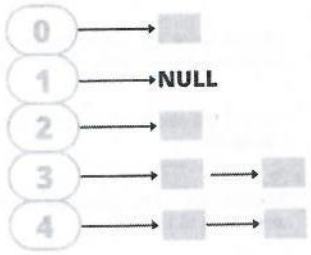
2. Hashtable(int initialCapacity): This creates a hash table that has an initial size specified by initialCapacity and the default load factor is 0.75.
Hashtable<K, V> ht = new Hashtable<K, V>(int initialCapacity);

3. Hashtable(int size, float fillRatio): This version creates a hash table that has an initial size specified by size and fill ratio specified by fillRatio. fill ratio: Basically, it determines how full a hash table can be before it is resized upward and its Value lies between 0.0 to 1.0.
Hashtable<K, V> ht = new Hashtable<K, V>(int size, float fillRatio);

4. Hashtable(Map<? extends K,? extends V> m): This creates a hash table that is initialized with the elements in m.
Hashtable<K, V> ht = new Hashtable<K, V>(Map m);

Internal Working of Hashtable

Hashtable datastructure is an array of buckets which stores the key/value pairs in them. It makes use of **hashCode() method** to determine which bucket the key/value pair should map. The hash function helps to determine the location for a given key in the bucket list. Generally, hashcode is a non-negative integer that is equal for equal Objects and may or may not be equal for unequal Objects. To determine whether two objects are equal or not, hashtable makes use of the equals() method.
It is possible that two unequal Objects have the same hashcode. This is called a **collision**. To resolve collisions, hashtable uses an array of lists. The pairs mapped to a single bucket (array index) are stored in a list and list reference is stored in the array index.



Methods of Hashtable

- **K** – The type of the keys in the map.
- **V** – The type of values mapped in the map.

METHOD	DESCRIPTION
clear()	Clears this hashtable so that it contains no keys.
clone()	Creates a shallow copy of this hashtable.
compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

METHOD	DESCRIPTION
contains(Object value)	Tests if some key maps into the specified value in this hashtable.
containsKey(Object key)	Tests if the specified object is a key in this hashtable.
containsValue(Object value)	Returns true if this hashtable maps one or more keys to this value.
elements()	Returns an enumeration of the values in this hashtable.
entrySet()	Returns a Set view of the mappings contained in this map.
equals(Object o)	Compares the specified Object with this Map for equality, as per the definition in the Map interface.
get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
hashCode()	Returns the hash code value for this Map as per the definition in the Map interface.
isEmpty()	Tests if this hashtable maps no keys to values.
keys()	Returns an enumeration of the keys in this hashtable.
keySet()	Returns a Set view of the keys contained in this map.
merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

METHOD	DESCRIPTION
put(K key, V value)	Maps the specified key to the specified value in this hashtable.
putAll(Map<? extends K,? extends V> t)	Copies all of the mappings from the specified map to this hashtable.
rehash()	Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently.
remove(Object key)	Removes the key (and its corresponding value) from this hashtable.
size()	Returns the number of keys in this hashtable.
toString()	Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).
values()	Returns a Collection view of the values contained in this map.

Methods declared in interface java.util.Map

METHOD	DESCRIPTION
forEach(BiConsumer<? super K,? super V> action)	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

METHOD	DESCRIPTION
putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
remove(Object key, Object value)	Removes the entry for the specified key only if it is currently mapped to the specified value.
replace(K key, V value)	Replaces the entry for the specified key only if it is currently mapped to some value.
replace(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if currently mapped to the specified value.
replaceAll(BiFunction<? super K,? super V,? extends V> function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Generics in Java

Generics means **parameterized types**. The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

The **Object** is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature. We will discuss that type of safety feature in later examples.
Generics in Java are similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc., use generics very well. There are some fundamental differences between the two approaches to generic types.

Types of Java Generics

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: *In Parameter type we can not use primitives like 'int','char' or 'double'.*

```
// Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T> {
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("IACSD Pune");
        System.out.println(sObj.getObject());
    }
}
```

We can also pass multiple Type parameters in Generic classes.

```
// Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
```

```
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.

```
// Java program to show working of user defined
// Generic functions

class Test {
    // A Generic method example
    static <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()
                           + " = " + element);
    }
```

Generics Work Only with Reference Types:

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like **int, char.**
Test<int> obj = new Test<int>(20);

The above line results in a compile-time error that can be resolved using type wrappers

to encapsulate a primitive type.

But primitive type arrays can be passed to the type parameter because arrays are

reference types.

ArrayList<int[]> a = new ArrayList<>();

Type Parameters in Java Generics

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.
2. Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Need of Concurrent Collections in java:-

As we already know Collections which is nothing but collections of Objects where we deals with the Objects using some pre-defined methods. But There are several problems which occurs when we use Collections concept in multi-threading. The problems which occurs while using Collections in Multi-threaded application:

- Most of the Collections classes objects (like ArrayList, LinkedList, HashMap etc) are non-synchronized in nature i.e. multiple threads can perform on a object at a time simultaneously. Therefore objects are not thread-safe.
- Very few Classes objects (like Vector, Stack, HashTable) are synchronized in nature i.e. at a time only one thread can perform on an Object. But here the problem is performance is low because at a time single thread execute an object and rest thread has to wait.
- The main problem is when one thread is iterating an Collections object then if another thread cant modify the content of the object. If another thread try to modify the content of object then we will get RuntimeException saying ConcurrentModificationException.
- Because of the above reason Collections classes is not suitable or we can say that good choice for Multi-threaded applications.

To overcome the above problem SUN microSystem introduced a new feature in JDK 1.5Version, which is nothing but Concurrent Collections

Concurrent Collections

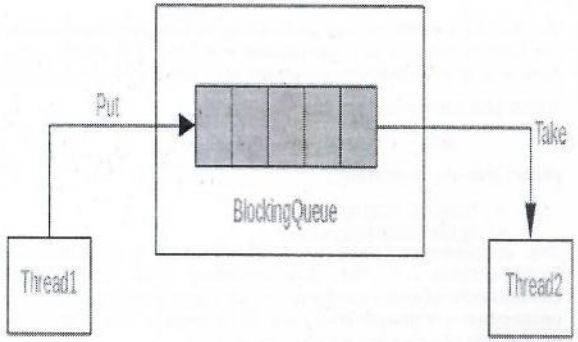
The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
- ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

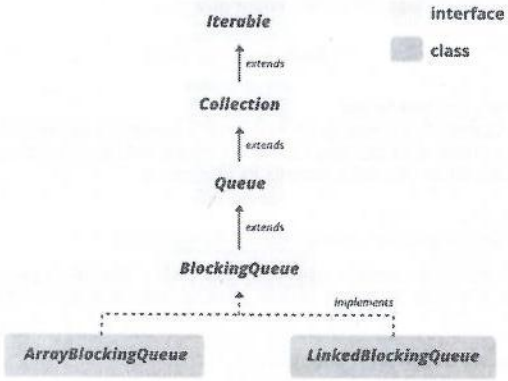
BlockingQueue Interface in Java

The **BlockingQueue interface** in Java is added in Java 1.5 along with various other concurrent Utility classes like ConcurrentHashMap, **Counting Semaphore**, CopyOnWriteArrrayList, etc. BlockingQueue interface supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely. Similarly, it blocks a thread trying to delete from an empty queue until some other threads insert an item. BlockingQueue does not accept a null value. If we try to enqueue the null item, then it throws **NullPointerException**.
Java provides several BlockingQueue implementations such as LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, **SynchronousQueue**, etc. Java BlockingQueue interface implementations are thread-safe. All methods of BlockingQueue are atomic in nature and use internal locks or other forms of concurrency control. Java 5 comes with BlockingQueue implementations in the **java.util.concurrent package**.

Usage of BlockingQueue



The Hierarchy of BlockingQueue



Declaration
public interface BlockingQueue<E> extends Queue<E>

Here, E is the type of elements stored in the Collection.

Classes that Implement BlockingQueue

We directly cannot provide an instance of BlockingQueue since it is an interface, so to utilize the functionality of the BlockingQueue, we need to make use of the classes implementing it. Also, to use BlockingQueue in your code, use this import statement.

import java.util.concurrent.BlockingQueue;

(or)

import java.util.concurrent.*;

- LinkedBlockingQueue
- ArrayBlockingQueue

The implementing class of BlockingDeque is LinkedBlockingDeque. This class is the implementation of the BlockingDeque and the linked list data structure. The LinkedBlockingDeque can be optionally bounded using a constructor, however, if the capacity is unspecified it is Integer.MAX_VALUE by default. The nodes are added dynamically at the time of insertion obeying the capacity constraints.

The syntax for creating objects:
BlockingQueue<?> objectName = new LinkedBlockingDeque<?>();

(or)

LinkedBlockingDeque<?> objectName = new LinkedBlockingDeque<?>();

Example: In the code given below we perform some basic operations on a LinkedBlockingDeque, like creating an object, adding elements, deleting elements, and using an iterator to traverse through the LinkedBlockingDeque.

BlockingQueue Types

The BlockingQueue are **two types**:
1. Unbounded Queue: The Capacity of the blocking queue will be set to Integer.MAX_VALUE. In the case of an unbounded blocking queue, the queue will never block because it could grow to a very large size. when you add elements its size grows.
Syntax:

BlockingQueue blockingQueue = new LinkedBlockingDeque();

2. Bounded Queue: The second type of queue is the bounded queue. In the case of a bounded queue you can create a queue passing the capacity of the queue in queues constructor:
Syntax:
// Creates a Blocking Queue with capacity 5

BlockingQueue blockingQueue = new LinkedBlockingDeque(5);

/ Java program that explains the internal
// implementation of BlockingQueue

import java.io.*;


```
import java.util.*;

class BlockingQueue<E> {

    // BlockingQueue using LinkedList structure
    // with a constraint on capacity
    private List<E> queue = new LinkedList<E>();

    // limit variable to define capacity
    private int limit = 10;

    // constructor of BlockingQueue
    public BlockingQueue(int limit) { this.limit = limit; }

    // enqueue method that throws Exception
    // when you try to insert after the limit
    public synchronized void enqueue(E item)
        throws InterruptedException
    {
        while (this.queue.size() == this.limit) {
            wait();
        }
        if (this.queue.size() == 0) {
            notifyAll();
        }
        this.queue.add(item);
    }

    // dequeue methods that throws Exception
    // when you try to remove element from an
    // empty queue
    public synchronized E dequeue()
        throws InterruptedException
    {
        while (this.queue.size() == 0) {
            wait();
        }
        if (this.queue.size() == this.limit) {
            notifyAll();
        }

        return this.queue.remove(0);
    }

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

/ Java Program to demonstrate usage of BlockingQueue

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

```
import java.util.*;

public class Demo {

    public static void main(String[] args)
        throws InterruptedException
    {

        // define capacity of ArrayBlockingQueue
        int capacity = 5;

        // create object of ArrayBlockingQueue
        BlockingQueue<String> queue
            = new ArrayBlockingQueue<String>(capacity);

        // Add elements to ArrayBlockingQueue using put
        // method
        queue.put("StarWars");
        queue.put("SuperMan");
        queue.put("Flash");
        queue.put("BatMan");
        queue.put("Avengers");

        // print Queue
        System.out.println("queue contains " + queue);

        // remove some elements
        queue.remove();
        queue.remove();

        // Add elements to ArrayBlockingQueue
        // using put method
        queue.put("CaptainAmerica");
        queue.put("Thor");

        System.out.println("queue contains " + queue);
    }
}
```

Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

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

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

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
```

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

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
```

```
            "Thread " + Thread.currentThread().getId()
            + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Thread Class vs Runnable Interface

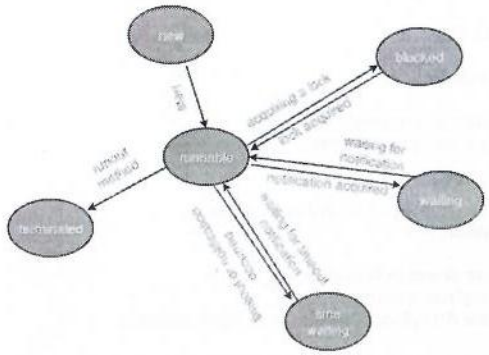
1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

Lifecycle and States of a Thread in Java

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

The diagram shown below represents various states of a thread at any instant in time.



Life Cycle of a thread

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it's in one of the following states:
    - Blocked
    - Waiting
4. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
    - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
    - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Implementing the Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as a summary of which is given below:

1. New
Declaration: public static final Thread.State NEW

Description: Thread state for a thread that has not yet started.

2. Runnable

Declaration: public static final Thread.State RUNNABLE

Description: Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

3. Blocked

Declaration: public static final Thread.State BLOCKED

Description: Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait().

4. Waiting

Declaration: public static final Thread.State WAITING

Description: Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

- Object.wait with no timeout
- Thread.join with no timeout
- LockSupport.park

5. Timed Waiting

Declaration: public static final Thread.State TIMED_WAITING

Description: Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

- Thread.sleep
- Object.wait with timeout
- Thread.join with timeout
- LockSupport.parkNanos
- LockSupport.parkUntil

6. Terminated

Declaration: public static final Thread.State TERMINATED

```
// Java program to demonstrate thread states
class thread implements Runnable {
    public void run()
    {
        // moving thread2 to timed waiting state
        try {
            Thread.sleep(1500);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(
            "State of thread1 while it called join() method on thread2 -"
            + Test.thread1.getState());
        try {
            Thread.sleep(200);
        }
```



```
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Test implements Runnable {
    public static Thread thread1;
    public static Test obj;

    public static void main(String[] args)
    {
        obj = new Test();
        thread1 = new Thread(obj);

        // thread1 created and is currently in the NEW
        // state.
        System.out.println(
            "State of thread1 after creating it - "
            + thread1.getState());
        thread1.start();

        // thread1 moved to Runnable state
        System.out.println(
            "State of thread1 after calling .start() method on it - "
            + thread1.getState());
    }

    public void run()
    {
        thread myThread = new thread();
        Thread thread2 = new Thread(myThread);

        // thread1 created and is currently in the NEW
        // state.
        System.out.println(
            "State of thread2 after creating it - "
            + thread2.getState());
        thread2.start();

        // thread2 moved to Runnable state
        System.out.println(
            "State of thread2 after calling .start() method on it - "
            + thread2.getState());

        // moving thread1 to timed waiting state
        try {
            // moving thread1 to timed waiting state
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println(
            "State of thread2 after calling .sleep() method on it - "
            + thread2.getState());

        try {
            // waiting for thread2 to die
            thread2.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(
            "State of thread2 when it has finished it's execution - "
            + thread2.getState());
    }
}
```

Explanation: When a new thread is created, the thread is in the NEW state. When the start() method is called on a thread, the thread scheduler moves it to Runnable state. Whenever the join() method is called on a thread instance, the current thread executing that statement will wait for this thread to move to the Terminated state. So, before the final statement is printed on the console, the program calls join() on thread2 making the thread1 wait while thread2 completes its execution and is moved to the Terminated state. thread1 goes to Waiting state because it is waiting for thread2 to complete its execution as it has called join on thread2.

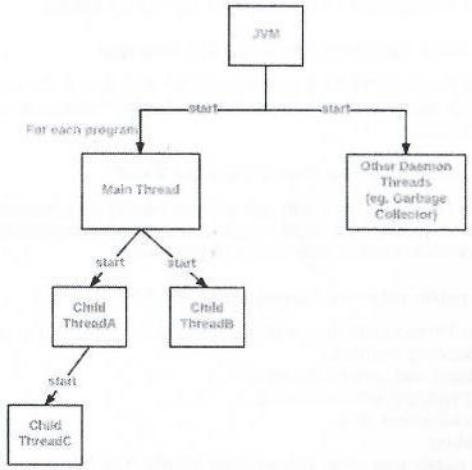
Main thread in Java

Java provides built-in support for multithreaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program because it is the one that is executed when our program begins.

There are certain properties associated with the main thread which are as follows:

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

The flow diagram is as follows:



How to control Main thread

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method *currentThread( )* which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

/ Java program to control the Main Thread

```
// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Main class extending thread class
public class Test extends Thread {

    // Main driver method
    public static void main(String[] args)
    {

        // Getting reference to Main thread
        Thread t = Thread.currentThread();

        // Getting name of Main thread
        System.out.println("Current thread: "
                           + t.getName());

        // Changing the name of Main thread
        t.setName("Geeks");
        System.out.println("After name change: "
```

```
                + t.getName());

        // Getting priority of Main thread
        System.out.println("Main thread priority: "
                + t.getPriority());

        // Setting priority of Main thread to MAX(10)
        t.setPriority(MAX_PRIORITY);

        // Print and display the main thread priority
        System.out.println("Main thread new priority: "
                + t.getPriority());

        for (int i = 0; i < 5; i++) {
            System.out.println("Main thread");
        }

        // Main thread creating a child thread
        Thread ct = new Thread() {
            // run() method of a thread
            public void run()
            {

                for (int i = 0; i < 5; i++) {
                    System.out.println("Child thread");
                }
            }
        };

        // Getting priority of child thread
        // which will be inherited from Main thread
        // as it is created by Main thread
        System.out.println("Child thread priority: "
                + ct.getPriority());

        // Setting priority of Main thread to MIN(1)
        ct.setPriority(MIN_PRIORITY);

        System.out.println("Child thread new priority: "
                + ct.getPriority());

        // Starting child thread
        ct.start();
    }
}

// Class 2
// Helper class extending Thread class
// Child Thread class
class ChildThread extends Thread {

    @Override public void run()
```



```
    {

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

            // Print statement whenever child thread is
            // called
            System.out.println("Child thread");
        }
    }
}
```

Now let us discuss the relationship between the main() method and the main thread in Java. For each program, a Main thread is created by JVM(Java Virtual Machine). The "Main" thread first verifies the existence of the main() method, and then it initializes the class. Note that from JDK 6, main() method is mandatory in a standalone java application.

Deadlocking with use of Main Thread(only single thread)

We can create a deadlock by just using the Main thread, i.e. by just using a single thread.

```
// Java program to demonstrate deadlock
// using Main thread

// Main class
public class Demo {

  // Main driver method
  public static void main(String[] args) {

    // Try block to check for exceptions
    try {

      // Print statement
      System.out.println("Entering into Deadlock");

      // Joining the current thread
      Thread.currentThread().join();

      // This statement will never execute
      System.out.println("This statement will never execute");
    }

    // Catch block to handle the exceptions
    catch (InterruptedException e) {

      // Display the exception along with line number
      // using printStackTrace() method
      e.printStackTrace();
    }
  }
}
```

The statement "Thread.currentThread().join()", will tell Main thread to wait for this thread(i.e. wait for itself) to die. Thus Main thread wait for itself to die, which is nothing but a deadlock.

Thread Priority in Multithreading

As we already know java being completely object-oriented works within a multithreading environment in which thread scheduler assigns the processor to a thread based on the priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

Priorities in threads is a concept where each thread is having a priority which in layman's language one can say every object is having priority here which is represented by numbers ranging from 1 to 10.

- The default priority is set to 5 as excepted.
- Minimum priority is set to 1.
- Maximum priority is set to 10.

Here 3 constants are defined in it namely as follows:

1. public static int NORM_PRIORITY
2. public static int MIN_PRIORITY
3. public static int MAX_PRIORITY

Let us discuss it with an example to get how internally the work is getting executed. Here we will be using the knowledge gathered above as follows:

- We will use *currentThread()* method to get the name of the current thread. User can also use *setName()* method if he/she wants to make names of thread as per choice for understanding purposes.
- *getName()* method will be used to get the name of the thread.

The accepted value of priority for a thread is in the range of 1 to 10.

Let us do discuss how to get and set priority of a thread in java.

1. **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
2. **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

```
// Java Program to Illustrate Priorities in Multithreading
// via help of getPriority() and setPriority() method

// Importing required classes
import java.lang.*;

// Main class
class ThreadDemo extends Thread {

    // Method 1
    // run() method for the thread that is called
    // as soon as start() is invoked for thread in main()
    public void run()
    {
        // Print statement
        System.out.println("Inside run method");
    }

    // Main driver method
```



```
    public static void main(String[] args)
    {
        // Creating random threads
        // with the help of above class
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        // Thread 1
        // Display the priority of above thread
        // using getPriority() method
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // Thread 1
        // Display the priority of above thread
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // Thread 3
        System.out.println("t3 thread priority : "
                           + t3.getPriority());

        // Setting priorities of above threads by
        // passing integer arguments
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

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

        // 2
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // 5
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // 8
        System.out.println("t3 thread priority : "
                           + t3.getPriority());

        // Main thread

        // Displays the name of
        // currently executing Thread
        System.out.println(
            "Currently Executing Thread : "
            + Thread.currentThread().getName());
```

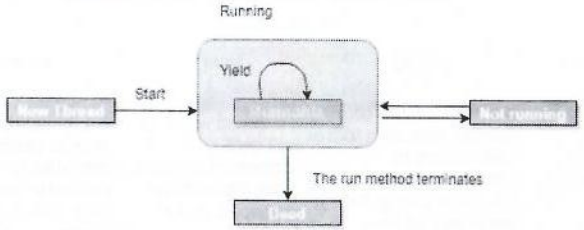
```
        System.out.println(
          "Main thread priority : "
          + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);

        System.out.println(
          "Main thread priority : "
          + Thread.currentThread().getPriority());
    }
}
```

Java Concurrency – yield(), sleep() and join() Methods

1. yield() Method

Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. The completion time for thread t1 is 5 hours and the completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent the execution of a thread in between if something important is pending. yield() helps us in doing so.

The **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



Use of yield method:

- Whenever a thread calls **java.lang.Thread.yield** method gives hint to the thread scheduler that it is ready to pause its execution. The thread scheduler is free to ignore this hint.
- If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.
- Once a thread has executed the yield method and there are many threads with the same priority is waiting for the processor, then we can't specify which thread will get the execution chance first.

- The thread which executes the yield method will enter in the Runnable state from Running state.
- Once a thread pauses its execution, we can't specify when it will get a chance again it depends on the thread scheduler.
- The underlying platform must provide support for preemptive scheduling if we are using the yield method.

2. sleep() Method

This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Syntax:

```
//  sleep for the specified number of milliseconds
public static void sleep(long millis) throws InterruptedException


//sleep for the specified number of milliseconds plus nano seconds
public static void sleep(long millis, int nanos)
                    throws InterruptedException
```

```
// Java program to illustrate
// sleep() method in Java

import java.lang.*;

public class SleepDemo implements Runnable {
    Thread t;
    public void run()
    {
        for (int i = 0; i < 4; i++) {
            System.out.println(
                Thread.currentThread().getName() + " "
                + i);
            try {
                // thread to sleep for 1000 milliseconds
                Thread.sleep(1000);
            }

            catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        Thread t = new Thread(new SleepDemo());

        // call run() function
        t.start();
```

```
        Thread t2 = new Thread(new SleepDemo());

        // call run() function
        t2.start();
    }
}
```

Note:

- *Based on the requirement we can make a thread to be in a sleeping state for a specified period of time*
- *Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power-saving mode).*

3. join() Method

The join() method of a Thread instance is used to join the start of a thread's execution to the end of another thread's execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing. The join() method waits at most this many milliseconds for this thread to die. A timeout of 0 means to wait forever

Syntax:

```
// waits for this thread to die.
public final void join() throws InterruptedException

// waits at most this much milliseconds for this thread to die
public final void join(long millis)
        throws InterruptedException

// waits at most milliseconds plus nanoseconds for this thread to die.
The java.lang.Thread.join(long millis, int nanos)
```

```
// Java program to illustrate join() method in Java

import java.lang.*;

public class JoinDemo implements Runnable {
    public void run()
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: "
                + t.getName());

        // checks if current thread is alive
```



```
        System.out.println("Is Alive? " + t.isAlive());
    }

    public static void main(String args[]) throws Exception
    {
        Thread t = new Thread(new JoinDemo());
        t.start();

        // Waits for 1000ms this thread to die.
        t.join(1000);

        System.out.println("\nJoining after 1000"
                + " milliseconds: \n");
        System.out.println("Current thread: "
                + t.getName());

        // Checks if this thread is alive
        System.out.println("Is alive? " + t.isAlive());
    }
}
```

Note:

- *If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.*
- *Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.*

Comparison of yield(), join(), sleep() Methods

Property	yield()	join()	sleep()
Purpose	If a thread wants to pass its execution to give chance to remaining threads of the same priority then we should go for yield()	If a thread wants to wait until completing of some other thread then we should go for join()	If a thread does not want to perform any operation for a particular amount of time, then it goes for sleep()
Is it overloaded?	NO	YES	YES
Is it final?	NO	YES	NO
Is it throws?	NO	YES	YES

Property	yield()	join()	sleep()
Is it native?	YES	NO	sleep(long ms)->native & sleep (long ms, int ns)-> non native
Is it static?	YES	NO	YES

Inter-thread Communication in Java

Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Note: *Inter-thread communication is also known as* ***Cooperation in Java.***

What is Polling, and what are the problems with it?

The process of testing a condition repeatedly till it becomes true is known as polling. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, a certain action is taken. This wastes many CPU cycles and makes the implementation inefficient.

For example, in a classic queuing problem where one thread is producing data, and the other is consuming it.

How Java multi-threading tackles this problem?

To avoid polling, Java uses three methods, namely, **wait(), notify(), and notifyAll().** All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

- **wait():** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- **notify():** It wakes up one single thread called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
- **notifyAll():** It wakes up all the threads called wait() on the same object.

```
/ Java program to demonstrate inter-thread communication
// (wait(), join() and notify())

import java.util.Scanner;

public class threadexample
{
    public static void main(String[] args) throws InterruptedException
    {
        final PC pc = new PC();

        // Create a thread object that calls pc.produce()
        Thread t1 = new Thread(new Runnable()
        {
            @Override
```



```
            public void run()
            {
                try
                {
                    pc.produce();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Create another thread object that calls
        // pc.consume()
        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                try
                {
                    pc.consume();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // t1 finishes before t2
        t1.join();
        t2.join();
    }

    // PC (Produce Consumer) class with produce() and
    // consume() methods.
    public static class PC
    {
        // Prints a string and waits for consume()
        public void produce()throws InterruptedException
        {
            // synchronized block ensures only one thread
            // running at a time.
            synchronized(this)
            {
                System.out.println("producer thread running");
```

```
            // releases the lock on shared resource
            wait();

            // and waits till some other method invokes notify().
            System.out.println("Resumed");
        }
    }

    // Sleeps for some time and waits for a key press. After key
    // is pressed, it notifies produce().
    public void consume()throws InterruptedException
    {
        // this makes the produce thread to run first.
        Thread.sleep(1000);
        Scanner s = new Scanner(System.in);

        // synchronized block ensures only one thread
        // running at a time.
        synchronized(this)
        {
            System.out.println("Waiting for return key.");
            s.nextLine();
            System.out.println("Return key pressed");

            // notifies the produce thread that it
            // can wake up.
            notify();

            // Sleep
            Thread.sleep(2000);
        }
    }
  }
}
```

As monstrous as it seems, it is a piece of cake if you go through it twice.

1. In the main class, a new PC object is created.
2. It runs produce and consume methods of PC objects using two different threads, namely t1 and t2, and waits for these threads to finish.

Let's understand how our produce and consume method works.

- First of all, the use of a synchronized block ensures that only one thread at a time runs. Also, since there is a sleep method just at the beginning of consume loop, the producing thread gets a kickstart.
- When the wait is called in produce method, it does two things. Firstly it releases the lock it holds on the PC object. Secondly, it makes the produce thread go on a waiting state until all other threads have terminated. It can again acquire a lock on a PC object, and some other method wakes it up by invoking notify or notifyAll on the same object.
- Therefore we see that as soon as the wait is called, the control transfers to consume thread, and it prints -"Waiting for return key."

- After we press the return key, consume method invokes notify(). It also does two things- Firstly, unlike wait(), it does not release the lock on shared resources therefore for getting the desired result, it is advised to use notify only at the end of your method. Secondly, it notifies the waiting threads that they can now wake up but only after the current method terminates.
- As you might have observed that even after notifying, the control does not immediately pass over to the produce thread. The reason for it is that we have called Thread.sleep() after notify(). We already know that the consume thread is holding a lock on a PC object. Another thread cannot access it until it has released the lock. Hence only after the consume thread finishes its sleep time and after that terminates by itself, the produce thread cannot take back the control.
- After a 2 second pause, the program terminates to its completion.

If you are still confused as to why we have used notify in consume thread, try removing it and running your program again, as you must have noticed now that the program never terminates.

The reason for this is straightforward. When you called to wait on the produce thread, it went on waiting and never terminated. Since a program runs till all its threads have terminated, it runs on and on.

There is a second way around this problem. You can use a second variant of wait().

void wait(long timeout)

This would make the calling thread sleep only for a time specified.

Thread Pools in Java

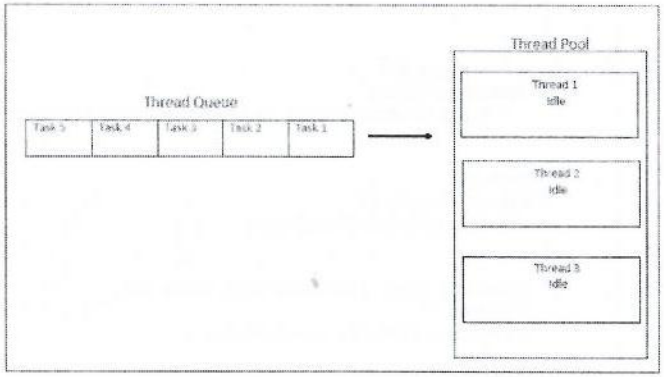
Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

What is ThreadPool in Java?

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.



Executor Thread Pool Methods

Method		Description
newFixedThreadPool(int)	--	Creates a fixed size thread pool.
newCachedThreadPool()	--	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available
newSingleThreadExecutor()	--	Creates a single thread.

In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

Thread Pool Example

In the following tutorial, we will look at a basic example of thread pool executor-FixedThreadPool.

Steps to be followed

1. Create a task(Runnable Object) to execute
2. Create Executor Pool using Executors
3. Pass tasks to Executor Pool
4. Shutdown the Executor Pool

```
// Java program to illustrate
// ThreadPool
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// Task class to be executed (Step 1)
class Task implements Runnable
{
    private String name;
```

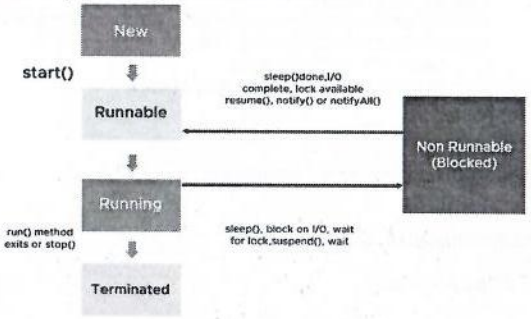


```
    public Task(String s)
    {
        name = s;
    }

    // Prints task name and sleeps for 1s
    // This Whole process is repeated 5 times
    public void run()
    {
        try
        {
            for (int i = 0; i<=5; i++)
            {
                if (i==0)
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Initialization Time for"
                            + " task name - "+ name +" = " +ft.format(d));
                    //prints the initialization time for every task
                }
                else
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Executing Time for task name - "+
                            name +" = " +ft.format(d));
                    // prints the execution time for every task
                }
                Thread.sleep(1000);
            }
            System.out.println(name+" complete");
        }

        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
public class Test
{
    // Maximum number of threads in thread pool
    static final int MAX_T = 3;

    public static void main(String[] args)
    {
        // creates five tasks
        Runnable r1 = new Task("task 1");
        Runnable r2 = new Task("task 2");
        Runnable r3 = new Task("task 3");
        Runnable r4 = new Task("task 4");
```

```
        Runnable r5 = new Task("task 5");

        // creates a thread pool with MAX_T no. of
        // threads as the fixed pool size(Step 2)
        ExecutorService pool = Executors.newFixedThreadPool(MAX_T);

        // passes the Task objects to the pool to execute (Step 3)
        pool.execute(r1);
        pool.execute(r2);
        pool.execute(r3);
        pool.execute(r4);
        pool.execute(r5);

        // pool shutdown ( Step 4)
        pool.shutdown();
    }
}
```

Java.lang.Thread Class in Java

Thread a line of execution within a program. Each program can have multiple associated threads. Each thread has a priority which is used by the thread scheduler to determine which thread must run first. Java provides a thread class that has various method calls in order to manage the behavior of threads by providing constructors and methods to perform operations on threads.



Ways of creating threads

1. Creating own class which is extending to parent Thread class
2. Implementing the Runnable interface.

Below are the pseudo-codes that one can refer to get a better picture about thread henceforth Thread class.

```
// Way 1
// Creating thread By Extending To Thread class

class MyThread extends Thread {

    // Method 1
    // Run() method for our thread
```



```
    public void run()
    {

        // Print statement
        System.out.println(
            "Thread is running created by extending to parent Thread class");
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of our thread class inside main()
        // method
        MyThread myThread = new MyThread();

        // Starting the thread
        myThread.start();
    }
}

class ThreadUsingInterface implements Runnable {

    // Method 1
    // run() method for the thread
    public void run()
    {

        // Print statement
        System.out.println("Thread is created using Runnable interface");
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of our thread class inside main()
        // method
        ThreadUsingInterface obj = new ThreadUsingInterface();

        // Passing the object to thread in main()
        Thread myThread = new Thread(obj);

        // Starting the thread
        myThread.start();
    }
}
```

Thread Class in Java

A thread is a program that starts with a method() frequently used in this class only known as the start() method. This method looks out for the run() method which is also a method of this class and begins executing the body of the run() method. Here, keep an eye over the sleep() method which will be discussed later below.

Note: *Every class that is used as thread must implement Runnable interface and over ride it's run method.*

Syntax:

public class Thread extends Object implements Runnable

Constructors of this class are as follows:

Constructor	Action Performed
Thread()	Allocates a new Thread object.
Thread(Runnable target)	Allocates a new Thread object.
Thread(Runnable target, String name)	Allocates a new Thread object.
Thread(String name)	Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target)	Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target, String name)	Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by a group.
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.
Thread(ThreadGroup group, String name)	Allocates a new Thread object.

Methods of Thread class:

Now let us do discuss all the **methods** of this class are illustrated as follows:

Methods	Action Performed
activeCount()	Returns an estimate of the number of active threads in the current thread's thread group and its subgroups
checkAccess()	Determines if the currently running thread has permission to modify this thread
clone()	Throws CloneNotSupportedException as a Thread can not be meaningfully cloned
currentThread()	Returns a reference to the currently executing thread object
dumpStack()	Prints a stack trace of the current thread to the standard error stream
enumerate(Thread[] tarray)	Copies into the specified array every active thread in the current thread's thread group and its subgroups
getAllStackTraces()	Returns a map of stack traces for all live threads
getContextClassLoader()	Returns the context ClassLoader for this Thread
getDefaultUncaughtExceptionHandler()	Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception
getId()	Returns the identifier of this Thread
getName()	Returns this thread's name
getPriority()	Returns this thread's priority
getStackTrace()	Returns an array of stack trace elements representing the stack dump of this thread
getState()	Returns the state of this thread

Methods	Action Performed
getThreadGroup()	Returns the thread group to which this thread belongs
getUncaughtExceptionHandler()	Returns the handler invoked when this thread abruptly terminates due to an uncaught exception
holdsLock(Object obj)	Returns true if and only if the current thread holds the monitor lock on the specified object
interrupt()	Interrupts this thread
interrupted()	Tests whether the current thread has been interrupted
isAlive()	Tests if this thread is alive
isDaemon()	Tests if this thread is a daemon thread
isInterrupted()	Tests whether this thread has been interrupted
join()	Waits for this thread to die
join(long millis)	Waits at most millis milliseconds for this thread to die
run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns
setContextClassLoader(ClassLoader cl)	Sets the context ClassLoader for this Thread
setDaemon(boolean on)	Marks this thread as either a daemon thread or a user thread

Methods	Action Performed
setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread
setName(String name)	Changes the name of this thread to be equal to the argument name.
setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Set the handler invoked when this thread abruptly terminates due to an uncaught exception
setPriority(int newPriority)	Changes the priority of this thread
sleep(long millis)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers
start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread
toString()	Returns a string representation of this thread, including the thread's name, priority, and thread group
yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor

Also do remember there are certain **methods inherited from class java. lang.Object that are as follows:**

1. *equals() Method*
2. *finalize() Method*
3. *getClass() Method*
4. *hashCode() Method*
5. *notify() Method*
6. *notifyAll() Method*
7. *toString() Method*
8. wait() Method

Runnable interface in Java

ava.lang.Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass

Thread and implement Runnable. There is no need of subclassing a Thread when a task can be done by overriding only run() method of Runnable.

Steps to create a new thread using Runnable

1. Create a Runnable implementer and implement the run() method.
2. Instantiate the Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instances.
3. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start() creates a new Thread that executes the code written in run(). Calling run() directly doesn't create and start a new Thread, it will run in the same thread. To start a new line of execution, call start() on the thread.

```
public class RunnableDemo {

    public static void main(String[] args)
    {
        System.out.println("Main thread is- "
                + Thread.currentThread().getName());
        Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
        t1.start();
    }

    private class RunnableImpl implements Runnable {

        public void run()
        {
            System.out.println(Thread.currentThread().getName()
                    + ", executing run() method!");
        }
    }
}
```

Deadlock in Java Multithreading

synchronized keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock. It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**. Below is a simple example of Deadlock condition.

```
// Java program to illustrate Deadlock
// in multithreading.
class Util
{
    // Util class to sleep a thread
    static void sleep(long millis)
    {
        try
        {
            Thread.sleep(millis);
```



```
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

// This class is shared by both threads
class Shared
{
    // first synchronized method
    synchronized void test1(Shared s2)
    {
        System.out.println("test1-begin");
        Util.sleep(1000);

        // taking object lock of s2 enters
        // into test2 method
        s2.test2();
        System.out.println("test1-end");
    }

    // second synchronized method
    synchronized void test2()
    {
        System.out.println("test2-begin");
        Util.sleep(1000);
        // taking object lock of s1 enters
        // into test1 method
        System.out.println("test2-end");
    }
}


class Thread1 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread1(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s1 enters
```

```
        // into test1 method
        s1.test1(s2);
    }
}


class Thread2 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread2(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s2
        // enters into test2 method
        s2.test1(s1);
    }
}


public class Deadlock
{
    public static void main(String[] args)
    {
        // creating one object
        Shared s1 = new Shared();

        // creating second object
        Shared s2 = new Shared();

        // creating first thread and starting it
        Thread1 t1 = new Thread1(s1, s2);
        t1.start();

        // creating second thread and starting it
        Thread2 t2 = new Thread2(s1, s2);
        t2.start();

        // sleeping main thread
        Util.sleep(2000);
    }
}
```

It is not recommended to run the above program with online IDE. We can copy the source code

and run it on our local machine. We can see that it runs for indefinite time, because threads are in deadlock condition and doesn't let code to execute. Now let's see step by step what is happening there.

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test1 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will be look like execution is paused.

Detect Dead Lock condition

We can also detect deadlock by running this program on cmd. We have to collect Thread Dump. Command to collect depends on OS type. If we are using Windows and Java 8, command is jcmd $PID Thread.print
We can get P ID by running jps command

Avoid Dead Lock condition

We can avoid dead lock condition by knowing its possibilities. It's a very complex process and not easy to catch. But still if we try, we can avoid this. There are some methods by which we can avoid this condition. We can't completely remove its possibility but we can reduce.

- **Avoid Nested Locks :** This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
- **Avoid Unnecessary Locks :** We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
- **Using thread join :** Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join with maximum time you think the execution will take.

Important Points :

- If threads are waiting for each other to finish, then the condition is known as Deadlock.
- Deadlock condition is a complex condition which occurs only in case of multiple threads.
- Deadlock condition can break our code at run time and can destroy business logic.
- We should avoid this condition as much as we can.

Daemon Thread in Java

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

Example of Daemon Thread in Java: Garbage collection in Java (gc), finalizer, etc.

Properties of Java Daemon Thread

- They can not prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution.
- If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
- It is an utmost low priority thread.

Default Nature of Daemon Thread

By default, the main thread is always non-daemon but for all the remaining threads, daemon nature will be inherited from parent to child. That is, if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

Note: *Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.*

Methods of Daemon Thread

1. void setDaemon(boolean status):
This method marks the current thread as a daemon thread or user thread. For example, if I have a user thread tU then tU.setDaemon(true) would make it a Daemon thread. On the other hand, if I have a Daemon thread tD then calling tD.setDaemon(false) would make it a user thread.

Syntax:
public final void setDaemon(boolean on)

Parameters:

- **on:** If true, marks this thread as a daemon thread.

Exceptions:

- **IllegalThreadStateException**: if only this thread is active.
- **SecurityException:** if the current thread cannot modify this thread.

2. boolean isDaemon():
This method is used to check that the current thread is a daemon. It returns true if the thread is Daemon. Else, it returns false.

Syntax:
public final boolean isDaemon()

Returns:
This method returns true if this thread is a daemon thread; false otherwise

```
// Java program to demonstrate the usage of
// setDaemon() and isDaemon() method.

public class DaemonThread extends Thread
{
    public DaemonThread(String name){
```



```
        super(name);
    }

    public void run()
    {
        // Checking whether the thread is Daemon or not
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() +" is Daemon thread");
        }

        else
        {
            System.out.println(getName() + " is User thread");
        }
    }

    public static void main(String[] args)
    {

        DaemonThread t1 = new DaemonThread("t1");
        DaemonThread t2 = new DaemonThread("t2");
        DaemonThread t3 = new DaemonThread("t3");

        // Setting user thread t1 to Daemon
        t1.setDaemon(true);

        // starting first 2 threads
        t1.start();
        t2.start();

        // Setting user thread t3 to Daemon
        t3.setDaemon(true);
        t3.start();
    }
}
```

Exceptions in a Daemon thread

If you call the setDaemon() method after starting the thread, it would throw **IllegalThreadStateException**.
This clearly shows that we cannot call the setDaemon() method after starting the thread.

Daemon vs. User Threads

1. **Priority:** When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
2. **Usage:** Daemon thread is to provide services to user thread for background supporting task.

Synchronization in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time. Java provides a way of creating threads and synchronizing their tasks using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
  // Access shared variables and other
  // shared resources
}
```

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

Following is an example of multi-threading with synchronized.

```
// A Java program to demonstrate working of
// synchronized.

import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
   public void send(String msg)
   {
      System.out.println("Sending\t"  + msg );
      try
      {
         Thread.sleep(1000);
      }
      catch (Exception e)
      {
         System.out.println("Thread  interrupted.");
      }
      System.out.println("\n" + msg + "Sent");
   }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
   private String msg;
   Sender  sender;

   // Receives a message object and a string
   // message to be sent
   ThreadedSend(String m,  Sender obj)
   {
      msg = m;
      sender = obj;
   }

   public void run()
   {
      // Only one thread can send a message
      // at a time.
      synchronized(sender)
      {
         // synchronizing the send object
         sender.send(msg);
      }
   }
}

// Driver class
class SyncDemo
{
   public static void main(String args[])
   {
      Sender send = new Sender();
      ThreadedSend S1 =
         new ThreadedSend( " Hi " , send );
      ThreadedSend S2 =
         new ThreadedSend( " Bye " , send );

      // Start two threads of ThreadedSend type
      S1.start();
      S2.start();

      // wait for threads to end
      try
      {
         S1.join();
         S2.join();
      }
      catch(Exception e)
      {
```

```
            System.out.println("Interrupted");
        }
    }
}
```

The output is the same every time we run the program.

In the above example, we choose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized**, producing the same result. Then we don't have to synchronize the Message object inside the run() method in ThreadedSend class.

```
// An alternate implementation to demonstrate
// that we can use synchronized with method also.

class Sender {
  public synchronized void send(String msg)
  {
    System.out.println("Sending\t" + msg);
    try {
      Thread.sleep(1000);
    }
    catch (Exception e) {
      System.out.println("Thread interrupted.");
    }
    System.out.println("\n" + msg + "Sent");
  }
}
```

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods make this possible.

Inner Class

In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is purely object-oriented so bringing it closer to the real world. Now geeks you must be wondering why they were introduced?

There are certain advantages associated with inner classes are as follows:

- Making code clean and readable.
- Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.
- Optimizing the code module.

We do use them often as we go advance in java object-oriented programming where we want certain operations to be performed, granting access to limited classes and many more which will be clear as we do discuss and implement all types of inner classes in Java.

In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is purely object-oriented so bringing it closer to the real world. Now geeks you must be wondering why they were introduced?

There are certain advantages associated with inner classes are as follows:

- Making code clean and readable.
- Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.
- Optimizing the code module.

We do use them often as we go advance in java object-oriented programming where we want certain operations to be performed, granting access to limited classes and many more which will be clear as we do discuss and implement all types of inner classes in Java.

Types of Inner Classes

There are basically four types of inner classes in java.

1. Nested Inner Class
2. Method Local Inner Classes
3. Static Nested Classes
4. Anonymous Inner Classes

Let us discuss each of the above following types sequentially in-depth alongside a clean java program which is very crucial at every step as it becomes quite tricky as we adhere forwards.

Type 1: Nested Inner Class

It can access any private instance variable of the outer class. Like any other instance variable, we can have access modifier private, protected, public, and default modifier. Like class, an interface can also be nested and can have access specifiers.

```
/ Java Program to Demonstrate Nested class

// Class 1
// Helper classes
class Outer {

    // Class 2
    // Simple nested inner class
```

```
    class Inner {

        // show() method of inner class
        public void show()
        {

            // Print statement
            System.out.println("In a nested class method");
        }
    }
}

// Class 2
// Main class
class Main {

    // Main driver method
    public static void main(String[] args)
    {

        // Note how inner class object is created inside
        // main()
        Outer.Inner in = new Outer().new Inner();

        // Calling show() method over above object created
        in.show();
    }
}
//------------------------------Example -2-----------------------------

// Java Program to Demonstrate Nested class
// Where Error is thrown

// Class 1
// Outer class
class Outer {

    // Method defined inside outer class
    void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");
    }

    // Class 2
    // Inner class
    class Inner {

        // Main driver method
        public static void main(String[] args)
        {

            // Display message for better readability
            System.out.println("inside inner class Method");
        }
    }
}
```


Note: ***We can not have a static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself. For example, the following program doesn't compile. But Since JAVA Version 16 we can have static members in our inner class also.***

Type 2: Method Local Inner Classes

Inner class can be declared within a method of an outer class which we will be illustrating in the below example where Inner is an inner class in outerMethod().

```
class Outer {

    // Method inside outer class
    void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");

        // Class 2
        // Inner class
        // It is local to outerMethod()
        class Inner {

            // Method defined inside inner class
            void innerMethod()
            {

                // Print statement whenever inner class is
                // called
                System.out.println("inside innerMethod");
            }
        }

        // Creating object of inner class
        Inner y = new Inner();

        // Calling over method defined inside it
        y.innerMethod();
    }
}

// Class 3
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of outer class inside main()
        // method
        Outer x = new Outer();

        // Calling over the same method
        // as we did for inner class above
        x.outerMethod();
    }
}
```

```
class Outer {
   void outerMethod() {
      int x = 98;
      System.out.println("inside outerMethod");
      class Inner {
         void innerMethod() {
            System.out.println("x= "+x);
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
   public static void main(String[] args) {
      Outer x=new Outer();
      x.outerMethod();
   }
}
```

Note: ***Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in method local inner class.***

Type 3: Static Nested Classes

Static nested classes are not technically inner classes. They are like a static member of outer class.

```
// Java Program to Illustrate Static Nested Classes

// Importing required classes
import java.util.*;

// Class 1
// Outer class
class Outer {

    // Method
    private static void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");
    }

    // Class 2
    // Static inner class
    static class Inner {

        public static void display()
        {

            // Print statement
            System.out.println("inside inner class Method");

            // Calling method inside main() method
            outerMethod();
        }
    }
}
```



```
// Class 3
// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {

        // Calling method static display method rather than an instance of that
class.
        Outer.Inner.display();
    }
}
```

Type 4: Anonymous Inner Classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

- As a subclass of the specified type
- As an implementer of the specified interface

```
// Importing required classes
import java.util.*;

// Class 1
// Helper class
class Demo {

    // Method of helper class
    void show()
    {
        // Print statement
        System.out.println(
            "i am in show method of super class");
    }
}

// Class 2
// Main class
class FlavorlDemo {

    //  An anonymous class with Demo as base class
    static Demo d = new Demo() {
        // Method 1
        // show() method
        void show()
        {
            // Calling method show() via super keyword
            // which refers to parent class
            super.show();

            // Print statement
            System.out.println("i am in FlavorlDemo class");
        }
    };

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {
```

```
        // Calling show() method inside main() method
        d.show();
    }
}
```

The benefits of using inner classes in Java are:

Encapsulation: Inner classes can access private variables and methods of the outer class. This helps to achieve encapsulation and improves code readability.
Code Organization: Inner classes allow you to group related code together in one place. This makes your code easier to understand and maintain.
Better Access Control: Inner classes can be declared as private, which means that they can only be accessed within the outer class. This provides better access control and improves code security.
Callbacks: Inner classes are often used for implementing callbacks in event-driven programming. They provide a convenient way to define and implement a callback function within the context of the outer class.
Polymorphism: Inner classes can be used to implement polymorphism. You can define a class hierarchy within the outer class and then create objects of the inner classes that implement the different subclasses.
Reduced Code Complexity: Inner classes can reduce the complexity of your code by encapsulating complex logic and data structures within the context of the outer class.
Overall, the use of inner classes can lead to more modular, maintainable, and flexible code.

Lambda Expressions in Java 8

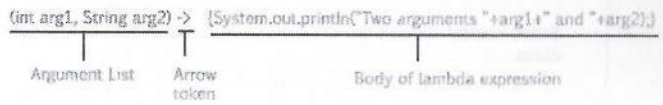
Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces
lambda expressions are added in Java 8 and provide below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.

```
// Java program to demonstrate lambda expressions
// to implement a user defined functional interface.

// A sample functional interface (An interface with
// single abstract method
interface FuncInterface
{
    // An abstract function
    void abstractFun(int x);

    // A non-abstract (or default) function
```



```
    default void normalFun()
    {
        System.out.println("Hello");
    }
}

class Test
{
    public static void main(String args[])
    {
        // lambda expression to implement above
        // functional interface. This interface
        // by default implements abstractFun()
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls above lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}
```



Syntax:-
lambda operator -> body

where lambda operator can be:

- **Zero parameter:**
  () -> System.out.println("Zero parameter lambda");
- **One parameter:–**
  (p) -> System.out.println("One parameter: " + p);

  It is not mandatory to use parentheses, if the type of that variable can be inferred from the context
- **Multiple parameters :**
  (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Please note: Lambda expressions are just like functions and they accept parameters just like functions.

```
/ A Java program to demonstrate simple lambda expressions
import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
        // Creating an ArrayList with elements
```

```
        // {1, 2, 3, 4}
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);

        // Using lambda expression to print all elements
        // of arrL
        arrL.forEach(n -> System.out.println(n));

        // Using lambda expression to print even elements
        // of arrL
        arrL.forEach(n -> { if (n%2 == 0) System.out.println(n); });
    }
}
```

Note that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements Consumer Functional Interface.
A Java program to demonstrate working of lambda expression with two arguments.

```
// Java program to demonstrate working of lambda expressions
public class Test
{
    // operation is implemented using lambda expressions
    interface FuncInter1
    {
        int operation(int a, int b);
    }

    // sayMessage() is implemented using lambda expressions
    // above
    interface FuncInter2
    {
        void sayMessage(String message);
    }

    // Performs FuncInter1's operation on 'a' and 'b'
    private int operate(int a, int b, FuncInter1 fobj)
    {
        return fobj.operation(a, b);
    }

    public static void main(String args[])
    {
        // lambda expression for addition for two parameters
        // data type for x and y is optional.
        // This expression implements 'FuncInter1' interface
        FuncInter1 add = (int x, int y) -> x + y;

        // lambda expression multiplication for two parameters
        // This expression also implements 'FuncInter1' interface
```



```
        FuncInter1 multiply = (int x, int y) -> x * y;

        // Creating an object of Test to call operate using
        // different implementations using lambda Expressions
        Test tobj = new Test();

        // Add two numbers using lambda expression
        System.out.println("Addition is " +
                           tobj.operate(6, 3, add));

        // Multiply two numbers using lambda expression
        System.out.println("Multiplication is " +
                           tobj.operate(6, 3, multiply));

        // lambda expression for single parameter
        // This expression implements 'FuncInter2' interface
        FuncInter2 fobj = message ->System.out.println("Hello "
                                  + message);
        fobj.sayMessage("Geek");
    }
}
```

Important points:

- The body of a lambda expression can contain zero, one or more statements.
- When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
- When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Block Lambda Expressions

Lambda expression is an unnamed method that is not executed on its own. These expressions cause anonymous class. These lambda expressions are called closures. Lambda's body consists of a block of code. If it has only a single expression they are called "Expression Bodies". Lambdas which contain expression bodies are known as "Expression Lambdas". Below is an example of Lambda expression in a single line.
Block Lambda contains many operations that work on lambda expressions as it allows the lambda body to have many statements. This includes variables, loops, conditional statements like if, else and switch statements, nested blocks, etc. This is created by enclosing the block of statements in lambda body within braces {}. This can even have a return statement i.e return value.

Syntax: Lambda Expressions
(parameters) -> { lambda body }

Now first let us do understand the Lambda expression to get to know about the block lambda expression.

e.g.

In this case, lambda may need more than a single expression in its lambda body. Java supports Expression Lambdas which contains blocks of code with more than one statement. We call this type of Lambda Body a "***Block Body***". Lambdas that contain block bodies can be known as "***Block Lambdas***".

```
// Java Program to illustrate Lambda expression

// Importing input output classes
import java.io.*;

// Interface
// If1 is name of this interface
interface If1 {

    // Member function of this interface
    // Abstract function
    boolean fun(int n);
}

// Class
class Demo {

    // Main driver method
    public static void main(String[] args)
    {
        // Lambda expression body
        If1 isEven = (n) -> (n % 2) == 0;

        // Above is lambda expression which tests
        // passed number is even or odd

        // Condition check over some number N
        // by calling the above function
        // using isEven() over fun() defined above

        // Input is passed as a parameter N
        // Say custom input N = 21
        if (isEven.fun(21))

            // Display message to be printed
            System.out.println("21 is even");
        else

            // Display message to be printed
            System.out.println("21 is odd");
    }
}
```

----------Example-2----------

```
// Java Program to illustrate Block Lambda expression

// Importing all classes from
// java.util package
import java.io.*;

// Block lambda to find out factorial
// of a number

// Interface
interface Func {
    // n is some natural number whose
    // factorial is to be computed
    int fact(int n);
}
```



```
// Class
// Main class
class Factorial {
    // Main driver method
    public static void main(String[] args)
    {
        // Block lambda expression
        Func f = (n) ->
        {
            // Block body

            // Initially initializing with 1
            int res = 1;

            // iterating from 1 to the current number
            // to find factorial by multiplication
            for (int i = 1; i <= n; i++)
                res = i * res;
            return res;
        };

        // Calling lambda function

        // Print and display n the console
        System.out.println("Factorial of 5 : " + f.fact(5));
    }
}
```

Here in this block lambda declares a variable 'res', for loop and has return statement which are legal in lambda body.

----------Example-3----------

```
// Java Program to illustrate Block Lambda expression

// Importing all input output classes
import java.io.*;

// Interface
// Functional interface named 'New'
interface New {

    // Boolean function to check over
    // natural number depicting calendar year

    // 'n' deepicting year is
    // passed as an parameter
    boolean test(int n);
}

// Class
// Main class
class Demo {

    // Main driver method
    public static void main(String[] args)
    {
        // block lambda
        // This block lambda checks if the
        // given year is leap year or not
        New leapyr = (year) ->
        {
            // Condition check
```

```
            // If year is divisible by 400 or the
            // year is divisible by 4 and 100 both
            if (((year % 400 == 0)
                 || (year % 4 == 0) && (year % 100 != 0)))

                // Returning true as year is leap year
                return true;
            else

                // Returning false for non-leap years
                return false;
        };

        // Calling lambda function over
        // custom input year- 2020

        // Condition check using the test()
        // defined in the above interface
        if (leapyr.test(2020))

            // Display message on the console
            System.out.println("leap year");
        else

            // Display message on the console
            System.out.println("Non leap year");
    }
}
```

Here in this block lambda has if-else conditions and return statements which are legal in lambda body.

Java Lambda Expression with Collections

Collections with Comparator (or without Lambda): We can use Comparator interface to sort, It only contains one abstract method: – compare(). An interface that only contains only a single abstract method then it is called a Functional Interface.

- Use of Comparator(I): –
- Prototype of compare() method: –

While defining our own sorting, JVM is always going to call Comparator to compare() method.

- returns negative value(-1), if and only if obj1 has to come before obj2.
- returns positive value(+1), if and only if obj1 has to come after obj2.
- returns zero(0), if and only if obj1 and obj2 are equal.

In List, Set, Map, or anywhere else when we want to define our own sorting method, JVM will always call compare() method internally. **When there is Functional Interface concept used, then we can use Lambda Expression in its place. Sorting elements of List(I) with Lambda Expression:** – Using lambda expression in place of comparator object for defining our own sorting in collections.

----------Example----------

```
import java.util.*;

public class Demo {
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
```



```
        al.add(205);
        al.add(102);
        al.add(98);
        al.add(275);
        al.add(203);
        System.out.println("Elements of the ArrayList " +
                           "before sorting : " + al);

        // using lambda expression in place of comparator object
        Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 :
                                         (o1 < o2) ? 1 : 0);

        System.out.println("Elements of the ArrayList after" +
                                          " sorting : " + al);
    }
}
```

Sorting TreeSet using Lambda Expression:

----------Example----------

```
import java.util.*;

public class Demo {
    public static void main(String[] args)
    {
        TreeSet<Integer> h =
                    new TreeSet<Integer>((o1, o2) -> (o1 > o2) ?
                                          -1 : (o1 < o2) ? 1 : 0);
        h.add(850);
        h.add(235);
        h.add(1080);
        h.add(15);
        h.add(5);
        System.out.println("Elements of the TreeSet after" +
                                          " sorting are: " + h);
    }
}
```

Sorting elements of TreeMap using Lambda Expression: Sorting will be done on the basis of the keys and not its value.

```
import java.util.*;

public class Demo {
    public static void main(String[] args)
    {
        TreeMap<Integer, String> m =
                new TreeMap<Integer, String>((o1, o2) -> (o1 > o2) ?
                                                  -1 : (o1 < o2) ? 1 : 0);
        m.put(1, "Apple");
        m.put(4, "Mango");
        m.put(5, "Orange");
        m.put(2, "Banana");
        m.put(3, "Grapes");
        System.out.println("Elements of the TreeMap " +
                           "after sorting are : " + m);
    }
}
```

It is also possible to specify a reverse comparator via a lambda expression directly in the call to the TreeSet() constructor, as shown here:

----------Example----------

```
// Use a lambda expression to create a reverse comparator
import java.util.*;
```

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

  // Pass a reverse comparator to TreeSet() via a lambda expression
  TreeSet<String> ts=new TreeSet<String>((aStr,bStr) -> bStr.compareTo(aStr));

  // Add elements to the Treeset
  ts.add("A");
  ts.add("B");
  ts.add("C");
  ts.add("D");
  ts.add("E");
  ts.add("F");
  ts.add("G");

  //Display the elements .
  for(String element : ts)
    System.out.println(element + "");

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

Lambda Expression Variable Capturing with Examples

Variable defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or static variable defined by its enclosing class. A lambda expression also has access to (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an intrinsic or static variable and call a method define by its enclosing class.

Lambda expression in java Using a local variable is as stated.
However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a variable capture. In this case, a lambda expression may only use local variables that are effectively final. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as final, although doing so would not be an error.
It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

There are certain keypoints to be remembered, which are as follows:

1. Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be effectively final , or a compile-time error occurs where the use is attempted.
2. Any local variable used but not declared in a lambda body must be definitely assigned before the lambda body, or a compile-time error occurs.
3. Similar rules on variable use apply in the body of an inner class . The restriction to effectively final variables prohibits access to dynamically-changing local variables, whose capture would likely introduce concurrency problems. Compared to the final restriction, it reduces the clerical burden on programmers.
4. The restriction to effectively final variables includes standard loop variables, but not enhanced-for loop variables, which are treated as distinct for each iteration of the loop.

The following program illustrates the difference between effectively final and mutable local

variables:

----------Example----------

```
// Java Program Illustrating Difference between
// Effectively final and Mutable Local Variables

// Importing reqiored classes
import java.io.*;
// An example of capturing a local variable from the
// enclosing scope

// Inrterface
interface MyFunction {

    // Method inside the interface
    int func(int n);
}

// Main class
class Demo{

    // Main driver method
    public static void main(String[] args)
    {

        // Custom local variable that can be captured
        int number = 10;

        MyFunction myLambda = (n) ->
        {

            // This use of number is OK It does not modify
            // num
            int value = number + n;

            // However, the following is illegal because it
            // attempts to modify the value of number

            // number++;
            return value;
        };

        // The following line would also cause an error,
        // because it would remove the effectively final
        // status from num. number = 9;

        System.out.println("IACSD");
    }
}
```

Output explanation:

As the comments indicate, number is effectively final and can, therefore, be used inside myLambda. However, if number were to be modified, either inside the lambda or outside of it, number would lose its effective final status. This would cause an error, and the program would not compile.

----------Example----------

```
// Java Program Illustrating Difference between
// Effectively final and Mutable Local Variables

// Importing input output classes
import java.io.*;

// Interface
interface MyInterface {

    // Method inside the interface
    void myFunction();
}

// Main class
class Demo {

    // Custom initialization
    int data = 170;

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of this class
        // inside the main() method
        Demo d = new Demo();

        // Creating object of interface
        // inside the main() method
        MyInterface intFace = () ->
        {
            System.out.println("Data : " + d.data);
            gfg.data += 500;

            System.out.println("Data : " + d.data);
        };

        intFace.myFunction();
        gfg.data += 200;

        System.out.println("Data : " + d.data);
    }
}
```

Note: *It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.*

How to Create Thread using Lambda Expressions in Java?

Lambda Expressions are introduced in Java SE8. These expressions are developed for **Functional Interfaces**. A functional interface is an interface with only one abstract method.

Syntax:

(argument1, argument2, .. argument n) -> {

// statements

};

Here we make use of the **Runnable Interface**. As it is a Functional Interface, Lambda expressions can be used. The following steps are performed to achieve the task:

- Create the Runnable interface reference and write the Lambda expression for the run() method.
- Create a Thread class object passing the above-created reference of the Runnable interface since the start() method is defined in the Thread class its object needs to be created.
- Invoke the start() method to run the thread.

----------Example----------

```
public class Test {

    public static void main(String[] args)
    {

        // Creating Lambda expression for run() method in
        // functional interface "Runnable"
        Runnable myThread = () ->
        {

            // Used to set custom name to the current thread
            Thread.currentThread().setName("myThread");
            System.out.println(
                Thread.currentThread().getName()
                + " is running");
        };

        // Instantiating Thread class by passing Runnable
        // reference to Thread constructor
        Thread run = new Thread(myThread);

        // Starting the thread
        run.start();
    }
}
```

----------Example----------

```
public class Test {

    public static void main(String[] args)
    {
        Runnable basic = () ->
        {

            String threadName
                = Thread.currentThread().getName();
            System.out.println("Running common task by "
                               + threadName);
        };

        // Instantiating two thread classes
        Thread thread1 = new Thread(basic);
        Thread thread2 = new Thread(basic);

        // Running two threads for the same task
        thread1.start();
        thread2.start();
    }
}
```

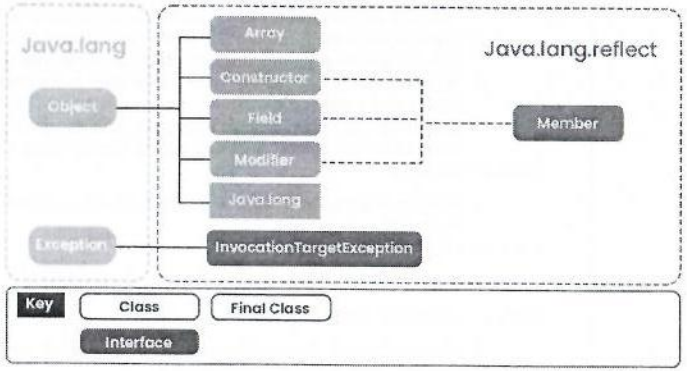
```
import java.util.Random;

// This is a random player class with two functionalities
// playGames and playMusic
class RandomPlayer {

    public void playGame(String gameName)
        throws InterruptedException
    {

        System.out.println(gameName + " game started");

        // Assuming game is being played for 500
        // milliseconds
        Thread.sleep(500); // this statement may throw
                           // interrupted exception, so
                           // throws declaration is added

        System.out.println(gameName + " game ended");
    }

    public void playMusic(String trackName)
        throws InterruptedException
    {

        System.out.println(trackName + " track started");

        // Assuming music is being played for 500
        // milliseconds
        Thread.sleep(500); // this statement may throw
                           // interrupted exception, so
                           // throws declaration is added

        System.out.println(trackName + " track ended");
    }
}

public class Test {

    // games and tracks arrays which are being used for
    // picking random items
    static String[] games
        = { "COD",      "Prince Of Persia", "GTA-V5",
            "Valorant", "FIFA 22",          "Fortnite" };
    static String[] tracks
        = { "Believer", "Cradles", "Taki Taki", "Sorry",
            "Let Me Love You" };

    public static void main(String[] args)
    {

        RandomPlayer player
            = new RandomPlayer(); // Instance of
                                  // RandomPlayer to access
                                  // its functionalities

        // Random class for choosing random items from above
        // arrays
        Random random = new Random();
```



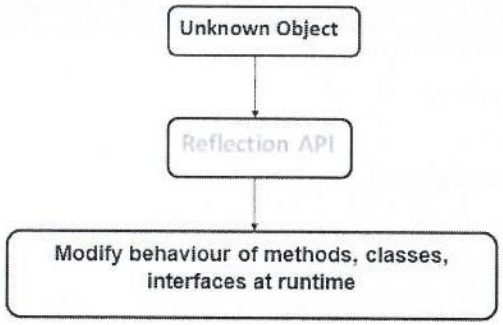
```
        // Creating two lambda expressions for runnable
        // interfaces
        Runnable gameRunner = () ->
        {

            try {

                player.playGame(games[random.nextInt(
                    games.length)]); // Choosing game track
                                     // for playing
            }
            catch (InterruptedException e) {

                e.getMessage();
            }
        };

        Runnable musicPlayer = () ->
        {

            try {

                player.playMusic(tracks[random.nextInt(
                    tracks.length)]); // Choosing random
                                      // music track for
                                      // playing
            }
            catch (InterruptedException e) {

                e.getMessage();
            }
        };

        // Instantiating two thread classes with runnable
        // references
        Thread game = new Thread(gameRunner);
        Thread music = new Thread(musicPlayer);

        // Starting two different threads
        game.start();
        music.start();

        /*
         *Note: As we are dealing with threads output may
         *differ every single time we run the program
         */
    }
```

Reflection in Java

Reflection is an API that is used to examine or modify the behavior of methods, classes, and interfaces at runtime. The required classes for reflection are provided under **java.lang.reflect** package which is essential in order to understand reflection. So we are illustrating the package with visual aids to have a better understanding as follows:



- Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.
- Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.



Reflection can be used to get information about class, constructors, and methods as depicted below in tabular format as shown:

Class	The getClass() method is used to get the name of the class to which an object belongs.
Constructors	The getConstructors() method is used to get the public constructors of the class to which an object belongs.
Methods	The getMethods() method is used to get the public methods of the class to which an object belongs.

We can invoke a method through reflection if we know its name and parameter types. We use two methods for this purpose as described below before moving ahead as follows:

1. getDeclaredMethod()
2. invoke()

Method 1: getDeclaredMethod(): It creates an object of the method to be invoked.
Syntax: The syntax for this method
Class.getDeclaredMethod(name, parametertype)

Parameters:

- Name of a method whose object is to be created
- An array of Class objects

Method 2: invoke(): It invokes a method of the class at runtime we use the following method.
Syntax:
Method.invoke(Object, parameter)

Tip: *If the method of the class doesn't accept any parameter then null is passed as an argument.*
Note: *Through reflection, we can access the private variables and methods of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.*
Method 3: Class.getDeclaredField(FieldName): Used to get the private field. Returns an object of type Field for the specified field name.

Method 4: Field.setAccessible(true): Allows to access the field irrespective of the access modifier used with the field.

Important observations Drawn From Reflection API

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members of classes.
- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code that are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Java.lang.Class class in Java

Java provides a class with name **Class** in java.lang package. Instances of the class Class represent classes and interfaces in a running Java application. The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword *void* are also represented as Class objects. It has no public constructor. Class objects are constructed automatically by the Java Virtual Machine(JVM). It is a final class, so we cannot extend it. The Class class methods are widely used in Reflection API.
In Java, the java.lang.Class class is a built-in class that represents a class or interface at runtime. It contains various methods that provide information about the class or interface, such as its name, superclass, interfaces, fields, and methods.

Here are some commonly used methods of the Class class:

1. getName(): Returns the name of the class or interface represented by this Class object.
2. getSimpleName(): Returns the simple name of the class or interface represented by this Class object.
3. getSuperclass(): Returns the superclass of the class represented by this Class object.
4. getInterfaces(): Returns an array of interfaces implemented by the class or interface represented by this Class object.
5. getField(String name): Returns a Field object that represents the public field with the specified name in the class or interface represented by this Class object.
6. getMethod(String name, Class<?>... parameterTypes): Returns a Method object that represents the public method with the specified name and parameter types in the class or interface represented by this Class object.
7. newInstance(): Creates a new instance of the class represented by this Class object using its default constructor.
8. isInstance(Object obj): Returns true if the specified object is an instance of the class or interface represented by this Class object, false otherwise.
9. isAssignableFrom(Class<?> cls): Returns true if this Class object is assignable from the specified Class object, false otherwise.

```
public class Main {
   public static void main(String[] args) throws Exception {
      Class<?> myClass = String.class;

      System.out.println("Name: " + myClass.getName());
      System.out.println("Simple name: " + myClass.getSimpleName());
      System.out.println("Superclass: " + myClass.getSuperclass());
      System.out.println("Interfaces: " + Arrays.toString(myClass.getInterfaces()));
      System.out.println("Is string assignable from Object? " +
String.class.isAssignableFrom(Object.class));
   }
}
```


Creating a Class object

There are **three** ways to create Class object :

1. **Class.forName("className") :** Since class Class doesn't contain any constructor, there is static **factory** method present in class Class, which is *Class.forName()* , used for creating object of class Class. Below is the syntax :

```
Class c = Class.forName(String className)
```

1. The above statement creates the Class object for the class passed as a String argument(className). Note that the parameter className must be fully qualified name of the desired class for which Class object is to be created. The methods in any class in java which returns the same class object are also known as factory methods. The class name for which Class object is to be created is determined at run-time.
2. **Myclass.class :** When we write .class after a class name, it references the Class object that represents the given class. It is mostly used with primitive data types and only when we know the name of class. The class name for which Class object is to be created is determined at compile-time. Below is the syntax :

```
Class c = int.class
```

1. Please note that this method is used with class name, not with class instances. For example

```
A a = new A();  // Any class A

Class c = A.class; // No error

Class c = a.class; // Error
```

1. **obj.getClass():** This method is present in Object class. It return the run-time class of this(obj) object. Below is the syntax :

```
A a = new A();  // Any class A

Class c = a.getClass();
```

Methods:

1. **String toString()** : This method converts the Class object to a string. It returns the string representation which is the string "class" or "interface", followed by a space, and then by the fully qualified name of the class. If the Class object represents a primitive type, then this method returns the name of the primitive type and if it represents *void* then it returns "void".

Syntax :
public String toString()
Parameters :
NA
Returns :
a string representation of this class object.
Overrides :
toString in class Object

```
/ Java program to demonstrate toString() method
public class Test {
   public static void main(String[] args)
      throws ClassNotFoundException
   {
      // returns the Class object for the class
      // with the specified name
      Class c1 = Class.forName("
```

```
                    java.lang.String &amp;quot;);
        Class c2 = int.class;
        Class c3 = void.class;

        System.out.print(" Class represented by c1
                    : ");

        // toString method on c1
        System.out.println(c1.toString());

        System.out.print(" Class represented by c2
                    : ");

        // toString method on c2
        System.out.println(c2.toString());

        System.out.print(" Class represented by c3
                    : ");

        // toString method on c3
        System.out.println(c3.toString());
    }
}
```

1. **Class<?> forName(String className)** : As discussed earlier, this method returns the Class object associated with the class or interface with the given string name. The other variant of this method is discussed next.

```
// Java program to demonstrate forName() method
public class Test
{
    public static void main(String[] args)
                throws ClassNotFoundException
    {
        // forName method
        // it returns the Class object for the class
        // with the specified name
        Class c = Class.forName("java.lang.String");

        System.out.print("Class represented by c : " + c.toString());
    }
}
```

Class<?> forName(String className,boolean initialize, ClassLoader loader) : **This method also returns the Class object associated with the class or interface with the given string name using the given class loader. The specified class loader is used to load the class or interface. If the parameter loader is null, the class is loaded through the bootstrap class loader in. The class is initialized only if the initialize parameter is true and if it has not been initialized earlier.**

```
// Java program to demonstrate forName() method
public class Test
```



```
{
    public static void main(String[] args)
                throws ClassNotFoundException
    {
        // returns the Class object for this class
        Class myClass = Class.forName("Test");

        ClassLoader loader = myClass.getClassLoader();

        // forName method
        // it returns the Class object for the class
        // with the specified name using the given class loader
        Class c = Class.forName("java.lang.String",true,loader);

        System.out.print("Class represented by c : " + c.toString());
    }
}
```

T newInstance() : This method creates a new instance of the class represented by this Class object. The class is created as if by a *new* expression with an empty argument list. The class is initialized if it has not already been initialized.

```
Class myClass = Class.forName("Test");

    // creating new instance of this class
    // newInstance method
    Object obj = myClass.newInstance();
```

boolean isInstance(Object obj) : This method determines if the specified Object is assignment-compatible with the object represented by this Class. It is equivalent to instanceof operator in java.

```
    Class c = Class.forName("java.lang.String");

String s = "IACSD";
    int i = 10;


boolean b1 = c.isInstance(s);
    boolean b2 = c.isInstance(i);

    System.out.println("is s instance of String : " + b1);
    System.out.println("is i instance of String : " + b1);
```

Java.lang.Class class in Java

1. int getModifiers() : This method returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the Java Virtual Machine's constants for public, protected, private, final, static, abstract and interface. These modifiers are already decoded in Modifier class in java.lang.Reflect package.

Syntax :
public int getModifiers()
Parameters :
NA
Returns :
the int representing the modifiers for this class

```
import java.lang.reflect.Modifier;

public abstract class Test
{
    public static void main(String[] args)
    {
        // returns the Class object associated with Test class
        Class c = Test.class;

        // returns the Modifiers of the class Test
        // getModifiers method
        int i = c.getModifiers();

        System.out.println(i);

        System.out.print("Modifiers of " + c.getName() + " class are : ");

        // getting decoded i using toString() method
        // of Modifier class
        System.out.println(Modifier.toString(i));
    }
}
```

2. T[] getEnumConstants() : This method returns the elements of this enum class. It returns null if this Class object does not represent an enum type.
Syntax :
public T[] getEnumConstants()
Parameters :
NA
Returns :
an array containing the values comprising the enum class represented by this Class object in the order they're declared,
or null if this Class object does not represent an enum type

```
// Java program to demonstrate getEnumConstants() method

enum Color
{
    RED, GREEN, BLUE;
}

public class Test
{
    public static void main(String[] args)
    {
        // returns the Class object associated with Color(an enum class)
        Class c1 = Color.class;

        // returns the Class object associated with Test class
        Class c2 = Test.class;
```



```
        // returns the elements of Color enum class in an array
        // getEnumConstants method
        Object[] obj1 = c1.getEnumConstants();

        System.out.println("Enum constants of " + c1.getName() + " class are :");

        // iterating through enum constants
        for (Object object : obj1)
        {
            System.out.println(object);
        }

        // returns null as Test Class object does not represent an enum type
        Object[] obj2 = c2.getEnumConstants();

        System.out.println("Test class does not contain any Enum constant.");
        System.out.println(obj2);

    }
}
```

4. boolean desiredAssertionStatus() : This method returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked.
Syntax :
public boolean desiredAssertionStatus()
Parameters :
NA
Returns :
the desired assertion status of the specified class.

5. Class<?> getComponentType() : This method returns the Class representing the component type of an array. If this class does not represent an array class this method returns null.
Syntax :
public Class<?> getComponentType()
Parameters :
NA
Returns :
the Class representing the component type of this class if this class is an array

6. Class<?>[] getDeclaredClasses() : Returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object. This method includes public, protected, default (package) access, and private classes and interfaces declared by the class, but excludes inherited classes and interfaces. This method returns an array of length 0 if the class declares no classes or interfaces as members, or if this Class object represents a primitive type, an array class, or void.
Syntax :
public Class<?>[] getDeclaredClasses()
Parameters :
NA
Returns :
the array of Class objects representing all the declared members of this class
Throws :
SecurityException - If a security manager, s, is present

7. Field getDeclaredField(String fieldName) : This method returns a Field object that reflects the specified declared field of the class or interface represented by this Class object. The name parameter is a String that specifies the simple name of the desired field. Note that this method will not reflect the length field of an array class.

Syntax :
public Field getDeclaredField(String fieldName)
throws NoSuchFieldException,SecurityException
Parameters :
fieldName - the field name
Returns :
the Field object for the specified field in this class
Throws :
NoSuchFieldException - if a field with the specified name is not found.
NullPointerException - if fieldName is null
SecurityException - If a security manager, s, is present.

8. Field[] getDeclaredFields() : This method returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object. This includes public, protected, default (package) access, and private fields, but excludes inherited fields. This method returns an array of length 0 if the class or interface declares no fields, or if this Class object represents a primitive type, an array class, or void.

Syntax :
public Field[] getDeclaredFields() throws SecurityException
Parameters :
NA
Returns :
the array of Field objects representing all the declared fields of this class
Throws :
SecurityException - If a security manager, s, is present.

****