Interview Questions and answers (In progress)

https://docs.google.com/document/d/1mx_uKkQeubJd57DWUcAt4ozc4zOq-B_6aAz0YFzoMnQ/edit#heading=h.xthd4iya0y9t

Core Java Index

Java is Simple

- **Java is very easy to learn, and its syntax is simple, clean and easy to understand.**
- **Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.**
- **There is no need to remove unreferenced objects because there is an Automatic Garbage Collec Java.**

Java Introduction

Small history of java !!

- **Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991.**
- **The first publicly available version of Java (Java 1.0) was released in 1995.**
- **Sun Microsystems was acquired by the Oracle Corporation in 2010.**
- **Oracle now has the steermanship for Java.**

Why Java ?

- **Java Programming language is invented for developing internet applications by providing platform(Operating System Ex: Windows) independence (Write once run anywhere → WORA).**

Definition of Java !!

- **Java is a Simple, High level, Secure, Platform independent, ,Robust, Dynamic, Multithreading and Object oriented programming language used for developing both standalone and web(internet) applications.**

Java is Platform Independent

- **Platform independent language means once compiled you can execute the program on any platform (OS).**
- **Java is platform independent, because the Java compiler converts the source code to bytecode, which is Intermediate Language.**
- **Bytecode can be executed on any platform (OS) using JVM(Java Virtual Machine).**

Java is Object-oriented

- **Java is an object oriented programming language.**
- **Everything in Java is an object.**
- **Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.**

Java is Robust

- **Robust means strong.**
- **Java is robust because it uses strong memory management.**
- **There is a lack of pointers that avoids security problems.**
- **There is automatic garbage collection in Java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.**
- **There are exception handling and the type checking mechanism in Java.**

Java is Multi-threaded

- **A thread is like a separate program, executing concurrently.**
- **We can write Java programs that deal with many tasks at once by defining multiple threads.**
- **The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.**
- **Threads are important for multimedia, Web applications, etc.**

Java is Dynamic

- **Java is a dynamic language.**
- **It supports dynamic loading of classes.**
- **It means classes are loaded on demand.**
- **It also supports functions from its native languages, i.e., C and C++.**
- **Java supports dynamic compilation and automatic memory management (garbage collection).**

Java is Secured

- **Java is best known for its security.**
- **With Java, we can develop virus-free systems.**
- **Java is secured because, No explicit pointer**
- **Java Programs run inside a virtual machine sandbox.**
- **Java is secure with access modifiers and Encapsulation mechanisms too.**

What is Classloader

- **Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically.**
- **It adds security by separating the package for the classes of the local file system from those that are imported from network sources.**
- **Bytecode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.**

Security Manager

- **It determines what resources a class can access such as reading and writing to the local disk.**
- **Java provides these securities by default.**
- **Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.**

Terminologies !!

Platform :

- **The Operating System and the architecture of the processor together is known as a platform.**

Source Code :

- **The group of statements present in the high level format which looks like English is known as source code.**

Compiled code :

- **Any code that we get after the compilation could be called the compiled code.**

Executable code :

- **The group of statements which are present in the low level format (machine language) which looks like a stream of the 0's and 1's known as executable code.**
- **The format of the executable code changes from OS to OS thus executable code is always specific to the OS.**

Format :

- **The way we represent the data or the procedure we want to follow for representing the data is known as format.**

Bytecode :

- **The group of statements representing the high level java program in the middle level using byte format is known as bytecode.**
- **Byte is the name of the format which is used by the java compiler for representing high level java statements in the middle level format.**
- **Since the bytecode is not the executable code it would not be specific to any OS, thus ByteCode is always Platform independent.**

What is a Java Interpreter ?

- **"Interpreted" means that the computer looks at the language, and then turns it into native machine language.**
- **The term Java interpreter refers to a program which implements the JVM specification and actually executes the bytecodes by running your program.**

Compiler VS Interpreter

- **Compiling happens when the writing of the program is finished, like calling javac.**
- **Example: Through command line javac HelloWorld.java**
- **Compilation of java file will generate class file like (ex: HelloWorld.class)**
- **Interpretation happens at runtime(which means while running the Java program), Example: Through command line**

Static loading :- vs Dynamic loading

- **The concept of allocating the memory and loading the functionalities to the RAM before the program is under execution (or) before a function call is made is known as static loading.**

- **Structured programming like C would be working based on the concept of static loading.**
- **Static loading increases the overhead in the system and complexity.**

 # include<stdio.h>

- **The concept of allocating the memory and loading the functionalities to the RAM dynamically at runtime when the program is under execution.**

- **i.e when a method call is made is known as Dynamic loading.**
- **Object Oriented programming languages like java would be working based on the concept of Dynamic loading.**

String s = "srikanth";
s.length()E; // 8

JVM

Why do we need JVM :

- **Operating systems would not provide an environment which is required for the processor to execute a java application.**
- **In order to provide a complete environment which is required for the processor and system to execute a java application we need JVM.**

What is JVM :

- **JVM is a collection of programs, which are required to provide a complete environment which is required by the processor**
  to execute bytecode or Java applications.
- **Jvm controls the operations of the CPU with respect to executing Java applications (or) bytecode.**
- **Anything which performs multiple operations automatically by itself can be called a machine.**
- **Since JVM takes care of performing multiple operations automatically by itself, we call that machine.**
- **Any program which is under execution is always virtual (or) imaginary.**
- **Since JVM is also a program which is under execution we call Java virtual.**

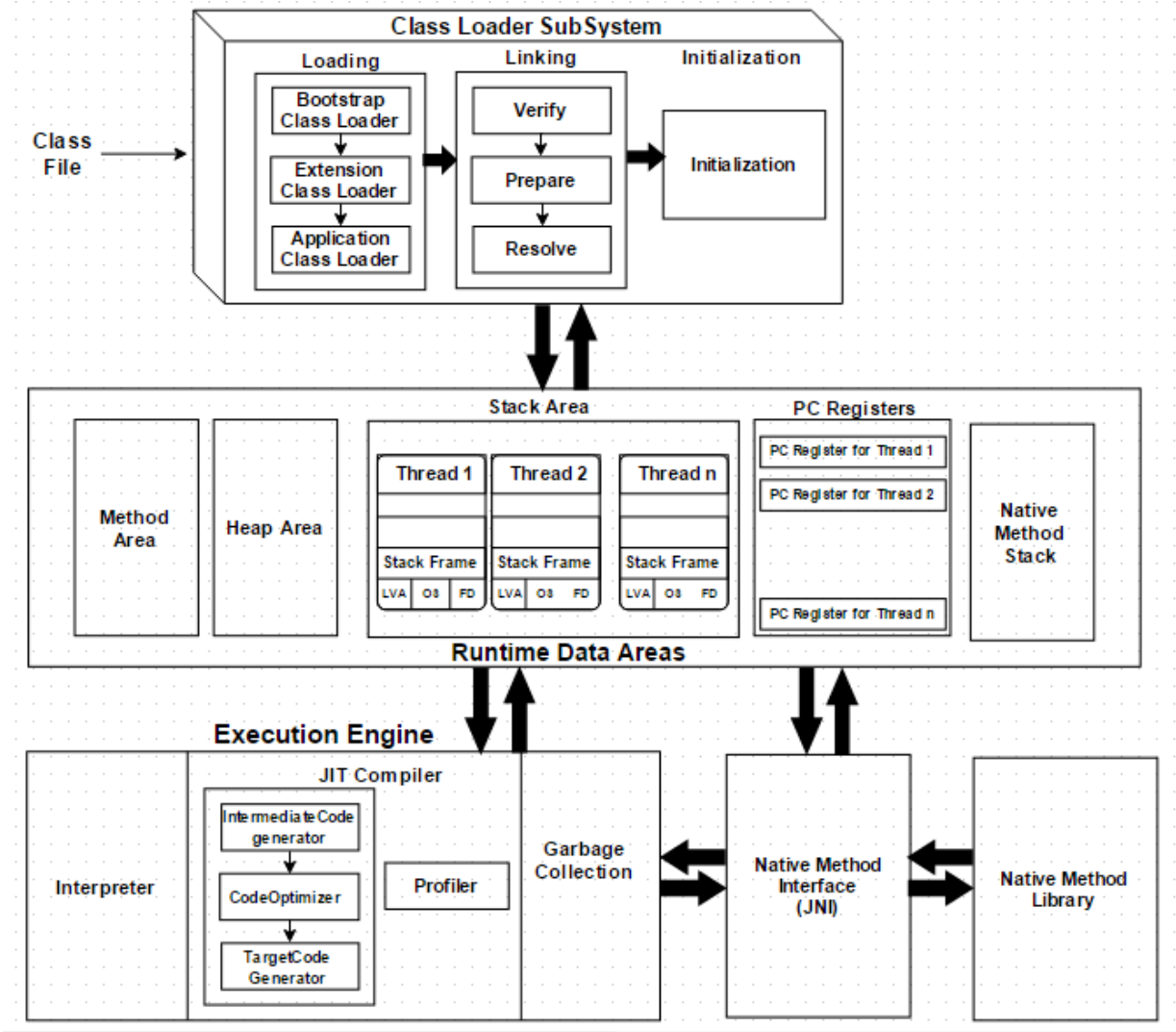
Since JVM is a collection of programs related to the JAVA technologies we call that a Java Virtual machine.

JVM Architecture

- **JVM is responsible for loading and running java applications.**

HelloWorld.java →
Compilation : Javac FirstExample.java →

FirstExample.class
.class →



1. ClassLoader Subsystem

- **Java's dynamic class loading functionality is handled by the ClassLoader subsystem.**
- **It loads, links and initializes the class file when it refers to a class for the first time at runtime, not compile time.**

1.1 Loading

- **Classes will be loaded by this component.**
- **BootStrap ClassLoader,**
- **Extension ClassLoader, and**
- **Application ClassLoader are the three ClassLoaders that will help in achieving it.**

BootStrap ClassLoader :

- **Responsible for loading classes from the bootstrap classpath, nothing but rt.jar.**
- **Highest priority will be given to this loader.**
- **C:\Program Files\Java\jre1.8.0_261\lib**

Extension ClassLoader – Responsible for loading classes which are inside the ext folder (jre\lib).

Application ClassLoader

- **Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.**

- **The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.**

1.2 Linking

Verify – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
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.

1.3 Initialization

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

2. Runtime Data Area or Memory Management

The Runtime Data Area is divided into five 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 shared 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 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 subentities:**

Local Variable Array –

- **Related to the method, how many local variables are involved and the corresponding values will be stored here.**

Operand stack –

- **If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.**

Frame data –

- **All symbols corresponding to the methods are 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 the 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.**

3. Execution Engine

- **The bytecode, which is assigned to the Runtime Data Area, will be executed by the Execution Engine.**
- **The Execution Engine reads the bytecode and executes it piece by piece.**

Interpreter –

- **The interpreter interprets the bytecode 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.
- **Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed.**
- **Garbage collection of the JVM collects the objects that are created.**

Java Native Interface (JNI):
- **JNI will be interacting with the Native Method Libraries and provide 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.**

What is class ?

- **A class is a blueprint of an object.**
- **(blueprint means plan or architecture or overview or prototype)**
- **A class is a user defined data type we can say.**
- **It represents the set of properties or methods that are common to all objects of one type.**

What is an Object ?

- **An instance of a class is called an Object.**
- **Object is a dynamic memory allocation.**
- **Object is a real time entity.**
- **An Object consists of State, Behavior and Identity.**

Note :
Whereas Class is dummy and the Object is real.

Q : A file can contain more than one class ?

Ans : Yes ! But at most only one public class should be there.

Q : FIle name and class name should be the same ?

Ans : No need to be the same, until and unless your class is public.

Q : Differences between JDK, JRE and JVM ?

JDK = Java Runtime Environment (JRE) + Development tools.
JRE = Java Virtual Machine (JVM) + Libraries to run the application.
JVM = Only Runtime environment for executing the Java byte code.

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 components of JRE in order to understand the importance of it and to perceive how it actually works.
For this let us discuss components.
Differences between JDK, JRE and JVM ?

JDK = Java Runtime Environment (JRE) + Development tools.
JRE = Java Virtual Machine (JVM) + Libraries to run the application.
JVM = Only Runtime environment for executing the Java byte code.

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.

Q: Diff between path and classpath?

The main difference between PATH and CLASSPATH is that
Path is set for java tools in java programs like java and javac, which are used to compile your code. (It is for Generic way)
Whereas CLASSPATH is used by System or Application class loaders to locate and load compiled Java bytecodes stored in the . class file. (It is for Specific way)

****End of Java introduction ****

Language fundamentals

1) **Identifiers.**
2) **Reserved words.**

3) Data types.
4) Literals.
5) Arrays.
6) Types of variables.
7) var arg method.
8) main() method.
9) command line arguments.
10) Java coding standards.
11) Constructors.

Identifiers

- A name in a Java program is called identifier, which can be used for identification purposes.
- It can be class name or method name, variable name or label name.
- Ex: class Test(1) {
  public static void main(3) ((2)String[] (4)args){
  int x(5)=0;

}

- Total 5 identifiers in above example.
  Rules for defining identifiers :

  - The only characters allowed in java identifiers are :
    Ex: A → z, A → Z, 0 → 9 and $, _.

  - By mistake if you are trying to use any other character, we will get a compile time error.
  - Ex: Total_Number (Valid)
    Total#(Not valid)

- Identifiers can't start with a digit.
  Ex: 123Total → invalid
  Ex: total123 → valid

- Java identifiers are case sensitive ofcourse, java language itself case- sensitive programming language.
  Ex: class Test{
  int number =10;
  int number1 =20;
  int number2 =30;
  }

Java Language Keywords

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

List of Keywords 👏

Application Level 14	Access Modifiers 12	Data Types 8	Exceptions Handling 5	Logics or control flow 11
package	private	byte	try	if
import	default	short	catch	else
class	protected	int	finally	do
interface	public	long	throws	while
enum	static	float	throw	for
new	strictfp	double		switch
instanceof	native	char		case
void	synchronized	boolean		break
return	abstract			continue
this	final			goto
super	transient			const
extends	volatile			
implements				
assert				

Application level :

package

- **Java package is a group of similar classes and interfaces.**
- **Packages are declared with the package keyword.**

import

- **Used at the beginning of a source file to specify classes or entire Java packages to be referred to later without including their package names in the reference.**
- **Since J2SE 5.0, import statements can import static members of a class.**

class

- **A type that defines the implementation of a particular kind of object.**
- **A class definition defines instance and class fields, methods, and inner classes as well as specifying the interfaces the class implements and the immediate superclass of the class.**
- **If the superclass is not explicitly specified, the superclass is implicitly Object.**
- **The class keyword can also be used in the form Class.**
- **class to get a Class object without needing an instance of that class.**
- **For example, String.class can be used instead of doing new String().getClass().**

interface

- **Used to declare a special type of class that only contains abstract or default methods, constant (static final) fields and static interfaces.**
- **It can later be implemented by classes that declare the interface with the implements keyword.**
- **As multiple inheritance is not allowed in Java, interfaces are used to circumvent it.**
- **An interface can be defined within another interface.**

extends

- **Used in a class declaration to specify the superclass; used in an interface declaration to specify one or more superinterfaces.**
- **Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y.**
- **An interface Z extends one or more interfaces by adding methods. Class X is said to be a subclass of class Y; Interface Z is said to be a subinterface of the interfaces it extends.**
- **Also used to specify an upper bound on a type parameter in Generics.**

implements

- **Included in a class declaration to specify one or more interfaces that are implemented by the current class.**
- **A class inherits the types and abstract methods declared by the interfaces.**

new

- **Used to create an instance of a class or array object. Using keyword for this end is not completely necessary (as exemplified by Scala), though it serves two purposes: it enables the existence of different namespace for methods and class names, it defines statically and locally that a fresh object is indeed created, and of what runtime type it is (arguably introducing dependency into the code).**

instanceof

- **A binary operator that takes an object reference as its first operand and a class or interface as its second operand and produces a boolean result.**
- **The instanceof operator evaluates to true if and only if the runtime type of the object is assignment compatible with the class or interface.**

void

- **The void keyword is used to declare that a method does not return any value.**

this

- **Used to represent an instance of the class in which it appears.**
- **this can be used to access class members and as a reference to the current instance.**
- **This keyword is also used to forward a call from one constructor in a class to another constructor in the same class.**

super

- **Inheritance is basically used to achieve dynamic binding or run-time polymorphism in java. Used to access members of a class inherited by the class in which it appears.**
- **Allows a subclass to access overridden methods and hidden members of its superclass.**
- **The super keyword is also used to forward a call from a constructor to a constructor in the superclass.**
- **Also used to specify a lower bound on a type parameter in Generics.**

return

- **Used to finish the execution of a method.**
- **It can be followed by a value required by the method definition that is returned to the caller.**

Access Modifiers

private :

- **The private keyword is used in the declaration of a method, field, or inner class;**

private members can only be accessed by other members of their own class.

default

- **The default keyword can optionally be used in a switch statement to label a block of statements to be executed if no case matches the specified value;**
- **see switch.[8][9] Alternatively, the default keyword can also be used to declare default values in a Java annotation.**
- **From Java 8 onwards, the default keyword can be used to allow an interface to provide an implementation of a method.**

protected

- **The protected keyword is used in the declaration of a method, field, or inner class;**
- **protected members can only be accessed by members of their own class, that class's subclasses or classes from the same package.**

public

- **The public keyword is used in the declaration of a class, method, or field; public classes, methods, and fields can be accessed by the members of any class.**

static

- **Used to declare a field, method, or inner class as a class field. Classes maintain one copy of class fields regardless of how many instances exist of that class.**
- **static also is used to define a method as a class method. Class methods are bound to the class instead of to a specific instance, and can only operate on class fields. (Classes and interfaces declared as static members of another class or interface are actually top-level classes and are not inner classes.)**

strictfp (added in J2SE 1.2)

- **A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.**

synchronized

- **Used in the declaration of a method or code block to acquire the mutex lock for an object while the current thread executes the code.**
- **For static methods, the object locked is the class's Class.**
- **Guarantees that at most one thread at a time operating on the same object executes that code.**
- **The mutex lock is automatically released when execution exits the synchronized code. Fields, classes and interfaces cannot be declared as synchronized.**

abstract

- **Abstracts are used to implement an abstraction in Java.**
- **A method with no definition must be declared as abstract and the class containing it must be declared as abstract.**
- **Abstract classes cannot be instantiated.**

- **Abstract methods must be implemented in the sub classes.**
- **The abstract keyword cannot be used with variables or constructors.**
- **Note that an abstract class isn't required to have an abstract method at all.**

final

- **Define an entity once that cannot be changed nor derived from later.**
- **More specifically: a final class cannot be subclassed, a final method cannot be overridden, and a final variable can occur at most once as a left-hand expression on an executed command.**
- **All methods in a final class are implicitly final.**

transient

- **Declares that an instance field is not part of the default serialized form of an object.**
- **When an object is serialized, only the values of its non-transient instance fields are included in the default serial representation.**
- **When an object is deserialized, transient fields are initialized only to their default value.**
- **If the default form is not used, e.g. when a serialPersistentFields table is declared in the class hierarchy, all transient keywords are ignored.**

volatile

- **Used in field declarations to specify that the variable is modified asynchronously by concurrently running threads.**
- **Methods, classes and interfaces thus cannot be declared volatile, nor can local variables or parameters.**

Data Types :

byte

- **The byte keyword is used to declare a field that can hold an 8-bit signed two's complement integer.**
- **This keyword is also used to declare that a method returns a value of the primitive type byte.**

short

- **The short keyword is used to declare a field that can hold a 16-bit signed two's complement integer.**
- **This keyword is also used to declare that a method returns a value of the primitive type short.**

int

- **The int keyword is used to declare a variable that can hold a 32-bit signed two's complement integer.**
- **This keyword is also used to declare that a method returns a value of the primitive type int.**

long

- **The long keyword is used to declare a variable that can hold a 64-bit signed two's complement integer.**
- **This keyword is also used to declare that a method returns a value of the primitive type long.**

float

- **The float keyword is used to declare a variable that can hold a 32-bit single precision IEEE 754 floating-point number.**
- **This keyword is also used to declare that a method returns a value of the primitive type float.**

double

- **The double keyword is used to declare a variable that can hold a 64-bit double precision IEEE 754 floating-point number.**
- **This keyword is also used to declare that a method returns a value of the primitive type double.[6][7]**

boolean

- **Defines a boolean variable for the values "true" or "false" only. By default, the value of the boolean primitive type is false.**
- **This keyword is also used to declare that a method returns a value of the primitive type boolean.**

char

- **Defines a character variable capable of holding any character of the java source file's character set.**

Exceptions :

try

- **Defines a block of statements that have exception handling.**
- **If an exception is thrown inside the try block, an optional catch block can handle declared exception types.**
- **Also, an optional finally block can be declared that will be executed when execution exits the try block and catch clauses, regardless of whether an exception is thrown or not.**
- **A try block must have at least one catch clause or a finally block.**

catch

- **Used in conjunction with a try block and an optional finally block.**
- **The statements in the catch block specify what to do if a specific type of exception is thrown by the try block.**

finally

- **Used to define a block of statements for a block defined previously by the try keyword.**
- **The finally block is executed after execution exits the try block and any associated catch clauses regardless of whether an exception was thrown or caught, or execution left method in the middle of the try or catch blocks using the return keyword.**

throw

- **Causes the declared exception instance to be thrown.**
- **This causes execution to continue with the first enclosing exception handler declared by the catch keyword to handle an assignment compatible exception type.**
- **If no such exception handler is found in the current method, then the method returns and the process is repeated in the calling method.**
- **If no exception handler is found in any method call on the stack, then the exception is passed to the thread's uncaught exception handler.**

throws

- **Used in method declarations to specify which exceptions are not handled within the method but rather passed to the next higher level of the program.**
- **All uncaught exceptions in a method that are not instances of RuntimeException must be declared using the throws keyword.**

Logics or control flow :

assert

- **Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort. Optionally enabled by the ClassLoader method.**

break

- **Used to end the execution in the current loop body.**

case

- **A statement in the switch block can be labeled with one or more case or default labels.**
- **The switch statement evaluates its expression, then executes all statements that follow the matching case label; see switch.**

continue

- **Used to resume program execution at the end of the current loop body. If followed by a label, continue resumes execution at the end of the enclosing labeled loop body.**

do

- **The do keyword is used in conjunction with while to create a do-while loop, which executes a block of statements associated with the loop and then tests a boolean expression associated with the while.**
- **If the expression evaluates to true, the block is executed again; this continues until the expression evaluates to false.**

else

- **The else keyword is used in conjunction with if to create an if-else statement, which tests a boolean expression; if the expression evaluates to true, the block of statements associated with the if are evaluated; if it evaluates to false, the block of statements associated with the else are evaluated.**

enum (added in J2SE 5.0)

- **A Java keyword used to declare an enumerated type. Enumerations extend the base class Enum.**

for

- **The for keyword is used to create a for loop, which specifies a variable initialization, a boolean expression, and an incrementation.**
- **The variable initialization is performed first, and then the boolean expression is evaluated.**
- **If the expression evaluates to true, the block of statements associated with the loop are executed, and then the incrementation is performed.**
- **The boolean expression is then evaluated again; this continues until the expression evaluates to false.**
- **As of J2SE 5.0, the for keyword can also be used to create a so-called "enhanced for loop",[16] which specifies an array or Iterable object; each iteration of the loop executes the associated block of statements using a different element in the array or Iterable.**

if

- **The if keyword is used to create an if statement, which tests a boolean expression;**
- **if the expression evaluates to true, the block of statements associated with the if statement is executed.**
- **This keyword can also be used to create an if-else statement; see else.**

native

- **Used in method declarations to specify that the method is not implemented in the same Java source file, but rather in another language.**

switch

- **The switch keyword is used in conjunction with case and default to create a switch statement, which evaluates a variable, matches its value to a specific case, and executes the block of statements associated with that case.**
- **If no case matches the value, the optional block labelled by default is executed, if included.**

while

- **The while keyword is used to create a while loop, which tests a boolean expression and executes the block of statements associated with the loop if the expression evaluates to true; this continues until the expression evaluates to false.**
- **This keyword can also be used to create a do-while loop; see do.**

Data Types :

- **In java every variable should have some type and every expression should have type.**
- **Each and every assignment is checked by the compiler for type compatibility and every type is clearly defined.**
- **Hence java lang|| is considered as a strongly typed programming language.**

Primitive Data Types	Default Values	Size	Min value	Max Value	Wrapper Object Data Types
byte	0	1 byte (8 bits)	-128	127	Byte
short	0	2 byte(16 bits)	- 32768	32767	Short
int	0	4 byte(32 bits)	- 2147483648	2147483647	Integer
long	0	8 byte(64 bits)	-9,223,372,036,854,775,808	9,223,372,036,854,775,807.	Long
float	0.0	4 byte(32 bits)(5-6 decimal)	-3.4e38 (3.4*10^(38))	3.4e^38	Float
double	0.0	8 byte(64 bits)(14-15 decimal)	-1.7e308	1.7e^308	Double
boolean	false	1 bit	true	false	Boolean
char		2 byte	0	65535	Character

Literals

- **Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables.**
- **Java language specifies five major types of literals.**
- **Literals can be any number, text, or other information that represents a value.**
- **This means what you type is what you get.**
- **We will use literals in addition to variables in Java statements.**
- **While writing a source code as a character sequence, we can specify any value as a literal such as an integer.**
- **Any constant value which can be assigned to the variable is called literal.**

Ex: int x=10; → Constant value /literal

Integral literals

- **For integral data types (byte, short, int, long) we can specify literal values in the following ways.**

Decimal literals (base 10) :

- **Allowed digits are '0 to 9'.**

Ex: int x=9999;

Octal Literals (base 8):

- **Allowed digits are 0 to 7, literal value should be prefixed with zero.**

Ex: int x=060;

HexaDecimal Literals (base 16)

- **Allowed digits are 0 to 9, a to f /A to F. Literal value should be prefixed with 0x or 0X**
  Ex: int x= 0 X10;
- **For extra digits a to f we can use both lowercase and uppercase.**
- **This is one of the very few places where java is not case sensitive.**
- **Literal value should be prefixed with 0x or 0X**
  Ex: int x= 0X10;

Which of the following declarations are valid.

```
int x=10;   // yes
int x=0786; //No
int x=786;  //Yes
int x=0777; //Yes
int x=0XFace; //yes
int x=0xBeef;//Yes
int x=0xBeer //No

public class LiteralsTest1 {
public static void main(String args[]){
int x=10;
int x1=010;
int x2=0x10; //
int x3=0123;   //   0×8³+1×8²+2×8¹+3×8⁰ = 83
System.out.println("Literals " + x +"  " + x3 + "  " +x2);

}
}
```

- **By default every integral literal is of int type, but we can specify explicitly as long type by suffixing with l or L .**

Ex:

```
int x=10; //valid
long l=10L; //valid
long l=10; //valid
int x= 10L; // In valid
```

- **There is no way to specify byte and short literal explicitly.**
- **Whenever we are assigning integral literals to the byte variable and if the value is within the range of byte, the compiler treats automatically as byte literal, similarly short literal also.**

```
byte b=10;
byte b=127;
byte b=128;  //invalid
short = 32767;
short = 32768; //invalid
```

Floating Point literals

- **By default every floating point literal is of double type and hence we cannot assign directly to the float variable but we can specify explicitly floating point literal as float type by suffixing with f or F.**

Ex:
float f= 2134.123; // invalid
double d= 2134.123; // valid
float f= 123456.789F;//valid

- **We can specify explicitly floating point literal as double type by suffixing with d or D. Of course this convention is not required.**

Ex: double d = 123456.789D
float f= 123456.789D

- **We can specify floating point literal only in decimal form.**
- **octal and Hexadecimal forms are not allowed .**

Ex:
double d = 123.456;
double d = 0123.456;
double d = 0x123.456;

- **We can assign an integral literal directly to floating point variables and that integral literal can be specified either in decimal or Octal or HexaDecimal form.**

double d = 10;
double d=010;
double d=0x10;
which one is valid :
double d=0786; // invalid
double d = 0786.0; // valid
double d = 0777;// valid
double d=0xface;// valid

- **So, we can assign integral literals to floating point literals.**
- **But we cannot assign floating point literal to the integral types.**

Ex: int x=10.5;

- **We can assign floating point literals in exponential form also.**

Ex:
double d= 1.2e3; // 1.2* 10 * 10 * 10 =1200
float f=1.2e3; // invalid
float f = 1.2e3F;

Character literals

- **char data type is a single 16-bit Unicode character.**
- **We can specify a character literal as a single printable character in a pair of single quote characters such as**

'a', '#' and '3'.

- **You must know about the ASCII character set.**
- **The ASCII character set includes 128 characters including letters, numerals, punctuation etc.**
- **The table below shows a set of these special characters.**

char c =’a’; //valid
Char c =”b”;// invalid
char c =’ab’;// invalid
Ex: char c= ;

- **The character literal can be specified either in decimal or octal or hexadecimal form but allowed range is**

0 to 65535.
char ch =0777;
char ch =oxface;
Char ch =oxbeef;

- **Which is represent a char literal in unicode representation which is nothing but ‘\uxxxx’;**

- **Every escape character acts as char literal;**

Ex:

char ch =’\n’;
char ch =’\t’;
Char ch =’\r’;

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace

\r	**Carriage return**
\f	**Form feed**
****	**Backslash**
\'	**Single quotation mark**
\"	**Double quotation mark**
\d	**Octal**
\xd	**Hexadecimal**
\ud	**Unicode character**

- **If we want to specify a single quote, backslash, or a non-printable character as a character literals use an escape sequence.**
- **An escape sequence uses a special syntax to represent a character.**
- **The syntax begins with a single backslash character. You can see the below table to view the character literals using the Unicode escape sequence to represent printable and non-printable characters.**

'u0041'	**Capital letter A**
'\u0030'	**Digit 0**
'\u0022'	**Double quote "**
'\u003b'	**Punctuation ;**
'\u0020'	**Space**
'\u0009'	**Horizontal Tab**

String Literals

- **The set of characters is represented as String literals in Java.**
- **Always use "double quotes" for String literals.**
- **There are few methods provided in Java to combine strings, modify strings and to know whether two strings have the same values.**

" "	**The empty string**
" \" "	**A string containing double quote**
"This is a string"	**A string containing 16 characters**
"This is a " + "two-line string"	**actually a string-valued constant expression, formed from two string literals**

Null Literals

- **The final literal that we can use in Java programming is a null literal.**
- **We specify the Null literal in the source code as 'null'.**
- **To reduce the number of references to an object, use null literal.**
- **The type of the null literal is always null.**
- **We typically assign null literals to object reference variables.**

For instance
String s = null;

Boolean Literals :

- **The values are true and are treated as literals in Java programming.**

- When we assign a value to a boolean variable, we can only use these two values.
- Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java.
- We have to use the values true and false to represent a Boolean value.

Example :

```
boolean b= false;
```

- Remember that the literal true is not represented by the quotation marks around it.
- The Java compiler will take it as a string of characters, if it's in quotation marks.

Java Arrays :

- An array is a container that holds data (values) of one single type (homogeneous).
- For example, you can create an array that can hold 100 values of int type.
- Array is a fundamental construct in Java that allows you to store and access large numbers of values conveniently.

How to declare an array?

- Here's how you can declare an array in Java:

```
dataType[] arrayName;
```

- dataType can be a primitive data type like: int, char, Double, byte etc.
- or an object.
- arrayName is an identifier.

Example :

```
double[] data;
```

- Here, data is an array that can hold values of type Double

How many elements can this array hold?

- The next step is to allocate memory for array elements.

```
data = new double[10];
```

- The length of the data array is 10.
- Meaning, it can hold 10 elements (10 Double values in this case).
- Note, once the length of the array is defined, it cannot be changed in the program.

Another example:

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

Here, the age array can hold 5 values of type int.

- **It's possible to declare and allocate memory of an array in one statement.**
- **You can replace the two statements above with a single statement.**

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

Java Array Index

- **You can access elements of an array by using indices.**
- **Let's consider the previous example.**
- **int[] age = new int[5];--> 0 to 4**

The first element of the array is age[0], second is age[1] and so on.
If the length of an array is n, the last element will be arrayName[n-1].
Since the length of age array is 5, the last element of the array is age[4] in the above example.
The default initial value of elements of an array is 0 for numeric types and false for boolean.

We can demonstrate this:

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

    int[] age = new int[5];

    System.out.println(age[0]);
    System.out.println(age[1]);
    System.out.println(age[2]);
    System.out.println(age[3]);
    System.out.println(age[4]);
  }
}
```

There is a better way to access elements of an array by using looping construct (generally for loop is used).

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

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

    for (int i = 0; i < 5; ++i) {
      System.out.println(age[i]);
    }
  }
}
```

How to initialize arrays in Java?

- **In Java, you can initialize arrays during declaration or you can initialize (or change values) later in the program as per your requirement.**

- **Initialize an Array During Declaration**
- **Here's how you can initialize an array during declaration.**

  int[] age = {12, 4, 5, 2, 5};
- **This statement creates an array and initializes it during declaration.**

- **The length of the array is determined by the number of values provided which is separated by commas. In our example, the length of age array is 5.**

- **Let's write a simple program to print elements of this array.**

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

    int[] age = {12, 4, 5, 2, 5};

    for (int i = 0; i < 5; ++i) {
      System.out.println("Element at index " + i +": " + age[i]);
    }
  }
}
```

When you run the program, the output will be:

```
Element at index 0: 12
Element at index 1: 4
Element at index 2: 5
Element at index 3: 2
Element at index 4: 5
```

How to access array elements?

- **You can easily access and alter array elements by using its numeric index. Let's take an example.**

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

    int[] age = new int[5];

    // insert 14 to third element
    age[2] = 14;

    // insert 34 to first element
    age[0] = 34;

    for (int i = 0; i < 5; ++i) {
      System.out.println("Element at index " + i +": " + age[i]);
    }
  }
}
```

When you run the program, the output will be:

```
Element at index 0: 34
Element at index 1: 0
Element at index 2: 14
Element at index 3: 0
Element at index 4: 0
```

Example: Java arrays

The program below computes the sum and average of values stored in an array of type int.

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

    int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
    int sum = 0;
    Double average;

   for (int number: numbers) {
     sum += number;
   }

   int arrayLength = numbers.length;

   // Change sum and arrayLength to double as average is in double
   average =  ((double)sum / (double)arrayLength);

   System.out.println("Sum = " + sum);
   System.out.println("Average = " + average);
  }
}
```

When you run the program, the output will be:

```
Sum = 36
Average = 3.6
```

Couple of things here.

- **The for..each loop is used to access each element of the array.**
- **Learn more on how for...each loop works in Java.**
- **To calculate the average, int values sum and arrayLength are converted into double since average is double.**
- **This is type casting. Learn more on Java Type casting.**
- **To get the length of an array, length attribute is used. Here, numbers.length returns the length of the numbers array.**

Multidimensional Arrays

- **Arrays we have mentioned till now are called one-dimensional arrays.**
- **In Java, you can declare an array of arrays known as a multidimensional array.**
- **Here's an example to declare and initialize a multidimensional array.**

```
double[][] matrix = {{1.2, 4.3, 4.0},
    {4.1, -1.1}
 };
```

Here, the matrix is a 2-dimensional array.

- **To create and use an array of primitive data types (like: Double, int etc.), String array, and array of objects.**
- **It's also possible to create an array of arrays known as a multidimensional array.**

For example,

```
int[][] a = new int[3][4];
```

- **Here, a is a two-dimensional (2d) array.**
- **The array can hold a maximum of 12 elements of type int.**
- **Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.**
- **Similarly, you can declare a three-dimensional (3d) array.**
  For example,

```
String[][][] personalInfo = new String[3][4][2];
```

- **Here, personalInfo is a 3d array that can hold maximum of 24 (3*4*2) elements of type String.**

- **In Java, components of a multidimensional array are also arrays.**

How to initialize a 2d array in Java?

- **Here's an example to initialize a 2d array in Java.**

```
int[][] a = {
    {1, 2, 3},
    {4, 5, 6, 9},
```

```
    {7},
};
```

- As mentioned, each component of array a is an array in itself, and length of each row is also different.

write a program to prove it.

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

    int[][] a = {
        {1, 2, 3},
        {4, 5, 6, 9},
        {7},
    };

    System.out.println("Length of row 1: " + a[0].length);
    System.out.println("Length of row 2: " + a[1].length);
    System.out.println("Length of row 3: " + a[2].length);
  }
}
```

When you run the program, the output will be:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

- Since each component of a multidimensional array is also an array (a[0], a[1] and a[2] are also arrays), you can use length attributes to find the length of each row.

Example: Print all elements of 2d array Using Loop

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

    int[][] a = {
        {1, -2, 3},
        {-4, -5, 6, 9}
            };
```

```
    for (int i = 0; i < a.length; ++i) {
      for(int j = 0; j < a[i].length; ++j) {
        System.out.println(a[i][j]);
      }
    }
  }
}
```

- It's better to use for..each loop to iterate through arrays whenever possible.
- You can perform the same task using for..each loop as:

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

    int[][] a = {
        {1, -2, 3},
        {-4, -5, 6, 9},
        {7},
    };

    for (int[] innerArray: a) {
      for(int data: innerArray) {
        System.out.println(data);
      }
    }
  }
}
```

When you run the program, the output will be:

```
1
-2
3
-4
-5
6
9
7
```

How to initialize a 3d array in Java?
You can initialize a 3d array in a similar way like a 2d array. Here's an example:

```
// test is a 3d array
int[][][] test = {
            {
             {1, -2, 3},
             {2, 3, 4}
            },
            {
             {-4, -5, 6, 9},
             {1},
             {2, 3}
            }
};
```

Basically, a 3d array is an array of 2d arrays.

Similar to 2d arrays, rows of 3d arrays can vary in length.

Example: Program to print elements of 3d array using loop

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

    // test is a 3d array
    int[][][] test = {
            {
             {1, -2, 3},
             {2, 3, 4}
            },
            {
             {-4, -5, 6, 9},
             {1},
             {2, 3}
            }
    };

    // for..each loop to iterate through elements of 3d array
    for (int[][] array2D: test) {
      for (int[] array1D: array2D) {
        for(int item: array1D) {
          System.out.print(item);
        }
```

```
      }
    }
  }
}
```

When you run the program, the output will be:

```
1
-2
3
2
3
4
-4
-5
6
9
1
2
3
```

Types of variables :

Division 1 :

- **Based on type of value represented by a variable all variables are divided into 2 types;**

  1)Primitive variables: can be used to represent primitive values. (* TYPES)

```
Ex:
int a=10;
static int b=10;
m1(){
int x=10;
}
```

2) reference variables: can be used to refer to objects.

Ex: Student s= new Student();

Division 2 :

- **Based on behavior and position of declaration all variables are divided into 3 types.**

1)Instance variables 2) static variables 3) local variables.

Instance variables :

- **If the value of a variable is varied from object → object such types of variables are called instance variables.**
- **For every object a separate copy of instance variables will be created.**
- **Instance variables will be created at the time of object creation and destroyed at the time of object destruction, hence the scope instance variable is exactly the same as the scope of object.**
- **Instance variables will be stored in the heap memory as the part of an object.**
- **Instance variables should be declared within the class directly but outside of any method or constructor or block.**
- **We can't access instance variables directly from static areas but we can access them by using object reference.**
- **From the instance area we can access instance members directly.**
- **For instance, we are not required to perform initialization explicitly, because the JVM always provides default values.**
- **Instance variables are also known as object level variables or attributes.**

Static variables :-

- **If the value of a variable is not varied from object to object then it is recommended to declare that variable as a static variable, we have to declare such types of variables at class level by using a static modifier.**
- **In the case of instance variables a separate copy will be created for every object but in the case of static variables a single copy will be created at class level and shared by every object of that class.**

- **static variables will be created at the time of class loading and destroyed at the time of class unloading.**

- **Hence scope of static variables is exactly the same as the scope of the class.**

- **static variables will be stored in the method area.**

- **static variables should be declared within a class directly but outside of any method or block or constructor.**

- **We can access static variables either by object reference or by class name but recommended using class names..**

- **within the same class it's not required to use class names also and we can access directly.**

- **We can access static variables directly from both static and instance areas.**

- **For static variables JVM will provide default values and we are not required to perform initialization explicitly.**

- **static variables also known as class level variables or fields.**

Ex: Testvariables4.java

Note : If we perform any change to the instance variable that change will be reflected only for that particular object because for every object a separate copy of the instance variable is available.

- **If we perform any change to the static variable that change will be reflected for all objects. Because a single static variable is shared by all objects of that class.**

Local variables:

- **Sometimes we meet temporary requirements of programmers.**

- **We can declare variables inside a method / block /constructor such as local variables or temporary variables or automatic variables or stack variables.**

- **local variables will be stored inside stack memory.**
- **local variables will be created while executing the block in which we declared that variable.**

- **Once that block executing is completed, automatically that local variable will be destroyed.**

The scope of the local variable is the block in which we declare that local variable i.e from outside of the block we can't access.

Note : It is not recommended to perform initialization for local variables logical blocks.
Because there is no guarantee for the execution of its blocks at runtime.

→ It is highly recommended to perform initialization for local variables at the time of declaration at least with default values.

Note: The only applicable modifier for local variables is final by mistake if we are trying to declare with any other modifier we will get a compile time error.
Ex: Testvariables5.java

→ If we are not declaring with any modifier then it is <default> by default but this rule is applicable only for instance and static variables but not for local variables.

Conclusion :

- **For instance, static variables JVM will provide default values and we are not required to perform initialization explicitly, but for local variables JVM won't provide any default values, we should perform initialization explicitly before using that variable.**

Var arg methods :-

- **Until the 1.4 version we can't declare a method with a variable number of arguments.**
- **If there is a change in the number of arguments compulsory we should declare a new method which increases length of the code and reduces readability.**
- **To overcome this problem sun people introduced var arg methods in 1.5 version.**
- **According to this we can declare a method which can take any number of arguments, such types of methods are called var arg methods.**
- **We can declare a var arg method as follows.**
  m1(int ...x);

- **We can call this method by passing any number of int values including zero.**
  m1();
  m1(10, 20, 30);
  m1(10, 20, 30, 40, 50);

- **Ex : varargs Example 1:**

- **Internally var arg parameter will be converted into a one dimensional array.**
- **Hence within the var arg method we can differentiate values by using "index".**

- **Ex: Varargs Example 2**

Conclusions :

Case1:
m1(int… x);
m1 (int ...x);
m1(int x...);
m1(int. ..x);
m1(int ...x);
m1(int .x..);

Case 2:
- **We can mix var arg parameters with normal parameters .**
  m1(char ch, String… s);
  m1(int x, int… g);

Case 3 :
- **If we mix var arg parameter with normal parameter then var arg parameter should be the last parameter.**
  Ex: m1(double ... d, int x); CE
  m1(int x, double… b);

Case 4:
- **In the var-arg method we can take only one var-arg parameter if we are trying to take more than one var- arg parameter then we will get a compile time error.**

Ex: m1(double … x, int ...y);

Case 5 :
- **In general var arg method will get the lowest priority i.e if no other method matched then only var arg() will get a chance.**

It has exactly the same "default" case inside the switch.

Ex: Varargs Example

Case 6 :

- **With the same class we can't declare var arg method and corresponding array argument method simultaneously otherwise we will get a compile time error.**

```
Ex:
class Test {
        p s v m 1(int… x){   ----   }
public void m1(int[] x){-----}

        }
CE: can not declare both m1(int[]) and m1(int ...) in Test.
```

Equivalence between var arg parameter and one dimensional array :

Case 1 :

- **wherever one dimensional array is present we can replace it with a var arg parameter.**

m1(int [ ] x) ⇒ m1(int... x)

Ex: main(String[] args) ⇒ main(String… args);

Case2 :

- **wherever the var arg parameter is present we can't replace it with a one dimensional array.**
  m1(int… x) ⇒ m1(int[ ] x) ;

Note :

m1(int… x) ⇒ int[] x;

- **We can call this method by passing any number of int values and x will become a one- dimensional int array.**

m1(int[].. x) ⇒ int[][] x;

- **we can call this method by passing a group of dimension arrays and x will become a 2-0 int array.**

main() method.

- **whether class contains the main() method or not and whether main() is declared according to requirement or not.**
- **These things won't be checked by the compiler.**
- **At runtime JVM is responsible for checking these things.**
- **If JVM is unable to find the main() method then we will get a Runtime Exception saying.**

Error: Main method not found in class.

```
class Test { }
Javac Test.java
Java Test
```

RE: NoSuchMethodError: main

- **At runtime JVM will always search for the main() method with the following prototype.**

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

}
```

public → To call by JVM from anywhere.

static → without an existing object also JVM has to call this method or main() functionality is independent of Object.

void → main method won't return anything to JVM.
main → this is the name which is configured inside JVM.
String[] → command line argument.

- **If we perform any change to the above syntax, we will get a Runtime exception saying NoSuchMethodError and we won't get any compile time error.**
- **Even Though above syntax is very strict, the following changes are acceptable.**
- **We can change the order of modifiers i.e instead of public static we can take static public.**

- **We can declare String[] args in any acceptable form.**
  - **main(String[] args)**
  - **main(String args[])**

3)instead of args we can take any valid Java identifier.

4) We can replace String[] with the var arg parameter.

main(String… args);

5) We can declare main() method the following modifiers also
final 2) synchronized 3) strictfp

→ Which of the following main() method declarations are valid
public static void main(String args); //Invalid
public static void Main(String[] args); //Invalid
public void main(String args[]);//Invalid
public static int main(String args[]);//Invalid
final synchronized strictfp Public static void main(String args[]);//Invalid
final synchronized strictfp public static void main(String args[]); //valid
public static void main(String… args); //valid

Case 1 :
→ overloading of the main() method is possible but JVM will always call String[] argument main() only.
The overloaded method we have to call explicitly is like a normal method call.

Case 2 :
→ Inheritance concept applicable for main() method hence parent class main() by default available to child class due to this while executing child class, if a child doesn't contain the main method then the parent class main method will be executed.

Case 3 :
→ Overriding concept applicable for main() method but it is not overriding and method hiding.

Command line arguments:

The arguments which are passed from the command prompt are called command line arguments.
Java Test A B C
args[0] args[1] args[2];
args.length=3;

→ The main objective of command line arguments is that we can customize the behavior of main().

→ Within the main method command line arguments are available in String form.

```
Class Test{
        p s v  m(String[] args){
                for(int i=0 ; i<=args.length; i++){
                        sop(args[i]);
                }
}
}
```

Java Coding standards :

- **Readability of the code will be improved.**
- **class name Should start with a capital letter and camelcase.**

Ex: HelloWorld

- **Method, variables names should start with small and camel case**

Ex: additionNumbers
Ex: age, gender , name

- **package names should be small.**

Ex: opps, langfanda

Constructors

- **Constructor in java is used to create the instance of the class.**
- **Constructors are almost similar to methods except for two things - its name is the same as the class name and it has no return type.**
- **Sometimes constructors are also referred to as special methods to initialize an object.**

Constructor Types

1) Default constructors.

- **It's not required to always provide a constructor implementation in the class code.**
- **If we don't provide a constructor, then java provides default constructor implementation for us to use.**
- **Let's look at a simple program where the default constructor is being used since we will not explicitly define a constructor.**
- **Default constructor's only role is to initialize the object and return it to the calling code.**
- **Default constructor is always without argument and provided by the Java compiler only when there is no existing constructor defined.**

2) No-Arg Constructors.

- **Constructor without any argument is called a no-args constructor.**
- **It's like overriding the default constructor and used to do some pre-initialization stuff such as checking resources, network connections, logging, etc.**
- **Let's have a quick look at the no-args constructor in java.*/**

3) Parameterized Constructors.

- **Constructor with arguments is called a parameterized constructor.**
- **Let's look at the example of a parameterized constructor in java.*/**

Constructor Overloading in Java

- **When we have more than one constructor, then it's constructor overloading in java.**
- **Let's look at an example of constructor overloading in a Java program.*/**

Private Constructor in Java

- **Note that we can't use abstract, final, static and synchronized keywords with constructors.**
- **However we can use access modifiers to control the instantiation of class objects.**
- **Using public and default access is still fine, but what is the use of making a constructor private?**
- **In that case any other class won't be able to create the instance of the class.**
- **Well, a constructor is made private in case we want to implement a singleton design pattern.**
- **Since Java automatically provides default constructor, we have to explicitly create a constructor and keep it private.**
- **Client classes are provided with a utility static method to get the instance of the class.**
- **An example of a private constructor for TestConstructorsDemo class is given below.**

Constructor Chaining in Java

- **When a constructor calls another constructor of the same class, it's called constructor chaining.**
- **We have to use this keyword to call another constructor of the class.**
- **Sometimes it's used to set some default values of the class variables.**
- **Note that another constructor call should be the first statement in the code block.**
- **Also, there should not be recursive calls that will create an infinite loop.**

Example :

```
package com.languagefundamentals;
/*Let's see an example of constructor chaining in java program.*/
public class TestConstructorsDemo {
        int id;
        String name;
```

```
	// parameterized constructors
	TestConstructorsDemo(int id, String name, int age) {
		this.id = id;
		this.name = name;
		System.out.println(name + " Age : " + age);
	}

	// no argument constructors
	public TestConstructorsDemo() {
		this(18, "Kohli", 34); // Constructor Chaining in Java
		System.out.println("No arg constructors !!");
	}
	public static void main(String[] args) {
		System.out.println("Hello main method !!");
		// Creating an Object
		// TestConstructorsDemo() --> default constructors in Java
		TestConstructorsDemo t = new TestConstructorsDemo();
		TestConstructorsDemo t1 = new TestConstructorsDemo(7, "Dhoni", 40);
		System.out.println(t1.id);
		System.out.println(t1.name);
	}
	static {
		System.out.println("Hello static block !!");
	}

	{
		System.out.println("Hello instance Block !!");
	}
}
```

****End of Language Fundamentals****

Declarations and Access modifiers

Access Modifiers Table with 12 keywords 👍

Access Modifier	Class	Inner classes	Methods	Variables	Blocks	Interface	Scope

private	**NO**	**YES**	**YES**	**YES**	**NO**	**NO**	**Within the class only.**
<default>	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**Within the package.**
protected	**NO**	**YES**	**YES**	**YES**	**NO**	**NO**	**Within the package + outside of the packages of sub classes with subclasses objects.**
public	**YES**	**YES**	**YES**	**YES**	**NO**	**YES**	**Anywhere in the project.**
abstract	**YES**	**YES**	**YES**	**NO**	**NO**	**NO**	
static	**NO**	**YES**	**YES**	**YES**	**YES**	**NO**	
synchronized	**NO**	**NO**	**YES**	**NO**	**YES**	**NO**	
strictfp	**YES**	**YES**	**YES**	**NO**	**NO**	**YES**	
final	**YES**	**YES**	**YES**	**YES**	**NO**	**YES**	**final is for class means, we cannot extend that class.** **final is for methods means, we cannot override the method.** **Final is for variables. Then we can not change values.**
transient	**NO**	**NO**	**NO**	**YES**	**NO**	**NO**	
volatile	**NO**	**NO**	**NO**	**YES**	**NO**	**NO**	
native	**NO**	**NO**	**YES**	**NO**	**NO**	**NO**	

Java source file structure.

- **class level modifiers :-(classes)**
- **member level modifiers :- (variables and methods)**
- **A java program can conduct any number of classes but at most one class can be declared as public.**

- **If there is a public class then the name of the public class and the name of the program must be the same otherwise we will get a Compile Time Error .**

- **If there is no public class then any name we can use for the java program and there are no restrictions.**

- **case i)**
- **If there is no public class then we can use any name for java program and there are no restrictions**
- **ex:- A.java B.java C.java Test..java**

- **Case ii)**
- **If class B is declared as public and the name of the program is Test.java.**
- **Then we will get a compile time error saying Class B public should be declared in a file named B.java.**

- **case iii)**
- **If we declare both B and C classes as public and the name of the program is B.java then we will get a compile time error saying class C is public, Should be declared in a file named C.java.**

Ex: Test.java

Conclusions

- **We can compile a java program but we can't run a java class.**
- **Whenever we compile a Java program for every class in that program a separate .class file will be generated.**
- **Whenever we are executing a java class the corresponding class main method will be executed.**
- **If the class doesn't contain the main method then we will get R E saying NoSuchMethodError:main.**
- **Whenever we are trying to execute a java class . If the corresponding .class file is not available then we will get R E saying NoClassdefFoundError:B**
- **It is not recommended to take multiple classes in the same program.**
- **It is highly recommended to declare only one class for the source file and the name of the file and the name of the class must be matched. The advantage of this approach is readability and maintainability of the application will be improved.**

import statement :

Case 1)

- **There are 2 types of import statements**

1)Explicit class import
2)Implicit class import

Explicit class import :

- **Ex : import java.util.ArrayList;**
- **Recommended to use because it improves readability of the code.**

Implicit class import :

- **Ex: import java.util.*;**
- **Not recommended to use because it reduces readability of the code.**

Case 2)

- **Which of the following statements are meaningful?**

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

Case 3)

- **Consider the following code**

```
class Myobject extends java.rmi.RMISecurityManager {}
 {}
```

- **Even though we are not writing an import statement the code compiles fine because we are using a fully qualified name.**
- **Note :**
- **Whenever we are writing a fully qualified name, it is not required to write an import statement.**
- **Similarly whenever we are writing an import statement it is not required to write a fully qualified name.**

Case 4)

- **Ex:**

```
import java.util.*;
import java.sql.*;
class Test{
    public static void main(String args[]){
        Date d= new Date();
        }
}
```

Note : Even in the case of list also we may get same ambiguity problem because it is available in both util and AWT packages (List)

Case 5)

```
import java.util.Date;
import java.sql.*;
class Test{
        public static void main(String args[]){
                Date d= new Date();
        }
}
```

Note : In the above example util package Date will be considered .

- **While resolving class names, compilers will always consider precedence in the following order.**
  i) Explicit class imports
  ii)classes present in current working directories.
  iii)implicit class import.

Case 6)

- **Whenever we are importing a (util) package all classes and interfaces present in that (util)package are by default available but not sub package classes.**
- **To make sub package class compulsory we should write import statements until subpackage level.**

Ex: To use the Pattern class in our program directly, which import statement is required.
i) import java.util.*;
ii) import java.util.regex.Pattern;
iii) import java.*;
iv) No import is required.

Case 7)

- **The following 2 packages are not required to import, because all classes and interfaces present in these two packages are by default available in every java program.**
  i) java.lang package
  ii) default package i.e current working directory.

Case 8)

- **Import statements are totally a compile time issue.**
- **if there are more import statements then more will be the compile time.**
- **But there is no effect on execution time.**

Case 9) :

- **Difference between C and java language import statements.**

Note :

- **In the case of c language # include all specified header files will be loaded at the beginning only.**
- **This is something like static loading.**
- **But in the case of a java language import statement,no .class file will be loaded at the beginning.**
- **Whenever we are using a particular class then only the corresponding .class file will be loaded.**
- **This is something like dynamic loading(load- on fly or load- on demand).**

Static import

- **According to Sun people static import improves readability of the code but according to world wide programming experts (like us) usage of static import reduces readability and creates performance problems.**
- **Hence if there is no specific requirement then it is not recommended to use static imports.**
- **Usually we can access static members by using class names but whenever we are writing static imports, it's not required to use class names to access static members and we can access them directly.**

Explain about System.out.println();

```
class Test {
        static String name = "Srikanth";
}

Test.name.length(); //8
```

→ 'Test' is a class
→ 'name' is a static variable of type String present in Test class.
→ 'length()' is a method present in String class.

```
class System{
        static PrintStream out;
        }
System.out.println();
```

- **'System' is a class present in a java.lang package.**
- **'out' is a static variable of the PrintStream present in System class.**
- **'println()' method is present in a PrintStream class.**

Note : 'out ' is a static variable present in System class, hence we can access it by using class name System but whenever we are writing static imports it is not required to use class name and we can access directly.

Ex:

```
import static java.lang.System.out;
class Test {
        public static void main(String args[]){
                out.println("Hello");
                out.println("Hello");
                }
        }
```

Ex:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test {
        public static void main(String args[]){
                System.out.println(MAX_VALUE);
                }
        }
```

- **While resolving static members, the compiler will always use the precedence in the following order.**
  1) **Current class static members.**
  2) **Explicit static import.**
  3) **Implicit static import.**

Ex:TestSystem Out.java

```
import static java.lang.Integer.MAX_VALUE; //2
import static java.lang.*; //3

class Test{
        //static int MAX_VALUE=999; //1
        {
                public static void main(String args[]){
                        System.out.Println(MAX_VALUE);
        }
}
```

- **If we comment line 1) then explicit static import will be considered. In this case output is 2147483647(Integer class MAX_VALUE).**
- **If we comment both lines 1 and 2 then output is 127.(Byte class MAX_VALUE).**
- **Two packages contains a class with the same name is very rare and hence ambiguity**
- **The problem in the case of normal import is also very rare.**
- **But two classes containing a variable or method with the same name is a very common problem in static imports.**
- **Usage of static import reduces readability and understandability of the code hence if there are no specific requirements then it's not recommended to use static import.**

Packages or Package statement :

- **A group of related classes and interfaces into a single unit is nothing but Package. or**
- **Package is an encapsulation mechanism to group related classes and interfaces into a single unit.**
- **Ex: All classes and interfaces which are useful for database operations or grouped into a separate package which is nothing but java.Sql package.**
- **Ex: All classes and interfaces meant for file IO operations or grouped into a separate package which is nothing but java.io package.**

Advantages of package statement :

- **We can resolve naming conflicts.**
- **It improves the modularity of the application.**
- **It provides security for our classes.**
- **There is one universally accepted naming convention for packages i.e use the internet domain name in reverse.**

Ex:

- **com.icici bank.loan.housing loan.Account**
- **com.icicibank→ client internet domain name in reverse loan → module name**

housing loan → sub module name.

Account → class name

- **Ex: JavaC Test.java**
- **Generated .class file will be placed in the current working directory.**
- **javac -d . Test.java**

"-d" means destination to place generated class files.

- **Generated .class file will be placed in the corresponding package structure.**

- **If the package structure is not already available then this command itself will create the required package structure.**
- **instead of . we can use any valid directory name.**

Ex: javac -d c: Test.java

- **If the specified destination is not already available then we will get a compile time error .**

Ex: javac -d Z: Test.java

- **If 'Z: ' is not available then we will get a compile time error, at the time of execution compulsory we have to use fully qualified name**

```
Java com.srikanth.scjp.Test
```

Conclusion 1:

- **Inside any java program at most one package statement is allowed.**
- **i.e if we are trying to take more than one package statement then we will get a compile time error.**

```
Ex:
        package pack1;
        //package pack 2;
        Class A{}
```

Conclusion 2 :

- **Inside any java program the first non comment statement should be a package statement (if it is available) otherwise we will get CE.**

```
Ex:
        //import java.util.*
        Package pack1;
        Class A{ }   ⇒ CE: class, interface or enum expected.
```

The following is a valid java source file structure.

At most one package statement
Any number of import statement
Any number of class / interface / enum declarations

Note : An empty source file is a valid Java program the following are valid java source files.
Test.java
package pack1;
import java.util.*;

Test.java
→ import java.util.*;

Test.java

→ class { --- } :

class level modifiers :

- **Whenever we are writing a java class, we have to provide some information about our class to the JVM.**
- **Whether this class is accessible from anywhere or not.**
- **Whether child class creation is possible or not.**
- **Whether object creation is possible or not .**
- **We can specify this information by declaring with appropriate modifiers.**
- **Only applicable modifiers for top level classes are**

public
default
final
abstract
strictfp

- **But for inner classes applicable modifiers are**

public	**private**
default	**protected**
final	**+ static**
abstract	
Strictfp	

Access specifiers and Access modifiers.

- **public, private, protected and default considered as Access specifiers.**
- **Except these remaining are considered as modifiers but this rule is applicable, only for old languages but not for java.**
- **In Java all are considered as Access modifiers only.**
  There is no terminology like access specifiers.

ex:- private class Test{
}
CE: modifier private not allowed here.

Public classes

- **If a class is declared as public then we can access that class from anywhere, either within the package or from outside the package.**

```
package pack1 ;
public class A
{
        public void m1(){
          System.out.println("A class method ");
}
}
package pack2;
import pack 1.A;
class B {
  Public static void main(String args[]){
         A a= new A();
          a.m1();
        }
}
```

- **In the above example class A is not public then while compiling B class, we will get a compile time error. saying**
  pack1.A is not public and pack1 cannot be accessed from outside the package.

default classes

- **If a class is declared with a default modifier (i.e if we are not declared in public) then we can access that class only within the current package i.e from outside the package we can't access.**
- **Hence default access is also known as package level access.**

final modifier

- **final is the modifier applicable for classes, methods and variables.**

final methods

- **Whatever methods a parent has by default available to the child, sometimes the child may not satisfy the parent implementation then the child is allowed to redefine that method based on its requirement, this is called overriding.**
- **If the parent method is declared with the final then we can't override that method in the child class.**
- **If we are trying to override then we will get a compile time error.**

final class

- **If a class is declared as a final then we can't extend that class i.e we can't create a child class for that class.**
  ex:-
- **Note : Every method present inside the final class is always final whether we are declaring or not.**

- **But every variable present inside a final class need not be final**
  Note: The main advantage of final keyword is
- **We can achieve security because no one is allowed to change our implementation.**
- **But the problem with the final keyword is we are missing the benefits of oops .**

1) **Inheritance → final classes**
2) **Polymorphism → final methods**

- **Hence if there are no specific requirements then it is not recommended to use the final keyword.**

abstract modifiers

- **abstract is a modifier applicable for classes and methods but not for variables.**

abstract method :

- **Even Though, we don't know about implementation still we can declare methods with abstract i.e for abstract methods only declaration is available but not implementation.**
- **Hence abstract method declaration should end with a semicolon.**

```
//valid
public abstract void sum() ;

//invalid
public abstract void m1(){
        System.out.println("Hello");
}
```

- **child class is responsible to provide implementation for parent class abstract methods.**

- **The main advantage of declaring abstract methods in the parent class is we can provide guidelines to the child classes such that which methods are compulsory they require to implement.**

- **abstract modifiers never talk about implementation.**
- **If any modifier talks about implementation then it finds illegal combinations with abstract modifiers.**
- **The following are various illegal combinations of modifier methods.**

abstract → final
abstract → synchronized
abstract → native
abstract → private
abstract → static
abstract → strictfp

abstract class

- **For any java class if we don't want object creation then we have to declare that class with abstract modifier i.e for abstract classes instantiation is not possible.**

ex:-

```
abstract class Test
{
public abstract m1();
}
```

1)
Test t = new Test1();
C.E: Test is abstract : cannot be instantiated.

abstract class vs abstract method

- **If a class contains at least one abstract method then it is compulsory, we should declare the class as abstract, otherwise we will get compile time errors.**
- **reason : If a class contains at least one abstract method then implementation is not complete.**
- **Hence we can't create objects to restrict object creation, we should declare class as abstract.**
- **Even though class doesn't contain any abstract method, still we can declare class as abstract i.e an abstract class can contain zero number of abstract methods also.**
- **Ex: HttpServlet class is abstract but it doesn't contain any abstract method.**
- **Every adapter class is recommended to declare as abstract even though it does not contain any abstract methods.**

Abc interface → declaration 1000 methods
Xyz class implements Abc→ should define 1000 methods.

abstract vs final

- **final methods we can't override.**
- **abstract methods we should override**

Strictfp modifier :- 1.2 version

- **This modifier is applicable for classes and methods but not for variables.**

strictfp method ;

- **The result of floating point calculation varies from platform to platform.**
- **If we want platform independent results then we should go for**

strictfp modifier.

- **If a method is declared as strictfp then all floating point calculation in that method has to follow IEEE 754 standard so that we will get platform independent results.**

strictfp class.

- **If a class is declared as strictfp every concrete method in that class has to follow IEEE 754 standard . so that we will get platform independent results.**

strictfp vs abstract :

- **strictfp method always talks about implementation whereas abstract method never talks about implementation, hence abstract, strictfp is an illegal combination for methods.**
- **We can declare abstract class with strictfp modifier**

i.e abstract strictfp combination is legal for class but illegal for methods.

Ex:
abstract strictfp class Test{ } → Valid
abstract strictfp m1(); → Invalid
→ CE : Illegal combination of modifiers abstract and strictfp.

Members(variables or Methods) modifiers :

public members :

- **If a member is public then we (can access that member from anywhere) but the corresponding class should be visible i.e before checking member visibility we have to check class visibility.**

```
package pack1;
class A{
        public void m1(){}
        }
Public class B {
        P s v m(String args){
                A a = new A();
                a.m1();

}

package pack2;
import pack 1.B;

class D extends A  {
        P s v m(String args){
                D d = new A();
                d.m1();
}
}
```

- **In the above example even though m1 is public we can't access it from outside pack1 because the corresponding class A is not public.**
- **If both class and method are public then only we can access the method from outside the package.**

default members :

- **If a member is declared as default then we can access that member within the current package anywhere hence default access is also known as package level access.**

private members :

- **If a member is declared as private then we can access that member only within the class i.e from outside of the class we can't access.**

- **private methods are not visible to the child classes whereas abstract methods should be visible to the child classes to provide implementation.**
- **Hence private abstract combination is illegal for methods.**

protected members :

- **If a member declared as protected then we can access that member**

within the current package anywhere and outside package only in child classes.

Protected = <default > + kids

- **Note :within the current package we can access protected members either by using parent references or by using child references but from the outside package we can access protected members only in the child classes and we should use child references only i.e parent references can't be used to access protected members from outside the package.**
- **we can access protected members from outside packages only in child classes and we should use child references only (compulsory current class child references only).**

Ex: From D class if we want to access we should use D references only.

Note :

→ The most restricted modifier is private.

→ The most accessible modifier is public.

→ private < default < protected < public

1<2<3<4

→ Recommended modifier for data members (variable) is private.

→ Recommended modifiers for data members (methods) are public.

final variables:

final instance variables:

final static variables:

final local variables:

- **formal parameters**
- **Formal parameters of a method simply access local variables of that method.**
- **Hence we can declare a formal parameter as final.**
- **If we declare formal parameters as final then within the method we can't perform re assignment.**

Ex: FormalParamsTest.java

String ="sai" ;

static modifier

- **static is the modifier applicable for methods and variables but not for classes.**
- **we can't declare top level classes as static but we can declare inner classes as static.**
- **In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.**

- **We can't access instance variables directly from the static area. But we can access it directly from the instance area.**
- **We can access static variables directly from both instance and static areas.**
- **For static methods compulsory implementation should be there whereas abstract method implementation should not be available.**
- **Hence abstract and static combination is Illegal.**
  Ex: StaticTest.java

- **Case 1:- overloading concept is applicable for static methods including main methods.**
  Ex: StaticTest.java
- **Jvm will always call the String[] argument main method only then other loaded methods, we have to call explicitly then it will be executed just like the normal method.**

- **Case 2 :**
- **Inheritance concept is applicable for static methods including main method hence while executing child class, if a child doesn't contain main method then parent class main method will be executed.**

synchronized modifier

- **synchronize is a modifier that is applicable for methods and blocks but not for classes and variables.**
- **If a method or block is declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so we can overcome the DataInConsistency problem.**
- **But the problem with synchronized is that it increases thread time and creates performance problems.**
- **Hence if there is any specific requirement then only we can use synchronized .**

native modifier

- **native is the modifier applicable for methods but not for classes and variables.**
- **The methods which are implemented in non-java are called native methods or foreign methods.**
- **The main advantage of native keywords is to improve the performance of the system.**
- **To achieve memory level and machine level communication.**
- **To use already existing legacy non-java code.**
- **pseudo code to use native keyword :-**
- **For native methods implementation is already available in other languages and we are not responsible for providing implementation.**
- **Hence native declaration should hence with ";"**

public native void m1(); → valid
public native m1(){ } → not valid

- **For native methods inheritance, overloading and overriding concepts are applicable.**
- **The main methods of inheritance, overloading and overriding concepts are applicable.**
- **The main advantage of native keywords is, it improves the performance of the system but the main problem with native keywords is it breaks platform independent nature of java.**
- **For native method implementation is already available in old languages but for abstract methods implementation shouldn't be available, hence native abstract combination is an illegal combination for methods.**
- **native, Strictfp combination is an illegal method because there is no guarantee whether old language follows IEEE 754 standard or not.**

transient keyword

- **transient is the modifier applicable for only variables but not for classes and methods.**
- **transient keyword is useful in the Serialization context at the time of serialization, if you don't want to save a particular variable to meet security constraints then we have to declare that variable as transient.**
- **While performing serialization JVM ignores original value of transient value and saves default value to the file.**
- **Hence, transient means not to serialize.**

Id 211, name abs, transient age 22 → write → id name age → 0

Interfaces

→ Introduction
→ Interface declaration and implementation
→ extends vs implementation
→ interface methods
→ interface variables

→ interface naming conflicts
    Method naming conflicts
    Variable naming conflicts
→ marker interface
→ adapter interface
→ interface vs abstract class vs concrete class
→ Differences between interface and abstract class.

- interface Def 1:
- Any service requirement specification is considered as an interface .

- Ex:1
- JDBC API acts as a requirement specification to develop database drivers.
- Database vendor is responsible for providing implementation.
    SUN → JDBC API → Oracle Driver, Mysql driver, DBdriver.

- Ex:2
- Servlet API acts as a Requirement specification to develop web servers.
- Web server vendors are responsible for providing implementation.
- SUN→ Servlet API → Apache Tomcat Server, Oracle weblogic, IBM webSphere.

- interface Def 2 :
- From the client point of view, interfaces define the set of services that they expect.
- From a service provider point of view, the interface defines the set of services that are offered.
- Hence interface is considered as a contract between client and service provider.

- Ex: ATM GUI Screen represents the set of services that bank people are offering.
- At the same time the same GUI Screen represents the set of services that the customer is expecting, hence this GUI Screen acts as a contract between customer and Bank.

- interface Def 3:
- Inside the interface we can take only abstract methods and hence, the interface is considered as 100% pure abstract class.

Summary :

- Any service requirement specification (or) any contract between client and service provider or 100% pure abstract class is nothing but interface.

Interface declaration and implementation

- **Whenever we are implementing an interface for each and every method of that interface we should provide implementation otherwise we have to declare class as abstract.**
- **In this case the child class is responsible to provide implementation.**
- **Each and every interface method is always public whether we are declaring or not hence while implementing interface method compulsory we should declare as public otherwise, we will get compile time error.**

```
interface Interf
        {
                void m1();
                void m2();
        }
 class ServiceProvider implements Interf{
                public void m1(){
                        -----
                        -----
                                }
                public void m2(){ ----}
        }
        class SubSrerviceProvider extends ServiceProvider implements interf2  {
                public void m2(){ ----}
        }
```

extends vs implements :

- **A class can extend only one class at a time.**
- **A class can implement any number of interfaces simultaneously.**
  class A , class B
  Interf1, Interf2, Interf3 → 10 methods
  class A extends B implements Interf1, Interf2, Interf3{
- **A class can extend a class and can implement any number of interfaces simultaneously.**
- **An interface can extend any number of interfaces simultaneously.**

Ex :

interface A { }
interface B { }
interface C extends A, B { }

Q : Which of the following are valid.

- **A class can extend any number of classes simultaneously. invalid**
- **A class can implement only one interface at a time. Invalid**
- **A class can extend a class or implement an interface but not both simultaneously. invalid**
- **An interface can extend only one interface at a time. invalid**
- **None of the above.**
- **Consider expression A extends B for which of the following possibilities the above expression is valid .**
- **Both A and B should be classes. invalid**
- **Both A and B should be interfaces. invalid**
- **Both A and B either classes or interfaces. valid**

No restrictions.
Q: A extends B, C
→ A, B And C should be interfaces.

Q: A extends B implements C, D
→ A, B classes
→ c,d interfaces

Q: A implements B, C
A → class
B,C are interfaces

Q: A implements B extends C
Note : First we have to extends and then implements.

interface methods.

- **Every method present inside the interface is always public and abstract whether we are declaring or not.**
  interface Interf{
  void m1();
  }
- **Why public and abstract?**

- **public to make this method available to every implementation class.**
- **abstract implementation class is responsible for providing implementation.**

- **Hence inside the interface the following method declarations are equal.**
  void m1();
  public void m1();
  abstract void m1();
  public abstract void m1();
- **As every interface method is always public and abstract, we can't declare it with the following modifiers.**
- **public → private , protected.**
- **abstract → final, static, strictfp, native and synchronized.**
- **Inside interface which of the following method declaration are valid**
  → public void m1(){ } → Invalid
  → private void m1(); → Invalid
  → protected void m1(); → Invalid
  → static void m1(); → Invalid
  → public abstract native void m1(); → Invalid
  → abstract public void m1(); → valid

interface variables :

- **An interface can contain variables The main purpose of interface variables is to define requirement level constants.**
- **Every interface variable is always public, static and final, whether we are declaring or not.**
-

```
interface Interf
        {
              int x=10;
       }
```

- **public: To make this variable available to every implementation class.**
- **static: Without an existing object also we have to access this variable.**
- **final: implementation class can access but can't modify, because it is a common variable for all implementation classes.**

- **Hence the following variable declarations are equal inside the interface.**
  int x= 10;
  public int x=10;
  static int x=10;

final int x=10;
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;

- **As every interface variable is always public, static and final, we can't declare it with the following modifiers.**
  public → private, protected .
  static → transient.
  final → volatile.
- **For interface variables compulsory we should perform initialization at the time of declaration.**

Ex:
```
interface interf{
        int x; // CE
}
```

Q: Inside interface which of the following variable declarations are valid.?

int x ;
protected int x=10;
protected int x=10;
volatile int x=10;
transient int x=10;
public static int x=10;

- **From the implementation class we can access interface variables, but we can't modify its values.**

Ex :
```
interface interf{
        int x=10;
}

class Test implements interf{
public static void main (String args[]){
        int x=888;
        System.out.println(x);
        System.out.println(interf.x);
}
```

interface Naming conflicts :

- **method naming conflicts :**

- **Case 1: If two interfaces contain a method with the same signature (Method name and Method arguments) and same return type then in the implementation class one method implementation is enough.**

Ex :

→ interface left {
public void m1();
}
→ interface right{
public void m1();
Public void m2();

}

- **class Test implements left, right {**
public void m1(){ }
Public void m2(){ }

- **Case 2 :**
- **If two interfaces contain a method with the same name but different argument types then in the implementation class we have to provide implementation for both methods and methods that act as overloaded methods.**

Ex:

```
interface left {
public void m1();
}
interface right{
public void m1(int x);
}
class Test implements left, right {
// overloaded methods
public void m1(){}
}
public void m1(int x){}
}
```

- **Case 3:**
- **If two interfaces contain a method with the same signature**

(same method name and same arguments) but different return type then it is impossible to implement both interfaces simultaneously.

```
Ex:
        interface Left {
                void m1();
                }
    interface Right{
                int m1( );
                }
```

- **We can't write any class which implements both interfaces simultaneously .**

- **Q : Is it possible that a class can implement any number of interfaces simultaneously.?**
  Yes
- **Except if two interfaces contain a method with the same signature but different return types.**

variable naming conflicts :

- **Two interfaces can contain a variable with the same name and there may be a chance of variable naming conflicts using interface name.**

```
Ex :
→ interface Left {
        int x=888;
        }
→ interface Right{
        int x=999;
        }
class Test implements Left, Right{
        public static void main(String args[]){
                {
                        sop(x);                 // CE
                        sop(left.x);  // 888
                        sop(right.x); //999
                }
        }
```

Adapter classes

- **Adapter class is a simple java class that implements an interface only with empty implementation (dummy implementation)**

```
interface MyInterface{
        m1();
        m2();
        m3();
        ---
        ---
        m1000();
}
abstract class AdaptorClass implements MyInterface{
        m1(){
}
        m2(){
}
        m3(){}
        ----
        ----
        m1000(){
}
}
Class MyClass extends AdaptorClass {
        m1(){
        sop("Hiii");
}
        m2(){
                sop("Hiii");
}
}
```

- **If we implement an interface directly, we should provide implementation for each and every method of that interface whether it is required or not.**

```
class Test implements MyInterface{
        m1(){---}
        m2(){---}
        m3(){---}
        -----
        -----
        m1000(){---}
}
```

- The problem of this approach is that it increases the length of the code and reduces readability.
- To overcome this problem we should go for an adapter class.
- Instead of implementing interfaces directly, if we extend the adapter class then we have to provide implementation only for required methods and it is not required to provide implementation for every method of that interface.
- class Test extends AdaptorClass{
  m3(){---}
  }
- class Example extends AdaptorClass{
  m5(){---}
}
- class Demo extends AdaptorClass{
  m1000(){---}
}
- Hence the main advantage of adapter class is, it reduces the length of the code and improves readability.

Note :
- We can develop a servlet by implementing Servlet interface directly by extending.

Ex: GenericServlet

- If we implement Servlet interface directly, we should provide implementation for each and every method of Servlet interface .
- Instead of implementing Servlet interface directly, if we extend Genericservlet we have to provide implementation only for service() method and it is not required to provide implementation for all methods of the Servlet interface.
- Hence more or less GenericServlet acts as an adapter class for Servlet interface.
  MyServlet → implements Servlet (I) → extends Genericservlet (AC)

marker interface

- Marker interface in Java is an interface with no field or methods or in simple words an empty interface in java is called marker interface.
- Examples of market interface are Serializable, Cloneable and Remote interface.
- These are used to indicate something to the compiler or JVM.
- By implementing a Serializable interface our objects are able to save to the file and can travel across the network.
- By implementing a Cloneable interface our objects are in a position to produce exactly duplicate objects.

Q: Without having any methods in the marker interface how will the objects get some ability ?
Ans : → Internally JVM is responsible to provide that ability.

Q: Why is JVM providing required ability in marker interface?

Ans : → To reduce complexity of the programming.

Q : Is it possible to define our own marker interface?
Ans : → Yes But customization of JVM is required.

Note :-

- **If an interface doesn't contain any methods and required ability provided by JVM such types of interfaces are called marker interfaces.**
- **marker interface doesn't contain methods because the programmer is not responsible to provide ability and JVM is responsible for that.**
- **If an interface contains methods and programmers provide ability by implementing those methods, such types of interfaces are normal interfaces.**

Interfaces vs abstract vs concrete class :

- **If we never talk about implementation and we have just requirement specifications then we should go for interfaces.**
  Ex: Servlet
- **If we are talking about implementation but not completely (partial implementation)then we should go for abstract class**
  Ex: GenericServlet, HttpServlet

- **If we are talking about implementation completely and ready to provide service then we should go for concrete class**
  Ex: MyownServlet
  Servlet (I) → GenericServlet (AC) or HttpServlet(AC) → MyOwnServlet(CC)
  plan → partially implemented building → Fully completed building

Differences between interfaces and abstract class

interface	abstract
If we don't know anything about implementation and just have requirement specifications then we should go for interfaces.	If we are talking about implementation but not completely (partially implementation) then we should go for abstract class.
Every interface method is always public and abstract whether we are declaring or not.	abstract class methods need not be public and abstract. We can take concrete methods also inside abstract class.

We can't declare interface methods with the following modifiers. private, protected, final, native, strictfp, synchronized.	**There are no restrictions on abstract class method modifiers.**
Every variable inside the interface is always public, static and final whether we declare it or not.	**abstract class variables need not be public, static, final.**
We can't declare interface variables with the following modifiers private, protected, transient and volatile	**There are no restrictions on abstract class variable modifiers.**
For interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time errors.	**For abstract class variables it is not required to perform initialization at the time of declaration.**
Inside the interface we can't declare instance and static blocks.	**Inside abstract class can declare instance and static blocks.**
Inside the interface we can't declare constructors.	**inside abstract we can declare constructors**

******End of Declarations Access Modifiers.******

OOPs Concept

Encapsulation and Abstraction → Security
Polymorphism → flexibility
Inheritance → reusability

Encapsulation

Definition Encapsulation :

- **Binding the data and corresponding methods into a single unit is called encapsulation.**

```
Ex:
class Student{
    Data + methods.
}
```

- **If any component follows data hiding and abstraction, that component is said to be encapsulated.**

Encapsulation = Data hiding and Abstraction

- **The main advantage of Encapsulation are:**
- **i) We can achieve security.**
- **ii) Enhancements will be easy.**
- **iii) It improves maintainability.**

Data Hiding

- **Our internal data should not go out directly i.e.**
- **Outside people cannot access our internal data directly, this oop concept is nothing but Data Hiding.**
- **We can implement data hiding by declaring data members (variables+methods) with private modifiers.**
- **The main advantage of Data hiding is security.**
- **Recommended modifier for data members (Variables and methods) is private.**

Abstraction :

- **Hiding internal implementation and just highlighting the set of services that we are offering is the concept of abstraction.**

- **By using interfaces and abstract classes, we can implement abstraction.**

Interface example : MyInterface.java

```
interface MyInterface
{
  void addition();
  void Subtraction();
}
```

DemoInterface.java

```
public class DemoInterface implements MyInterface {
        private int j = 9;

private void method3(){
System.out.println(“Implementation of  Method 3”);
}

        public void addition() {
//implementation
        }

        public void subtraction() {
//implementation
        }

        public void method1() {
                System.out.println("implementation of method1");
        }

        public void method2() {
                System.out.println("implementation of method2");
        }

        public static void main(String arg[]) {
                MyInterface obj = new DemoInterface();
```

```
            obj.method1();
        }
}
```

The main advantage of Abstraction

- **We can achieve security, because we are not highlighting our internal implementation.**
- **With affecting outside people we can't perform any type of classes in our internal design, hence enhancement will become very easy.**
- **It improves the maintainability of the program.**

Inheritance :

- **IS-A relationship :**
- **It is also known as inheritance.**
- **By using extends keywords, we can implement inheritance.**
- **The main advantage of inheritance is code reusability.**

Without inheritance :

```
class  PLoan{
  300 methods
}
                                        900 methods
class  VLoan{
  300 methods
}
class  HLoan{
  300 methods
}
```

With inheritance

```
class Loan{
 250 common methods;
}                                       400 methods
```

```
class  PLoan extends Loan{
  50 specific methods.
}

class  VLoan extends Loan{
  50 specific methods
}

class  HLoan extends Loan{
 50 specific  methods
}
```

Note for Inheritance

- **The most common methods which are applicable for any child class are required to be defined in parent class.**
- **The most specific methods for a particular child class we have to define in child class.**

Conclusion:

- **Whatever method Parent has by default available to the child class.**
- **Hence by using child object reference we can call both parent and child class methods.**

```
Child c= new Child();
```

- **Whatever methods child has by default not available to the parent**
- **Hence by parent object reference we cannot call child specific methods.**

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

- **A parent reference can be used to hold a child object but by using that reference We cannot call child specific methods and we can call only parent class methods.**

```
Parent p1= new Child();
```

- **We cannot use child references to hold parent objects.**

```
Child c1 = new Parent();
```

- **Total Java API is implemented based on inheritance only.**
  Object → String,StringBuffer
  Number → Integer, Byte, Long
  Xyz.java
- **Note : Object class contains the most common methods which are applicable for any java class, hence the Object class acts as root so that its methods by default are available to every java class.**

Multiple Inheritance :

- **A java class can't extend more than one class simultaneously.**

Hence, Java won't provide support for multiple inheritance in classes.

```
Ex:
class A {
}
class  B{
}
class C extends A, B
{
}
```

Note : If our class won't extend any other class then our class is a direct child class of objects.

```
class A {                → A is a child of Object.
}
```

- **If our class extends another class then our class is an indirect child class of objects.**

```
class B extends A{
}
Class C extends B{
```

```
   C c = new C();
}
Class D extends A{
}
```

- **Hence either directly or indirectly java won't provide support for multiple inheritance in classes.**
- **Why Java doesn't provide support for multiple inheritance in classes : (-)**
  P1→ m1() P2→ m1()
  |
  c.m1(); → ambiguity problem
- **There may be a chance of ambiguity problems.**
- **But interfaces can extend more than one at a time hence java provides support for multiple inheritance in interfaces.**

```
interface A {

}
interface B{

}
interface C extends A, B{
}

interface A{
  public void m1();
  public void m2();
}

class D implements C{

}
```

- **Why ambiguity problem won't be rise in interface :-**
- **Even though multiple method declarations are available but implementation is unique hence there is no ambiguity problem in interfaces.**

```
PI1→  m1()   PI2 → m1()
|
CI → m1 () // Unique implementation
```

Note :

- **But strictly speaking we cannot consider the above as inheritance.**
- **In general inheritance concept is applicable for classes but not interfaces (because in the case interface won't get any code reusability)**

Cyclic inheritance

- **cyclic inheritance is not allowed in java**

```
Ex:- class A extends A
{
}
CE: Cyclic inheritance involving A .
Ex:
class A extends B
{
}
class B extends A{
}
```

Has-A relation :

- **It is also known as composition or aggregation.**
- **There is no specific keyword to implement HAS -A relation but mostly we can use new keywords.**
- **The main advantage of HAS -A relationship is reusability of the code.**

```
Ex :
class Car {
   Engine e= new Engine();
   e.m1();
}

public class Engine {
  Engine functionalities ;
   Void m1();
}

⇒ class Car has-A engine reference.
```

Method signature :

- **It consists of method names and arguments types.**

```
public void addition(float number1, float number2)
|
addition(10.5 , 60.5 )
```

- **In java return type is not part of method signature, the compiler will use method signature while resolving method calls.**

```
class Test
m1(){}
m1(int, float){}
m2(int){}
m1(int, float, boolean){}

Test t = new Test();
t.m1();
t.m1(10, 10.5f);
t.m2(10);
t.m1(10);
t.m1(10, 5.5f, false);

CE:  cannot find symbol
Symbol : method m1(int)
Location : class Test
```

- **Within a class two methods with the same signature are not allowed otherwise we will get compile time errors.**

```
class Test {
public void m1(int i){ }
public void m1(){
return  10;
}
```

Overloading :

- **Two methods are said to be overloaded if and only if both have the same name but different argument types.**

```
addition(int i)
addition(float  f) → overloading methods.
addition(String s);
addition(double s);
```

- **In 'C' we can't take two methods with the same name but different arguments.**
- **If there is a change in argument type, we should go for a new method name.**

```
Ex:  addition(int)
     laddition(long)
     faddition(float)
```

- **The lack of overloading in the C language increases the complexity of programming, but in java we can declare multiple methods with the same name but different argument types.**

```
Ex:-
abs(int)
abs(long)
abs(float)
```

- **These methods are overloaded methods.**
- **Hence having overloading concepts in java reduces complexity of programming.**
- **In overloading methods resolution always takes care by compiler based on type.**
- **Hence overloading is also considered as compile time polymorphism or static polymorphism or early binding.**

Case 1 :

- **While resolving overloading methods if an exact matched method is not available then we won't get compile time errors immediately.**
- **First compiler promotes the argument to the next level and checks whether a matched method is available or not.**
- **If the matched method is available, then it will be considered, otherwise the compiler promotes the argument once again to the next level.**
- **This process will be continued until all possible promotions, still if the matched method is not available then we will get a compile time error.**
- **This is called automatic promotion in overloading.**
- **The following are various possible automatic promotions in overloading.**
  byte → short → int → long → float → double
  char→ int
- **In overloading method resolution the exact match will get high priority.**
- **In overloading method resolution, the child argument will get more priority than the parent argument.**

Note :

- **In general var -arg method will get lowest priority, i.e no other method matched, then only var -arg method will get the chance.**
- **This is exactly the same as the default case in switch .**
- **In overloading method resolution always takes care by compiler based on reference type (but not based on runtime object)**
- **The parent class method which is overridden is called overridden method child class method which is overriding is called overriding method.**
- **The overriding method resolution always takes care of JVM based on runtime objects.**
- **Hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.**

Rules for overriding :

- **In overriding method names and argument types must be matched**
  i.e method signatures must be the same.
- **In overriding the return type must be the same but this rule is applicable until 1.4 version.**
- **From 1.5 version onwards covariant return types are allowed.**
- **The child class method return type need not be the same as parent class method return type when it would be a wrapper object type, its child types are also allowed.**

Parent class method return type → Object
Child class method return type → Object / String /StringBuffer.
Number→ Number/ Byte/ Short.
String → Object (no)
double → int (no)

- **covariant return type concepts applicable only for object types but not for primitives.**

Primitives :
- **byte, short, int, long, float, double, char and boolean.**

Wrapper :
- **Byte, Short, Integer, Long, Float,Double, Character, Boolean.**

Objects :
- **String, StringBuffer, Object, Integer, Float…. Ect… XYZ, ABC**

- **Parent class private methods are not applicable to the child classes, hence overriding concepts is not applicable for private methods.**
- **Based on our requirement we can define exactly the same private method in children, it is valid but not overriding.**
- **We can't override the final methods.**
- **We can override the Parent class abstract method in child class to provide implementation.**
- **We can override a non abstract method (Concrete method) as abstract.**
- **The main advantage of this approach is we can stop the availability of parent method implementation to the next level child classes.**
- **The following modifier won't keep any restrictions in**

overriding : Synchronized, native, strictfp.

- **While overriding we can't reduce the scope of the access modifier but we can increase the scope.**

Public → protected → default → → private

- **For private overriding, the concept is not applicable.**
- **If the child class method throws any checked exception, the compulsory parent class method should throw the same checked exception or its parent, but there are no restrictions for unchecked exceptions.**
- **overriding with respect to static methods**
- **i) we can't override a static method as non-static, otherwise we will get a compile time error.**
- **ii) Similarly, we can't override a non static method as static. .**

```
class P{
public static void m1(){}
}
class C extends P{
public  static  void m1() {}
}
}
```

- **It is method hiding but not overriding :**
- **It seems the overriding concept is applicable for static methods but it is not overriding, it is method hiding.**

Method hiding :

- **It is exactly the same as overriding except for the following differences.**

Method hiding :

- **Both parent and child class methods should be static.**
- **In method hiding method resolution is always takes care by compiler based on ref|| type.**
- **It is compile time polymorphism or static polymorphism or early binding.**

Overriding :

- **Both parent and child classes methods should be nonstatic.**
- **Method resolution always takes care of JVM, based on Runtime objects.**
- **It is runtime polymorphism or dynamic polymorphism or late binding.**
- **overriding with respect to var arg method :**
- **we can override a var arg method with another var -arg method only,**
- **If we are trying to override with a normal method then it will become overloading but not overriding.**

- **If the child class method is also var arg then it will become overriding.**

- **Overriding concept is not applicable for variables and it is applicable only for methods, variable resolution always takes care by compiler based on reference type.**

- **This rule is the same whether the variable is static or non-static .**

Note :

- **In overloading we have to check only method names (must be same) and argument types (must be diff).**
- **The remaining things like the return type we are not required to check.**
- **But overriding we have to check everything like method names, argument type etc..**

Polymorphism :

- **Same name but multiple forms is the concept of polymorphism.**

```
Ex:
abs(int i)
abs(long l)
abs(double d)
```

- **We can use parent reference to hold any child class of object.**

```
List l = new ArrayList();
new LinkedList();
new vector();
new Stack();
```

- **Polymorphism → static /compile time / early binding → overloading & method hiding.**
- **Inheritance → Dynamic / runtime / late binding. → overriding**

****End Of Oops****

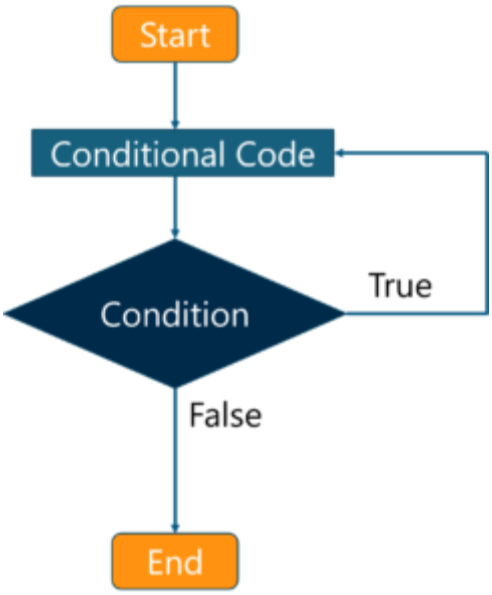
Control Statements or Logical Statements

- **In Java Control Statements is one of the fundamentals required for Java Programming.**
- **It allows the smooth flow of a program.**

Following pointers will be covered in this article:

- **Decision Making Statements**
- **Simple if statement**
- **if-else statement**
- **Nested if statement**
- **Switch statement**
- **Looping statements**
- **While**
- **Do-while**
- **For**
- **For-Each**
- **Branching statements**
- **Break**
- **Continue**

- **Every programmer is familiar with the term statement,**
- **which can simply be defined as an instruction given to the computer to perform specific operations.**
- **A control statement in java is a statement that determines whether the other statements will be executed or not.**
- **It controls the flow of a program.**
- **An 'if' statement in java determines the sequence of execution between a set of two statements.**
- **Control Statements can be divided into three categories, namely**
- **Selection statements**
- **Iteration statements**
- **Jump statements**
- **Moving on with this article on Control Statements in Java**



Decision-Making Statements

- **Statements that determine which statement to execute and when are known as decision-making statements.**
- **The flow of the execution of the program is controlled by the control flow statement.**
- **There are four decision-making statements available in java.**
- **Moving on with this article on Control Statements in Java**

Simple if statement

- **The if statement determines whether a code should be executed based on the specified condition.**

Syntax:

```
if (condition) {
Statement 1; //executed if condition is true
}
Statement 2; //executed irrespective of the condition
```

If..else statement

- **In this statement, if the condition specified is true, the if block is executed.**
- **Otherwise, the else block is executed.**

Example:

```
public class Main
{
public static void main(String args[])
{
int a = 15;
if (a > 20)
System.out.println("a is greater than 10");
else
System.out.println("a is less than 10");
System.out.println("Hello World!");
}
}
}
```

Output:

a is less than 10

Hello World!

Nested if statement

- **An if present inside an if block is known as a nested if block.**
- **It is similar to an if..else statement, except they are defined inside another if..else statement.**

Syntax:

```
if (condition1) {

Statement 1; //executed if first condition is true

if (condition2) {

Statement 2; //executed if second condition is true

}

else {

Statement 3; //executed if second condition is false

}

}
```

Example

```
public class Main
{
public static void main(String args[])
{
int s = 18;
if (s > 10)
{
if (s%2==0)
System.out.println("s is an even number and greater than 10!");
else
System.out.println("s is a odd number and greater than 10!");
}
else
{
System.out.println("s is less than 10");
}
System.out.println("Hello World!");
}
}
```

Output:

s is an even number and greater than 10!

Switch statement

- **A switch statement in java is used to execute a single statement from multiple conditions.**
- **The switch statement can be used with short, byte, int, long, enum types, etc.**
- **Certain points must be noted while using the switch statement:**
- **One or N number of case values can be specified for a switch expression.**
- **Case values that are duplicate are not permissible.**
- **A compile-time error is generated by the compiler if unique values are not used.**
- **The case value must be literal or constant.**
- **Variables are not permissible.**
- **Usage of break statement is made to terminate the statement sequence.**
- **It is optional to use this statement. If this statement is not specified, the next case is executed.**

Example:

```
public class Music {
public static void main(String[] args)
{
int instrument = 4;
String musicInstrument;
// switch statement with int data type
switch (instrument) {
case 1:
musicInstrument = "Guitar";
break;
case 2:
music Instrument = "Piano";
break;
case 3:
musicInstrument = "Drums";
break;
case 4:
musicInstrument = "Flute";
break;
case 5:
musicInstrument = "Ukulele";
break;
case 6:
```

```
        musicInstrument = "Violin";
        break;
        case 7:
        musicInstrument = "Trumpet";
        break;
        default:
        musicInstrument = "Invalid";
        break;
        }
        System.out.println(musicInstrument);
        }
        }
```

Output:

Flute

Looping Statements

- **Statements that execute a block of code repeatedly until a specified condition is met are known as looping statements.**
- **Java provides the user with three types of loops:**

- **While**
- **Known as the most common loop, the while loop evaluates a certain condition.**
- **If the condition is true, the code is executed.**
- **This process is continued until the specified condition turns out to be false.**
- **The condition to be specified in the while loop must be a Boolean expression.**
- **An error will be generated if the type used is int or a string.**

Syntax

```
while (condition)
{
statementOne;
}
```

Example:

```
public class whileTest
{
public static void main(String args[])
{
int i = 5;
while (i <= 15)
{
System.out.println(i);
i = i+2;
}
}
}
```

Output:

5

7

9

11

13

15

Do..while

- **The do-while loop is similar to the while loop, the only difference being that the condition in the do-while loop is evaluated after the execution of the loop body.**
- **This guarantees that the loop is executed at least once.**

Syntax:

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

Example:

```
public class Main
{
public static void main(String args[])
{
int i = 20;
do
{
System.out.println(i);
i = i+1;
} while (i <= 20);
}
}
```

Output:

20

For-Each

- **The traversal of elements in an array can be done by the for-each loop.**
- **The elements present in the array are returned one by one.**
- **It must be noted that the user does not have to increment the value in the for-each loop.**

Example:

```
public class foreachLoop{

public static void main(String args[]){

int s[] = {18,25,28,29,30};

for (int i : s) {

System.out.println(i);

}

}

}
```

Output:

18

25

28

29

30

For

- **The for loop in java is used to iterate and evaluate a code multiple times.**

- **When the number of iterations is known by the user, it is recommended to use the for loop.**

Syntax

```
for (initialization; condition; increment/decrement)
{
statement;
}
```

Example

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

Output:

5

6

7

8

9

10

Branching Statements

- **Branching statements in java are used to jump from a statement to another statement, thereby transferring the flow of execution.**
- **Moving on with this article on Control Statements in Java**

Break

- **The break statement in java is used to terminate a loop and break the current flow of the program.**

Example:

```
public class Test

{

public static void main(String args[])

{

for (int i = 5; i < 10; i++)

{

if (i == 8)

break;

System.out.println(i);

}

}

}
```

Output:

5

6

7

Continue

- **To jump to the next iteration of the loop, we make use of the continue statement.**

- **This statement continues the current flow of the program and skips a part of the code at the specified condition.**

Example

```
public class Main{

public static void main(String args[]){

for (int k = 5; k < 15; k++){

// Odd numbers are skipped

if (k%2 != 0)

continue;

// Even numbers are printed

System.out.print(k + " ");

}

}

}
```

- **With this, we come to the end of this Control Statements in Java Article. The control statements in java must be used efficiently to make the program effective and user-friendly.**

Operators

What is Operator

1. Operators in Java are the symbols used for performing specific operations. For example: +, -, *, / etc.

Types of Operators in Java

1. Arithmetic Operators
2. Unary Operator
3. Relational Operators

4. Logical Operators
5. Ternary Operator
6. Bitwise Operators
7. Shift Operators
8. Assignment Operators
9. instanceof operators

Operator Type	Category	Precedence
Unary	Postfix prefix	Expr++expr- ++expr--expr+expr-expr~!
Arithmetic	Multiplicative Additive shift	*/% +- << >> >>>
Relational	Comparison equality	< > <= >= instanceof == !=
Bitwise	bitwiseAND bitwiseexclusiveOR bitwiseinclusiveOR	& ^ |
Logical	logicalAND logicalOR	&& ||
Ternary	ternary	?:
Assignment	assignment	=+=-=*=/=%=&=^=|=<<=>>=>>>=

1) Arithmetic Operators

These operators involve the mathematical operators that can be used to perform various simple or advanced arithmetic operations on the primitive data types referred to as the

operands.

Program :

```
int a = 10;
int b = 5;
System.out.println("a + b = " + (a + b));
System.out.println("a - b = " + (a - b));
System.out.println("a * b = " + (a * b));
System.out.println("a / b = " + (a / b));
System.out.println("a % b = " + (a % b));
```

Output :

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
```

Operators	**Result**
+	Addition of two numbers
-	Subtraction of Two numbers
*	Multiplication Of Two numbers
/	Division Of Two numbers
%	Modulus Operator Divides Two numbers and returns the remainder.

Example 1 :

String s = “prasad”;

Int a = 10 , b = 20, c = 30;
1. a = a+b+c;
2. a = s+a;
3. a = s+ a + b ;
4. s = a +b+c;

Example 2 :

Rule : max(int , type of a , type of b) - int
byte a = 10 ;
byte b = 10 ;
byte c = a +b ; --- compile time error
byte c = (byte) (a +b) - works fine
byte b = 10 ;
byte c = b+1; // compile time error
'a' + 'b' -- >195 (int)
'a' + 0.5 -- > 95.5 (double)

In integral arithmetic byte , short , int long there is no way to represent infinity hence infinity is a result we will get arithmetic exception in integral arithmetic.
ex :
 s.o.p(10/0); // run time exception
s.o.p(10/0.0) - infinity
 • But in floating point arithmetic (float and double) there is a way to represent infinity.
float and double classes contains below constant
 POSITIVE_INFINITY and NEGATIVE_INFINITY

 s.o.p(-10/0.0) --infinity
s.o.p(0/0) - exception
s.o.p(0.0/0) - Nan (not a number)

2. Unary Operators

1.–:Unary minus : This operator can be used to convert a positive value to a negative one

2. + : Unary plus : Indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.

3. 'NOT' Operator(!) : This is used to convert true to false or vice versa. Basically, it reverses the logical state of an operand.

4. ++ :Increment operator : Used for incrementing the value by 1. There are two varieties of increment operators.

1. Post-Increment: Value is first used for computing the result and then incremented.
2. Pre-Increment: Value is incremented first, and then the result is computed.

5.— : Decrement operator : Used for decrementing the value by 1. There are two varieties of decrement operators.

1. Post-decrement: Value is first used for computing the result and then decremented.
2. Pre-Decrement: The value is decremented first, and then the result is computed

Example 1 :

```
int x=10;
Int y = -x;
 System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
System.out.println(y);//-10
```

Example 2:

```
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
```

3. Assignment Operators

1. These operators are used to assign values to a variable.

2. The left side operand of the assignment operator is a variable, and the right side operand of the assignment operator is a value.

3. The value on the right side must be of the same data type of the operand on the left side. Otherwise, the compiler will raise an error.

4. This means that the assignment operators have right to left associativity, i.e., the value given on the right-hand side of the operator is assigned to the variable on the left.

5. Therefore, the right-hand side value must be declared before using it or should be a constant.

• **Types of Assignment Operators in Java**
The Assignment Operator is generally of two types. They are:
1. Simple Assignment Operator:
• The Simple Assignment Operator is used with the "=" sign where the The left side consists of the operand and the right side consists of a value. The value of the right side must be of the same data type that has been defined on the left side.

2. Compound Assignment Operator:
• The Compound Operator is used where +,-,*, and / is used along with the = operator
a += 10
This means,
a = a + 10

Note: The compound assignment operator in Java performs **implicit type casting.**

Let's consider a scenario where x is an int variable with a value of 5.

```
int x = 5;
```

If you want to add the double value 4.5 to the integer variable x and print its value, there are two methods to achieve this:

```
method 1: x = x + 4.5
method 2: x += 4.5
```

As per the previous example, you might think both of them are equal.

But in reality, method 1 will throw a compiletimr error stating the "incompatible types: possible lossy conversion from double to int", Method 2 will run without any error and prints 9 as output.

Reason for the Above Calculation

• Method 1 will result in a compiletime error stating "incompatible types: possible lossy conversion from double to int."
The reason is that the addition of an int and a double results in a double value.
Assigning this double value back to the int variable x requires an explicit type casting because it may result in a loss of precision.
Without the explicit cast, the compiler throws an error.

• Method 2 will run without any error and print the value 9 as output.
The **compound assignment operator += performs an implicit type conversion**, also known as an **automatic narrowing primitive conversion** from double to int.

It is equivalent to x = (int) (x + 4.5), where the result of the addition is explicitly cast to

an int.

The fractional part of the double value is truncated, and the resulting int value is assigned back to x.

• It is advisable to use Method 2 (x += 4.5) to avoid runtime errors and to obtain the desired output.

• Same automatic narrowing primitive conversion is applicable for other compound assignment operators as well, including -=, *=, /=, and %=

4. Relational Operators

• These operators are used to check for relations like equality, greater than, and less than.
• They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements.

1. ==, Equal to returns true if the left-hand side is equal to the right-hand side.
2. !=, Not Equal to returns true if the left-hand side is not equal to the right-hand side.
3. <, less than: returns true if the left-hand side is less than the right-hand side.
4. <=, less than or equal to returns true if the left-hand side is less than or equal to the right-hand side.
5. >, Greater than: returns true if the left-hand side is greater than the right-hand side.
6. >=, Greater than or equal to returns true if the left-hand side is greater than or equal to the right-hand side

Example :
int a = 10;

```
int b = 3;
int c = 5;
System.out.println("a > b: " + (a > b));     // true
System.out.println("a < b: " + (a < b));    // false
System.out.println("a >= b: " + (a >= b)); // true
System.out.println("a <= b: " + (a <= b)); // false
System.out.println("a == c: " + (a == c)); // false
System.out.println("a != c: " + (a != c));  // true
System.out.println( 'a'== 'b')  // false
System.out.println('a' == 97.0);  // true
```

5. Logical operators

• Logical operators are used to perform logical "AND", "OR" and "NOT" operations, i.e. the function similar to AND gate and OR gate in digital electronics.
• They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition under particular consideration.
• One thing to keep in mind is, while using AND operator, the second condition is not evaluated if the first one is false.
• Whereas while using OR operator, the second condition is not evaluated if the first one is true
1. ANDOperator (&&)– if(a && b) [if true execute else don't]
2.OR Operator (||)– if(a || b) [if one of them is true to execute else don't]
3.NOT Operator (!)– !(a<b) [returns false if a is smaller than b}

6. Shift Operators

•The shift operator is a java operator that is used to shift bit patterns right or left

Name of operator	Sign	Description
Signed Left Shift	<<	The left shift operator moves all bits by a given number of bits to the left.
Signed Right Shift	>>	The right shift operator moves all bits by a given number of bits to the right.

Left Shift Operator

1. The left shift means that each of the bits is in binary representation toward the left.
2. Left shifting a number by certain positions is equivalent to multiplying the number by two raised to the power of the specified positions.
i.e left shift x by n positions <=> x * 2n

Examples : 1. System.out.println(30 << 2); // 120

Right Shiff Operator

• The Right Shift Operator moves the bits of a number in a given number of places to the right. The >> sign represents the right shift operator.

1. System.out.println(30 >> 2); // 7

7 Bitwise Operator

• These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types.

1. &, Bitwise AND operator:
• returns bit by bit AND of input values.
2. |, Bitwise OR operator:
• returns bit by bit OR of input values.
3. ^, Bitwise XOR operator:

• returns bit-by-bit XOR of input values.

Example :

```
System.out.println( 4 & 5 ) ;   4
```

Binary Representaion:

```
4  --1 0 0
5   --1 0 1
-------------------------------
4 & 5   --1 0 0
```

Example 2 : | operator

```
System.out.println( 4 | 5 ) ;   5
```

Binary Representaion:

```
4  --1 0 0
5   --1 0 1
-------------------------------
4  | 5   --1 0 1
```

Example 3 : ^ operator

```
System.out.println( 4 ^ 5 ) ;   1
```

Binary Representaion:

```
4  --1 0 0
5   --1 0 1
-------------------------------
4  | 5   --0 0 1
```

Example 4 :

```
int a = 10;
 int b = 20;
 System.out.println( a>5 & b <30) : // true
System.out.println( a>5 | ++b <30) : // true
 System.out.println(b); // 21
```

Note : incase of | operator both conditions will get executed even the the first condition is true. But incase of || operator if first condition is true the second condition will not execute.

8. Ternary Operator

• Java Ternary operator is used as one line replacement for if-then else statement and used a lot in Java programming.
It is the only
conditional operator which takes three operands.
Syntax : condition ? statement1: statement2 ;

• If the condition is true, then execute the statements after the '?' i.e statement1. else execute the statements after the ':' i.e statement2.

```
int a=2;
 int b=5;
 int min=(a<b)?a:b;
 System.out.println(min); // 2
```

9. InstanceOf Operator

• The instanceof the operator is used for type checking.

It can be used to test if an object is an instance of a class, a subclass, or an interface

```
 class Test{
 public static void main(String args[]){
 Test t=new Test();
 System.out.println(t instanceof Test);//true
 }
 }
```

****End Of Control Statements****

Exception Handling

Introduction

- **An unwanted event that disturbs the normal flow of the program is called Exception.**
- **It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of a program.**

Run Time Stack Mechanism

- **For every thread JVM will create a stack and every method call performed by that thread will be stored in the corresponding stack.**
- **Each entry in the stack is called stack frame or activation frame.**
- **After every method call, the corresponding entry from the stack will be removed after completing all the method calls, the stack will become empty and it will be destroyed by JVM just before terminating the thread.**

Default Exception Handling in JAVA :

- **Inside a method if an exception occurs, the method in which it is raised is responsible to create Exceptional objects by including the following information.**
- **Name of exception**
- **Description of exception**
- **Location at which an exception occurs.**

1. **After creating an exception object, method handovers that object to the JVM.**
2. **JVM will check whether the method contains any exception handling code or not.**

- **If the method does not contain exception handling code, the JVM terminates that method abnormally and removes corresponding entry from the stack.**

3. **The JVM identifies caller method and checks whether caller method has any handling code or not.**
4. **If the caller method does not contain handling code, JVM terminates that caller method abnormally and removes corresponding entry from stack.**

- **This process continues till the main method. If the main method also does not contain handling code, terminates the main method abnormally and removes the corresponding entry from the stack.**

- **Then, JVM handovers responsibility of exception information in the following format and the program is terminated abnormally.**

"Exception in Thread Main "XXX" : Name of Exception : Description Stack Trace " :

Eg:

```
public class TestExcep3 {
        public static void main(String args[]) {
        }
        public static void dostuff() {
                domoreStuff();
        }
        public static void domoreStuff() {
                System.out.println(10 / 0);
        }
}
```

O/P : Exception in Thread "Main" java.lang.Arithmeticexception :/by Zero

Note:

- **In a program if all methods terminate normally then only program termination is normal.**

- **If one method terminates abnormally, the program termination is abnormal termination.**

Exception Hierarchy :

- **The Throwable class acts as the root class for Java Exception Hierarchy.**
- **Throwable class defines two child classes.**

1. **Exception**
2. **Error**

Exception:

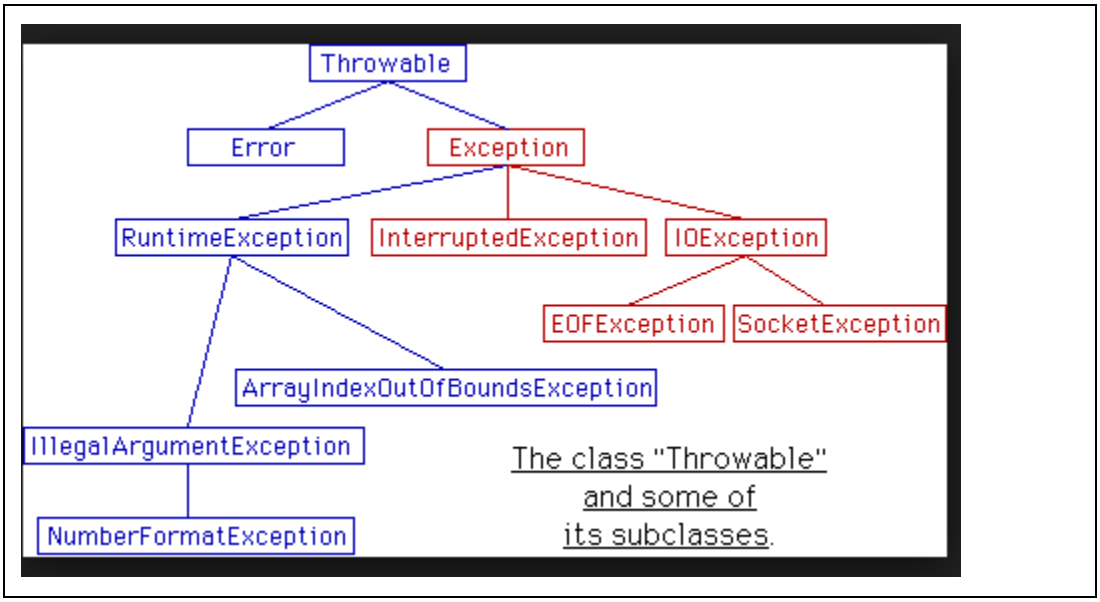
- **Most of the times Exceptions are caused by our program and these are recoverable.**

  Eg . FileNotFoundException , ArithmeticException

Error :

- **Most of the time errors are caused due to lack of system resources.**

  Eg : OutOfMemoryError.



Checked and Unchecked Exceptions

Checked Exception:

- **The Exceptions which are checked by the compiler at runtime for smooth execution of the program are called checked exceptions.**

  Eg:
  FileNotFoundException

IOException

- **In our program if there is a chance of raising a checked exception, then we should handle that checked exception (either by try catch or throw keyword) otherwise we will get a compile time error.**

Unchecked Exception :

- **The Exceptions which are not checked by the compiler whether the program is handling them or not are called unchecked exceptions.**

Eg :
ArithmeticException
NullPointerException

Note :

- **Whether it is checked or unchecked every exception occurs at runtime only.**
- **There is no chance of occurring any exception at compile time.**
- **RuntimeException and its child classes, Error and its child classes are unchecked exceptions. Remaining all are checked exceptions.**

Fully Checked Vs Partially Checked Exceptions :

- **A checked exception is said to be fully checked if all its child classes are also checked.**

Eg :
IOException.
InterruptedException.

- **A checked exception is said to be partially checked if some of its child classes are unchecked.**

Eg:
Exception
Throwable.

Note :

- **The only possible partially checked exceptions in JAVA are Exception,Throwable.**

Eg :

- **IOException : Checked - Fully**
- **RuntimeException : Unchecked**
- **InterruptedException : Checked - Fully**
- **Error : Unchecked**
- **Throwable : Checked : Partially**
- **ArithmeticException : Unchecked**
- **NullPointerException : Unchecked**
- **Exception : Checked : Partially**
- **FileNotFoundException : Checked : Fully**

Customized Exception Handling by Using Try- Catch :

- **It is highly recommended to handle exceptions.**
- **The code which raises an exception is called risky code and we have to define that code inside the try block and corresponding handling code should be defined in the catch block.**

```
try
{
   risky code;
}
catch (Exception e)
{
   handling code;
}
```

Control Flow in try-catch :

```
try
{
  Stmt 1;
  Stmt 2;
  Stmt 3;
}
catch (Exception e)
{
  Stmt 4;
}
  Stmt 5;
```

Case 1:

- **If there is no exception, catch block will not be executed**

Hence stmts 1,2,3,5 will be executed and will be a normal termination.

Case 2:

- **If there is an exception at stmt 2, and corresponding catch block matches, then flow is as follows. 1,4,5 and is normal termination.**

Case 3 :

- **If an exception is raised at statement 2 and the corresponding catch block is not matched, the flow is as stmt 1, abnormal termination.**

Case 4 :

- **If an exception is raised at stmt 4 (in catch block) or stmt 5, then it is always abnormal termination.**

Note:

- **Within the try block, if anywhere an exception is raised, then the rest of the try block will not be executed, even though we handled that exception.**
- **Hence, within the try block only have to take only risky code and the length of the try block should be as less as possible.**
- **In addition to try blocks, there may be a chance of raising an exception inside catch and finally blocks.**

- **If any statement which is not part of the try block and raises an exception, then it's always abnormal termination.**

Methods to print exception information

- **The Throwable class defines the following methods to print exception information.**

- **Method: printStackTrace()**
- **Printable Format: Name of Exception: Description : StackTrace**

- **Method: toString()**
- **Printable Format: Name of Exception : Description**

- **Method : getMessage()**
- **Printable Format : Description**

Example :

```
public class TestExp5
{
 public static void main(String args[])
 {
  try
  {
   System.out.println(10/0);
  }
  catch(ArithmeticException e)
  {
   e.printStackTrace();
   System.out.println(e);
   System.out.println(e.toString());
   System.out.println(e.getMessage());
  }
 }
```

NOTE :

- **Internally default exception handlers will use the printStackTrace() method to print exception information to the console.**
- **try with multiple catch blocks**
- **The way of handling an exception varies from exception to exception, hence, for every exception type, it is highly recommended to take separate catch blocks.**
- **i.e, try with multiple catch blocks is always possible and recommended to use.**

EX:1

```
try
    {
     Risky code
    }
    catch (Exception E)
    {
     Handle exception
    }
```

BAD PROGRAMMING PRACTICE

EX:2

```
try
    {
     Risky code
     St1
     St2
     St3
    }
    catch(ArithmeticException e)
    {
     Handling code
    }
    catch(SQLException e)
    {
     Handling code
    }
    catch(FileNotFoundException e)
    {
      //Use a local file instead of a remote file.
    }
```

Note :

- **If trying with multiple catch blocks is used, then order of catch blocks is very important.**
- **We have to take the child first and then the parent otherwise we will get a compile time error.**
- **"Unreachable catch block for NullPointerException. It is already handled by the catch block for Exception"**

```
EX:1

try
{
   S.O.P(10/0);
}
catch (Exception e)
{ }
catch(ArithmeticException e)
{ }
o/p :

 CTE : Exception /0 has already been caught
```

Note:

- **We cannot declare two catch blocks for the same exception, otherwise we will get a compile time error.**

```
try
{
Risky code
}
catch(Exception e)
{
Handling code
}
```

```
catch(Exception e)
{
Handling code
}
```

CTE: Exception: java.lang.ArithmeticException has already been caught

Finally Block :

- **It is not recommended to maintain clean up code in try catch blocks.**
- **We require some place to maintain cleanup code which should be executed always irrespective of whether an exception is raised or not, raised and handled, or not handled.**
- **For this purpose finally a block is used.**
- **Hence the main objective of finally block is to maintain clean up code.**

```
try
{
  try
  {

  }Catch
   {

   }
}
Catch
{

}
finally
{
  Clean up code
}
```

Example :

```
public class Test
{
```

```
public static void main(String args[])
{
try
{
  System.out.println(10/0);
}
catch(Exception e)
{
  System.out.println("catch");
}
  finally
  {
   System.out.println("finally");
  }
}
}
```

Example :

```
public class Test
{
public static void main(String args[])
{
try
{
  System.out.println("try");
  System.out.println(10/0);
}
catch(Exception e)
{
  System.out.println("catch");
}
  finally
  {
   System.out.println("finally");
  }
}
}
```

Example :

```
public class Test
{
public static void main(String args[])
{
try
{
  System.out.println("try");
  System.out.println(10/0);
}
catch(NullPointerException e)
{
  System.out.println("catch");
}
  finally
  {
   System.out.println("finally");
  }
}
}
```

finally vs return :

- **Even Though return statement is present inside try or catch,**

first finally block will be executed and after that only return statement will be considered,

ie. finally block dominates the return statement.

```
public class Test
{
public static void main(String args[])
{
try
{
  System.out.println("try");
  return;
```

```
}
catch(NullPointerException e)
{
  System.out.println("catch");
  return;
}
  finally
  {
   System.out.println("finally");
  }
}
}
```

Note :

- **If return is present inside the “try catch finally” block return statement will be considered.**

```
public class Test
{
public static void main(String args[])
{
  System.out.println(m1());
}
static public int m1()
{
try
{
  return 777;
}
catch(NullPointerException e)
{
  System.out.println("catch");
  return 888;
}
  finally
  {
   return 999;
  }
}
}
```

Finally vs system.exit(0) :

- **There is one situation where finally block will not be executed ie whenever we are using System.exit(0) ;**
- **Then JVM itself will shutdown in this particular case and finally the block will not be executed. Ie. system.exit(0) dominates the finally block.**

Example

```
public class Test
{
public static void main(String args[])
{
try
{
  System.out.println("try");
  System.exit(0);
}
catch(ArithmeticException e)
{
  System.out.println("catch");
  return;
}
  finally
  {
   System.out.println("finally");
  }
}
}
```

→ System.exit(0) :

- **Any int value is allowed.**
- **0(zero) means normal termination.Non zero means abnormal termination.**
- **Argument represents status code and it is internally used by JVM.**

- **Whether the argument is zero or nonzero, the result is the same with respect to program/programmer.**

Final vs finally vs finalize

final :

- **final is the modifier applicable for classes, methods and variables.**
- **If a class is declared as final we cannot extend that class.**
- **If a method is final then, we cannot override that method.**
- **If a variable is declared as final, we cannot perform reassignment to it.**

finally :

- **finally is a block always associated with try-catch to maintain clean up code.**
- **The speciality of finally block will be executed always irrespective of whether an exception is raised or not raised and whether handled or not handled.**

finalize() :

- **finalize is a method always executed by a garbage collector just before destroying an object to perform cleanup activities.**
- **once the finalize() method completes its execution immediately the garbage collector destroys that object.**

Note :

- **finally block is responsible to perform cleanup activities related to try block i.e whatever resources we opened as part of try block will be closed inside finally block.**
- **Whereas the finalize method is responsible to perform cleanup activities related to the object i.e whatever resources associated with the object deallocated before destroying an object by using the finalize method.**

Possible combinations of try-catch-finally :

Case 1 :

```
try
{

}

catch(E e){

}

//Valid
```

Case 2:

```
try
{

}
catch(X e){

}
catch (Y e)
{

}
//Valid
```

Case 3:

```
try
{ }
catch(ClassNotFoundException e)
{ }
catch (ClassNotFoundException e)
```

```
{  }

Compile time Error :
Unreachable catch block for ClassNotFoundException.
It is already handled by the catch block for ClassNotFoundException

//invalid
```

```
Case 5:

try
{ }
finally
{ }

//Valid
```

```
Case 6:

try
{
}
catch(X e)
{
}
try
{
}
//invalid
```

```
Case 7:

try
{

BlockStatements

}
```

```
//invalid
//Compile time Error: Syntax error, insert "Finally" to complete
```

Case 8:

```
catch (E e)
{
}

//Compile time Error:
//CE: Syntax error, insert "Statement" to complete //BlockStatements
```

Case 9:

```
finally
{

}
//Compile time Error: Syntax error on token "finally", this expected
```

Case 10:

```
try
{
}
finally
{
}
catch(E e)
{
}
//Compile time Error : catch without try block
```

Case 11:

```
try
{
}
System.out.println("Hello");
catch(E e)
{
}

//Compile time Error : try without catch or finally
```

Case 12:

```
try
{
}
catch(X e)
{
}
System.out.println("Hello");
Catch(Y e)
{
}


// Compile time Error : catch without try.
```

Case 13:

```
try
{
}
Catch(X e)
{
}
System.out.println(“Hello”);
finally
```

```
{
}

//Compile time Error : finally without try
```

Case 14:

```
try
{
  try
  {
  } catch(E e)
  {
  }
}
catch(E e)
{
}

//Valid
```

Case 15:

```
try
{
  try
  {
  }
}
catch(E e)
{
}

//Compile time Error : try without catch for child
```

Case 16:

```
try
```

```
{
  try
  { }
  finally
  {  }
}
catch(E e)
{
}

//Valid
```

Case 17:

```
try
{
}
catch(E e)
{
        try
        { }
        finally
        { }
}

 // Valid
```

Case 18:

```
try
{
}
catch(E e)
{
sop();
 finally
 { }
}
```

```
//CE: finally without try
```

Case 19:

```
try
{
}
catch(E e)
{
}
finally
{
  try
  { }
  catch(Ex e)
  { }
}

//Valid
```

Case 20 :

```
try
{
}
catch(E e)
{
}
finally
{
  finally
  { }
}

//Compile time Error : finally without try
```

Case 21: Invalid

```
try
{ }
catch(E e)
{ }
finally
{ }
finally
{ }
```

Case 22 :

```
try
  System.out.println("try")
catch(E e)
{
  System.out.println("Catch")
}
finally
{
}

//Invalid
```

Case 23 :

```
try
{
}
catch(E e)
  System.out.println("Catch");
finally
{
}

//Invalid
```

Case 24 :

```
try
{
}
catch(E e)
{
}
finally
  System.out.println("finally");

//Invalid
```

Note :

- **In try-catch-finally order is important.**
- **Whenever we are writing try, we should have either catch or finally blocks otherwise we will get a compile time error.**
- **Whenever we are writing a catch block, compulsory try block must be required,ie, catch without try is invalid.**
- **Whenever we are writing a finally block, we should write try block, otherwise we will get a compile time error.**
- **Inside try, catch, finally we can declare try-catch-finally blocks ie. nesting of try catches finally is allowed.**
- **For try -catch- finally blocks curly braces are mandatory.**

throw

- **Sometimes we can create Exception objects explicitly and hand them over to JVM manually.**
- **For this we have to use the throw keyword.**
  throw new ArithmeticException("\ by zero");
- **Hence, the main objective of the throw keyword is to handover our created exception object to the JVM manually.**

Example:

```
public class Test
{
  public static void main(String args[])
  {
    System.out.println(10/0);
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:5)
```

Example :

```
public class Test
{
  public static void main(String args[])
  {
    throw new ArithmeticException("/by Zero");
 }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException: /by Zero
        at Test.main(Test.java:5)
```

The result of the two programs is exactly the same.

Note :

- **Best use of the "throw" keyword is for *user defined exceptions or customized exceptions.***

Case 1 :

```
public class Test
{
  static ArithmeticException e= new ArithmeticException();
  public static void main(String Args[])
```

```
 {
  throw e;   //
 }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException
        at Test(Test.java:3)
```

Note :

throw e if e refers null then we get NullPointerException.

```
public class Test
{
 static ArithmeticException e;
public static void main(String Args[])
 {
  throw e;
 }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException at Test.main(Test.java:6)
```

Case 2 :

```
public class Test
{
 public static void main(String args[])
 {
  System.out.println(10/0);
  System.out.println("Hello");
 }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero    at Test.main(Test.java:5)
```

Example :

```
public class Test
```

```
{
  public static void main(String args[])
  {
    throw new ArithmeticException("/by Zero");
    System.out.println("Hello");
  }
}
```

Compile time error:

```
Test.java:6: error: unreachable statement
    System.out.println("Hello");
    ^
1 error
```

Note :

- **After throw statement , we are not allowed to write any statement directly, otherwise we will get compile time error saying “unreachable statement”**

Case 3 :

```
public class Test
{
  public static void main(String args[])
  {
    throw new Test();
  }
}
```

Compile time error.

```
/tmp/java_H1vHBx/Test.java:5: error: incompatible types: Test cannot be converted to Throwable
    throw new Test();
    ^
1 error
```

Example :

```
public class Test extends RuntimeException
{
  public static void main(String args[])
```

```
  {
    throw new Test();
  }
}
```

Output;

- **Exception in thread "main" Test at Test.main(Test.java:5)**

Note :

- **We can use the “throw” keyword only for “Throwable” types.**

If we are trying to use normal java objects we will get a compile time error saying “incompatible types”.

throws :

→ In our program if there is a chance of raising a checked exception then we should handle that checked exception otherwise we will get a compile time error saying “unreported ExceptionXXX must be caught or declared to be thrown”.

Example :

```
import java.io.*;
public class Test
{
  public static void main(String args[]) throws Exception
  {
    PrintWriter pw = new PrintWriter("abc.txt");
    pw.println("Hello");
  }
}
```

Compile time error

```
/tmp/java_hDoRg2/Test.java:6: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
    PrintWriter pw = new PrintWriter("abc.txt");
                     ^
```

1 error

Example 2 :

```
public class Test
{
  public static void main(String args[])
  {
   Thread.sleep(1000);
  }
}
```

Compile time error

```
/tmp/java_JsJpEL/Test.java:5: error: unreported exception InterruptedException; must be caught or declared to be thrown
   Thread.sleep(1000);
             ^
1 error
```

→ we can handle this compile time error by using the following two ways.

By not using try-catch blocks

```
public class Test
{
  public static void main(String args[])
  {
    try
    {
     Thread.sleep(100);
    }
    catch(InterruptedException e)
    {
      //System.out.println("using try-catch");
    }
  }
}
```

By using throws keyword

- **We can use throws to delegate responsibility of execution of exception handling to the caller(it may be another method or JVM) then the caller method is responsible to handle that exception.**

Example :

```
public class Test
{
  public static void main(String args[]) throws InterruptedException
  {
   Thread.sleep(1);
  }
}
```

Conclusions :

- **We can use throws to delegate responsibility of exception handling to the caller.(it may be a method or JVM).**
- **It is required only for checked exceptions and usage of throws for unchecked exceptions has no impact.**
- **It is required only to convince the compiler and usage of throws does not prevent abnormal termination of program.**

Note:

- **It is recommended to use try-catch instead of throwing a keyword.**

Case 1:

- **We can use throws keywords for methods and constructors but not for classes.**

```
public class Test
{
   Test() throws Exception
   {
   }
   public void m1() throws Exception
   {
   }
}
```

Case 2:

- **We can use throws keywords only for throwable types. If we are trying to use for normal java classes then we get compile time error saying "incompatible types"**

Example 1:

```
public class Test
{
  public void m1() throws Test
  {                              invalid : incompatible types compile time error
  }
}
```

Example 2 :

```
public class Test extends RuntimeException
{
  public void m1() throws Test            VALID
  {
  }
}
```

Case 3:

```
public class Test
{
 public static void main(String args[])
 {
   throw new Exception();
 }
}
```

Compile time error :

```
Test.java:5: error: unreported exception Exception; must be caught or declared to be thrown
   throw new Exception();
   ^
1 error
```

Example :

```
public class Test
{
  public static void main(String args[])
  {
    throw new Error();
  }
}
```

Runtime error:

```
Exception in thread "main" java.lang.Error
        at Test.main(Test.java:5)
```

Case 4:

Example:

```
public class Test
{
  public static void main(String args[])
  {
    try
    {
      System.out.println("Hello");
    }
    catch(ArithmeticException e)
    {
    }
  }
}
```

Example :

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

```
    catch(Exception e)
    {
    }
  }
}
```

Example :

```
import java.io.*;
public class Test
{
  public static void main(String args[])
  {
    try
    {
      System.out.println("Hello");
    }
    catch(IOException e)
    {
    }
  }
}
```

Compile time error :

```
Test.java:10: error: exception IOException is never thrown in body of corresponding try statement
    catch(IOException e)
    ^
1 error
```

Example :

```
import java.io.*;
public class Test
{
  public static void main(String args[])
  {
    try
    {
      System.out.println("Hello");
    }
    catch(InterruptedException e)
```

```
    {
    }
  }
}
```

Example:

```
import java.io.*;
public class Test
{
  public static void main(String args[])
  {
    try
    {
      System.out.println("Hello");
    }
    catch(Error e)
    {
    }
  }
}
```

Note :

- **If within the try block if there is no chance of raising an exception then we cannot write catch block for that exception, otherwise we will get compile time error saying "ExceptionXXX is never thrown in the body of try"**

But, this rule is applicable only for fully checked exceptions.

Exception handling keywords summary:

try → to maintain risky code
catch → to maintain handling code
finally → to maintain clean up code
throw → to handover our created exception object to JVM manually.
throws → to delegate responsibilities of exception handling to the caller.

- **Various possible compile time errors in exception handling.**

1. **Unreported exception xxx, must be caught or declared to be thrown**

2. **Exception xxx has already been caught.**
3. **Exception xxx is never thrown in body of corresponding try**
4. **Unreachable statement**
5. **Incompatible types**
6. **Try without catch**
7. **Catch without try**
8. **Finally without try**

- **Customized or user defined exceptions.**
- **Sometimes to meet programming requirements we can define our own exceptions.**
- **Such types of exceptions are called customized or user defined exceptions.**

Note :

- **throw keyword is best suitable for user defined or customized exceptions but not for predefined exceptions.**
- **It is highly recommended to define custom exceptions as unchecked i.e. we have to extend RuntimeException but not Exception.**

Top 10 Exceptions :

- **Based on the person who is raising an exception all exceptions are divided into two categories.**

1. **JVM exceptions**
2. **Programmatic exceptions**

JVM Exceptions

- **The exceptions which are raised automatically by JVM whenever a particular event occurs are called JVM exceptions.**

Eg:
ArithmeticException
NullPointerException

Programmatic Exceptions

- **The exceptions which are raised explicitly either by programmers or by API developers to indicate that something goes wrong are called programmatic exceptions.**

Eg: All user defined Exceptions

- **IllegalArgumentException**

1) **ArrayIndexOutOfBoundsException**

- **It is the child class of RuntimeException and hence is Unchecked.**

- **Raised automatically by JVM whenever we are trying to access array element with out of range index.**

Eg:

```
int[] x= new int[4];
S.O.P(x[0]);
S.O.P(x[10];) //RTE : ArrayIndexOutofbounds
S.O.P(x[-14]);//RTE : ArrayIndexOutofbounds
```

NullPointerException :

- **It is the child class of RuntimeException and hence it is unchecked.**
- **Raised automatically by JVM whenever we are trying to perform any operation on null..**

Eg:

```
String s=null;
S.O.P(s.length());    //RTE: NullPointerException
```

3.ClassCastException :

- **It is the child class of RuntimeException and hence it is unchecked.**

- **Raised automatically by JVM whenever we are trying to typecast parent objects to child type.**

Eg: String S = new String("ABC");
Object O = (Object)S;

Eg: Object O = new Object();
String S = (String) O;
RuntimeException : ClassCastException
Eg: Object O = new String("ABC");
String S = (String) O;

4.StackOverFlowError :

- **It is the child class of Error and hence it is unchecked..**
- **Raised automatically by JVM whenever we are trying to perform recursive method calls.**

5. NoClassDefFoundError

- **It is the child class of Error and hence it is unchecked.**
- **Raised automatically by JVM, whenever JVM is unable to locate/find required .class file.**

Eg: java Test
RuntimeException : NoClassDefFoundError :
→ Test is raised if Test.class is not available.

**** End Of Exception Handling ****

String Handling

String Introduction

- **String is probably the most commonly used class in the Java library.**
- **String class is encapsulated under the java.lang package.**
- **In Java, every String that you create is actually an object of type String.**
- **String objects are immutable, which means once a string object is created it cannot be altered.**

What is an Immutable object?

- **An object whose state cannot be changed after it is created is known as an Immutable object.**

- **String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

Creating an Immutable class

- Class should be final.
- The variable is also final.
- There should not be setter methods inside it.
- In this example StringDemo is an immutable class.
- StringDemo state cannot be changed once it is created.

```
public final class StringDemo
{
        final String str= “”;

  StringDemo(String s)
        {
                this.str = s;
        }
        public String get()
        {
                return str;
        }
}
```

creating a String object

- String can be created in a number of ways, here are a few ways of creating string objects.

1) Using a String literal

- **String literal is a simple string enclosed in double quotes "".**
- **A string literal is treated as a String object.**

```
String str1 = "Hello";
```

2) Using another String object

```
String str2 = new String(str1);
```

3) Using new Keyword

```
String str3 = new String("Java");
```

4) Using + operator (Concatenation)

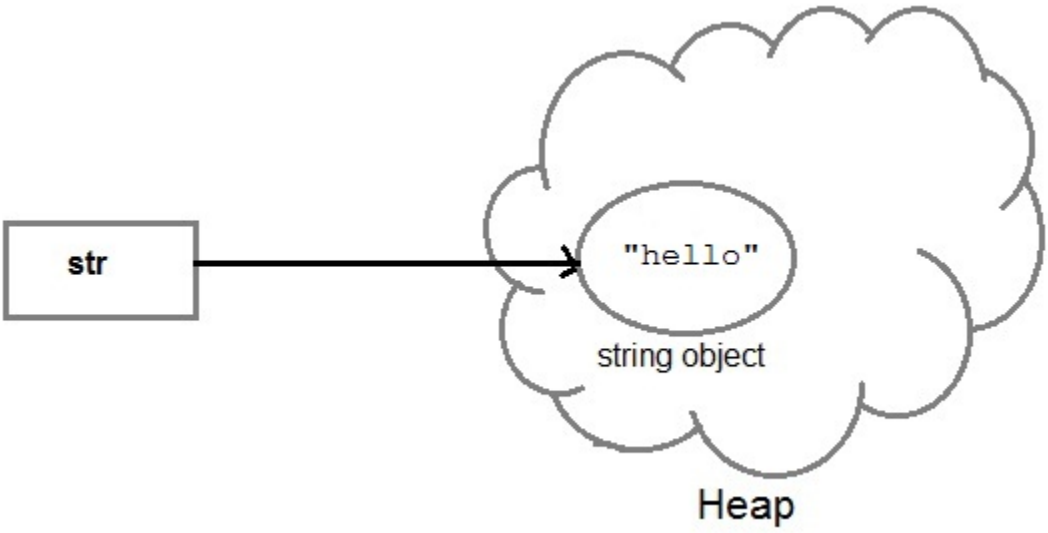
```
String str4 = str1 + str2;
or,
String str5 = "hello"+"Java";
```

- **Each time you create a String literal, the JVM checks the string pool first.**
- **If the string literal already exists in the pool, a reference to the pool instance is returned.**
- **If a string does not exist in the pool, a new string object is created, and is placed in the pool.**
- **String objects are stored in a special memory area known as a string constant pool inside the heap memory.**

String object and How they are stored

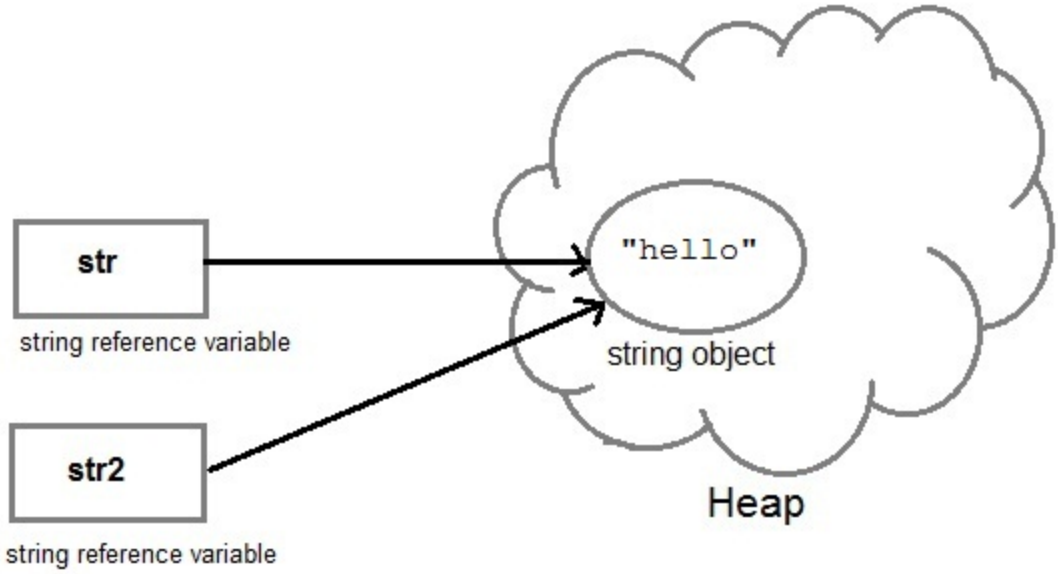
- **When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there.**

```
String str= "Hello";
```



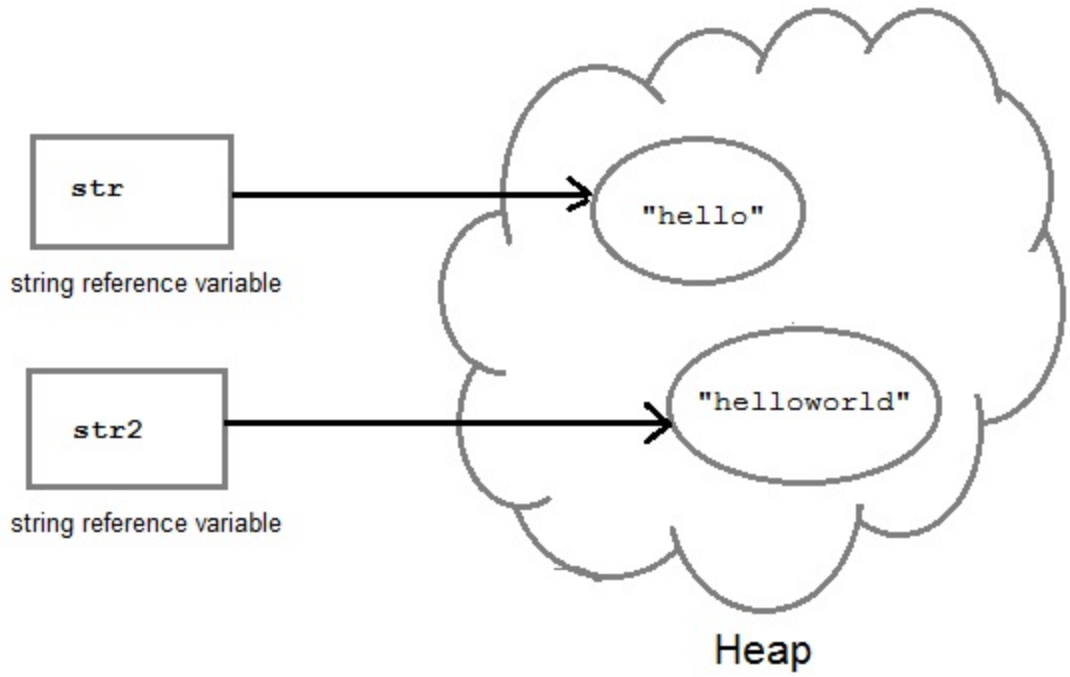
- And, when we create another object with the same string, then a reference to the string literal already present in the string pool is returned.

```
String str2=str;
```



- **But if we change the new string, its reference gets modified.**

```
str2=str2.concat("world");
```



Concatenating String

There are 2 methods to concatenate two or more strings.

1. **Using concat() method**
2. **Using + operator**

1) Using concat() method

```
string s = "Hello";
string str = "Java";
string str2 = s.concat(str);//HelloJava
String str1 = "Hello".concat("Java");    //works with string literals too.
```

2) Using + operator

```
string str = "Rahul";
```

```
string str1 = "Dravid";
string str2 = str + str1;
string st = "Rahul"+"Dravid";
```

String Comparison

String comparison can be done in 3 ways.

1. **Using equals() method**
2. **Using == operator**
3. **By CompareTo() method**

Using equals() method

- **equals() method compares two strings for equality.**
- **Its general syntax is,**

boolean equals (Object str)

- **It compares the content of the strings. It will return true if string matches, else return false.**

```
String s = "Hell";
String s1 = "Hello";
String s2 = "Hello";
s1.equals(s2);    //true
s.equals(s1) ;   //false
```

Using == operator

- **== operator compares two object references to check whether they refer to the same instance. This also, will return true on successful matches.**

```
String s1 = "Java";
```

String s2 = "Java";

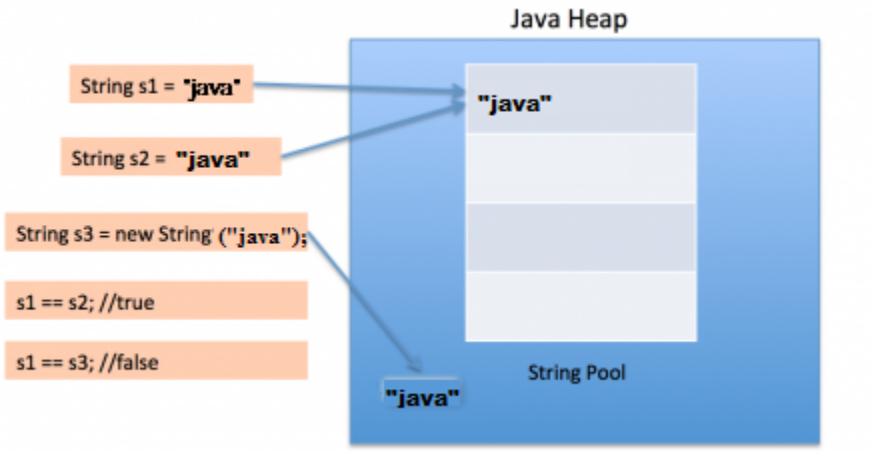
String s3 = new string ("Java");

test(s1 == s2) //true

test(s1 == s3) //false

Reason:

- **It's because we are creating a new object using the new operator, and thus it gets created in a non-pool memory area of the heap.**
- **S1 is pointing to the String in the string pool while s3 is pointing to the String in the heap and hence, when we compare s1 and s3, the answer is false.**
- **The following image will explain it more clearly.**



compareTo() method

- **compareTo() method compares values and returns an int which tells if the string compared is less than, equal to or greater than the other string.**
- **It compares the String based on natural ordering i.e alphabetically.**
- **Its general syntax is,**

int compareTo(String str)

String s1 = "Abhi";

String s2 = "Viraaj";

String s3 = "Abhi";

s1.compareTo(S2); //return -1 because s1 < s2

s1.compareTo(S3); //return 0 because s1 == s3

s2.compareTo(s1); //return 1 because s2 > s1

Java String class functions

- **The methods specified below are some of the most commonly used methods of the String class in Java.**
- **We will learn about each method with the help of small code examples for better understanding.**

charAt() method

- **charAt() function returns the character located at the specified index.**

String str = "srikanth";

System.out.println(str.charAt(2));

Output: i

NOTE:

- **Index of a String starts from 0, hence str.charAt(2) means the third character of the String str.**

equalsIgnoreCase() method

- **equalsIgnoreCase() determines the equality of two Strings, ignoring their case (upper or lowercase doesn't matter with this function).**

  String str = "java";

  System.out.println(str.equalsIgnoreCase("JAVA"));

true

indexOf() method

- **indexOf() function returns the index of the first occurrence of a substring or a character. indexOf() method has four forms.**

- **int indexOf(String str): It returns the index within this string of the first occurrence of the specified substring.**
- **int indexOf(int ch, int fromIndex): It returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.**
- **int indexOf(int ch): It returns the index within this string of the first occurrence of the specified character.**
- **int indexOf(String str, int fromIndex): It returns the index within this string of the first occurrence of the specified substring, starting at the specified index.**

Example:

```
public class Srikanth{
   public static void main(String[] args) {
      String str="srikanth";
      System.out.println(str.indexOf('i'));   //3rd form
      System.out.println(str.indexOf('n', 3));    //2nd form
      String subString="ant";
      System.out.println(str.indexOf(subString)); //1st form
      System.out.println(str.indexOf(subString,7));   //4th form
   }
}
```

Output 👍

```
2
5
-1
-1
```

NOTE: -1 indicates that the substring/Character is not found in the given String.

length() method

- **length() function returns the number of characters in a String.**

```
String str = "Count me";
System.out.println(str.length());
```

Output :
8

replace() method

- **replace() method replaces occurrences of character with a specified new character.**

```
String str = "Change me";
System.out.println(str.replace('m','M'));
```

Output :
Change Me

substring() method

- **substring() method returns a part of the string. substring() method has two forms,**

```
public String substring(int begin);
public String substring(int begin, int end);

/*
   the character of the
```

beginning index is inclusive

The end index is exclusive.

***/**

- **The first argument represents the starting point of the substring.**
- **If the substring() method is called with only one argument, the substring returned will contain characters from specified starting point to the end of the original string.**
- **But, if the call to substring() method has two arguments, the second argument specifies the end point of substring.**

```
String str = "0123456789";
System.out.println(str.substring(4));
```

output:

456789

```
System.out.println(str.substring(4,7));
```

456

toLowerCase() method

- **toLowerCase() method returns a string with all uppercase characters converted to lowercase.**

```
String str = "ABCDEF";
System.out.println(str.toLowerCase());
```

abcdef

toUpperCase() method

- **This method returns a string with all lowercase characters changed to uppercase.**

```
String str = "abcdef";
```

System.out.println(str.toUpperCase());

ABCDEF

valueOf() method

Overloaded versions of the valueOf() method are present in the String class for all primitive data types and for type Object.

NOTE: valueOf() function is used to convert primitive data types into Strings.

```
public class Example{
   public static void main(String args[]){
      int num = 35;
      String s1 = String.valueOf(num);    //converting int to String
      System.out.println(s1+"IAmAString");
   }
}
```

35IAmAString

But for objects, valueOf() method calls toString() function.

toString() method

- **toString() method returns the string representation of the object used to invoke this method.**
- **toString() is used to represent any Java Object into a meaningful string representation.**

- **It is declared in the Object class, hence can be overridden by any java class. (Object class is the super class of all java classes.)**

```
public class Car {
   public static void main(String args[])
   {
      Car c = new Car();
      System.out.println(c);
   }
   public String toString()
   {
      return "This is my car object";
   }
}
```

This is my car object

- **Whenever we try to print any object of class Car, its toString() function will be called. toString() can also be used with normal string objects.**

```
String str = "Hello World";
System.out.println(str.toString());
```

Output : Hello World

NOTE: If we don't override the toString() method and directly print the object, then it would print the object id.

Example:

```
public class Car {
```

```
    public static void main(String args[])
    {
        Car c = new Car();
        System.out.println(c);
    }
}
```

srikanth.Car@15db9742 (here srikanth is a user-defined package).

toString() with Concatenation

- **Whenever we concatenate any other primitive data type, or object of other classes with a String object, toString() function or valueOf() function is called automatically to change the other object or primitive type into string, for successful concatenation.**

```
int age = 10;
String str = "He is" + age + "years old.";
```

In above case 10 will be automatically converted into string for concatenation using valueOf() function.

trim() method

- **This method returns a string from which any leading and trailing whitespaces have been removed.**

```
String str = "   hello  ";
System.out.println(str.trim());
```

hello

NOTE:

If the whitespaces are between the strings, for example: String s1 = "sri kanth"; then System.out.println(s1.trim());

will output "sri kanth".

trim() method removes only the leading and trailing whitespaces.

StringBuffer class in Java

- **StringBuffer class is used to create a mutable string object i.e its state can be changed after it is created.**
- **It represents a growable and writable character sequence.**
- **As we know that String objects are immutable, so if we do a lot of changes with String objects, we will end up with a lot of memory leaks.**
- **So the StringBuffer class is used when we have to make a lot of modifications to our string.**
- **It is also thread safe i.e multiple threads cannot access it simultaneously.**
- **(Synchronous)**

StringBuffer defines 4 constructors. They are,

1. **StringBuffer ( )**
2. **StringBuffer (*int size*)**
3. **StringBuffer (*String str*)**
4. **StringBuffer (*charSequence [ ]ch*)**

- **StringBuffer() creates an empty string buffer and reserves room for 16 characters.**
- **stringBuffer(int size) creates an empty string and takes an integer argument to set the capacity of the buffer.**

Example showing difference between String and StringBuffer

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

```
 {
  String str = "sri";
  str.concat("kanth");
  System.out.println(str);      // Output: sri

  StringBuffer strB = new StringBuffer("sri");
  strB.append("kanth");
  System.out.println(strB);    // Output: srikanth
 }
}
```

Reason:

- **Output is such because String objects are immutable objects.**
- **Hence, if we concatenate on the same String object, it won't be altered(Output: sri).**
- **But StringBuffer creates mutable objects. Hence, it can be altered(Output: srikanth)**

Important methods of StringBuffer class

- **The following methods are some of the most commonly used methods of StringBuffer class.**

append()

- **This method will concatenate the string representation of any type of data to the end of the invoking StringBuffer object. append() method has several overloaded forms.**

StringBuffer append(String str)

StringBuffer append(int n)

StringBuffer append(Object obj)

- **The string representation of each parameter is appended to the StringBuffer object.**

StringBuffer str = new StringBuffer("test");

str.append(123);

System.out.println(str);

test123

insert()

- **This method inserts one string into another. Here are a few forms of insert() method.**

StringBuffer insert(int index, String str)

StringBuffer insert(int index, int num)

StringBuffer insert(int index, Object obj)

- **Here the first parameter gives the index at which position the string will be inserted and the string representation of the second parameter is inserted into the StringBuffer object.**

StringBuffer str = new StringBuffer("test");

str.insert(4, 123);

System.out.println(str);

test123

reverse()

This method reverses the characters within a StringBuffer object.

StringBuffer str = new StringBuffer("Hello");

str.reverse();

System.out.println(str);

olleH

replace()

This method replaces the string from the specified start index to the end index.

<

```
StringBuffer str = new StringBuffer("Hello World");
str.replace( 6, 11, "java");
System.out.println(str);
```

Hello java

capacity()

This method returns the current capacity of the StringBuffer object.

```
StringBuffer str = new StringBuffer();
System.out.println( str.capacity() );
```

16

Note: Empty constructor reserves space for 16 characters. Therefore the output is 16.

ensureCapacity()

- **This method is used to ensure minimum capacity of StringBuffer objects.**
- **If the argument of the ensureCapacity() method is less than the existing capacity, then there will be no change in existing capacity.**
- **If the argument of the ensureCapacity() method is greater than the existing capacity, then there will be change in the current capacity using following rule:**

newCapacity = (oldCapacity*2) + 2.

StringBuffer str = new StringBuffer();

System.out.println(str.capacity()); //output: 16 (since empty constructor reserves space for 16 characters)

str.ensureCapacity(30); //greater than the existing capacity

System.out.println(str.capacity()); //output: 34

StringBuilder class in Java

- **StringBuilder is identical to StringBuffer except for one important difference that it is not synchronized, which means it is not thread safe.**
- **It's because StringBuilder methods are not synchronized.**

StringBuilder Constructors

1. **StringBuilder ( ), creates an empty StringBuilder and reserves room for 16 characters.**
2. **StringBuilder (*int size*), creates an empty string and takes an integer argument to set the capacity of the buffer.**
3. **StringBuilder (*String str*), create a StringBuilder object and initialize it with string str.**

Difference between StringBuffer and StringBuilder class

StringBuffer class	StringBuilder class
StringBuffer is synchronized.	StringBuilder is not synchronized.
Because of synchronisation, StringBuffer operation is slower than StringBuilder.	StringBuilder operates faster.

Example of StringBuilder

```
class Test {
 public static void main(String args[])
 {
  StringBuilder str = new StringBuilder("sri");
  str.append( "kanth" );
  System.out.println(str);
  str.replace( 6, 13, "java");
  System.out.println(str);
  str.reverse();
  System.out.println(str);
  str.replace( 6, 13, "java");
 }
}
```

srikanth

srikanjava

Avajnakirs

********End of String Handling.********

File IO

- File
- FileWriter
- FileReader
- BufferedWriter
- BufferedReader
- PrintWriter

File :

```
File f =new File("D://abc.txt");
```

- This line won't create any physical file.
- First it will check if there is any physical file named with 'abc.txt' available or not.
- If it is available 'f ' simply refers to that file, if it is not already available then it won't create any physical file, just we are creating a 'Java file object ' to represent the name abc.txt.

    - File f = new File ("D://abc.txt");
    - System.out.println(f.exists()); //false //true

    - f.createNewFile();
    - System.out.println(f.exists()); //true //true

First run : false, true
Second run: true, true

- File f = new File("Sri5");
  System.out.println(f.exists()); //false
  f.mkdir();
  System.out.println(f.exist()); // true

- We can use java file objects to represent the directory also.
- Note : In unix everything is file and Java file IO concept is implemented based on the unix operating system, hence we can use java file objects to represent both files and directories.

File class constructors :

- **File f = new File(String name);**
- **It creates a java file object to represent the name of the File or directory in the current working directory.**

- **File f = new File(String subdir name, String name);**
- **It creates a java File object to represent the name of the file or directory in the specified location.**

- **Ex1 : Write code to create a file named with abc.txt in CWD .**
- **File f = new File ("abc.txt");**
  f.createNewFile();
- **Ex2: Write code to create a directory named with "Sri5" in CWD and create a file named demo.txt in that directory.**

```
File f = new File("Sri5");
f.mkdir();

//File f1 =new File("Sri5", "demo.txt");
File f1 = new FIle(f, "demo.txt");
f1.createNewFile();
```

- **Ex 3: Write code to create a file named with abc.txt in E://XYZ folder.**
- **File f = new File("E:\\xyz" , "abc.txt");**
  f.createNewFile();
- **Assume the E:\\xyz folder is already available in our system.**

⇒ Important methods of File class

- **boolean exists() : It returns true if the specified file or directory is already available.**

- **boolean createNewFile() : First this method will check whether that file is already available or not, if it is already available then this method returns false without creating any physical file.**

- **If it is not already available then this method creates a new file and returns true .**

- **boolean mkdir() : Same as above**

- **boolean isFile() : Returns true if the file reference points to a physical file.**

- **boolean isDirectory() : Same as above.**
- **String[] list() : Returns the names of all files and subdirectories present in specified directory.**
- **long length() : Returns the number of characters present in the File.**
- **boolean delete() :**

Write a program to display the names of all files and subdirectories in → D: Sri5

Example : 1

```
package fileio;

// To display the names of all files and subdirectories :
import java.io.File;
import java.io.IOException;

public class TestFile1 {
	public static void main(String args[]){
//		int count=0;

	//Text File creation i
		File f=new File("D://sri.txt");
		System.out.println(f.exists());

		try {

			f.createNewFile();
			System.out.println(f.exists());

		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}


	//Directory creation
		File f1=new File("D://Sri5");
		System.out.println(f1.exists());
```

```
            f1.mkdir();

        //Directory with file name
            File f2= new File("D://Sri5","abc.txt");

            try {
                f2.createNewFile();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            System.out.println(f2.exists());


            String[] s= f1.list();

//          for(String s1:s){
//              count ++;
//              System.out.println(s1);
//          }
//          System.out.println("The Total Number : " +count);
        }
}
```

FileWriter :

- **We can use FileWriter to write character data to the File.**

Constructors :

1) **FileWriter fw = new FileWriter(String name);**
2) **FileWriter fw = new FileWriter (File f);**

- **The above constructors are meant for overriding existing data.**
- **Instead of overriding if we want an append operation then we have to use the following two constructors.**

3) **FileWriter fw = new FileWriter(String fname, boolean append);**
4) **FileWriter fw = new FileWriter(FIle f, boolean append);**

- **Note : If the specified file is not already available then the above constructors will create the required file .**

- **Methods :**
  write(int ch) → To write a single character to the file
  write(char[] ch) →
  write(String s) →
  flush() → To give the guarantee that data including the last character is written properly to the file.
  close() ;

Example :

```
package fileio;

import java.io.FileWriter;
import java.io.IOException;

public class FileWriterDemo2 {
        public static void main(String args[]) throws IOException {

                FileWriter fw= new FileWriter("D://Sri5/hello.txt", true);

                fw.write(68); // Adding Single
                fw.write('\t');
                fw.write("Sri ");
                fw.write('\n');
                char[] ch1={'a', 'b' , 'c'};
                fw.write(ch1);
                fw.flush();
                fw.close();
        }
}
```

- **In the above program FileWriter is meant for overriding existing data.**
- **Instead of overriding if we want append operation then we have to create FileWriter object as follows.**
  - **FileWriter fw= new FileWriter () ;**
- **The main problem with FileWriter is we have to insert line separators manually, which varies from system to system. It is difficult for the programmer.**

FileReader

- **We can use FileReader to read character data from the file.**

Constructors :

FileReader fr = new FileReader(String fname);
FileReader fr = new FileReader(File f);

Methods :

- **int read()**
- **It attempts to read the next character in the file and returns its unicode value. If the next character is not available then this method returns '-1' .**
- **As this method returns a unicode value at the time of printing we have to perform type casting.**

Ex:

```
FileReader fr = new Filereader(abc.txt);
        int i = fr.read();
        while(i !=-1)
        {
                System.out.println((char)i);
                i=fr.read();
        }
```

- **int read(char[] ch) → It attempts to read enough characters from the file into char[] and returns the number of characters copied.**

Ex:

```
File f = new File(abc.txt);
char [] ch= new char[(int)f.length()];
FileReader fr = new FileReader(f);
fr.read(ch);

for(char ch1: ch)
{
System.out.println(ch1);
}
```

- **void close() ;**

Example :

```
package fileio;
```

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class FileReaderDemo {
	public static void main(String[] args) throws IOException{

//		File f =new File("D://Sri5/abc.txt");
//		FileReader fr=new FileReader(f);
//
//
//		char[] ch= new char[(int) f.length()];
//		fr.read(ch);
//
//		for(char ch1: ch){
//			System.out.println("**" + ch1);
//		}

		FileReader fr1 = new FileReader("D://Sri5/abc.txt");
		int i= fr1.read();

		while(i !=-1){
			System.out.println((char)i +"***");
			i = fr1.read();
		}

		fr1.close();
//		fr1.close();


	}
}
```

Note :

- **The main problem with Filereader is we have to read the data character by character which is not convenient to the programmer.**
- **Usage of FileWriter and FileReader is not recommended because**
- **While writing data by FileWriter we have to insert line separators manually which varies from system to system. It is difficult for the programmer.**
- **By using Filereader we can read data character by character which is not convenient to the programmer.**

- **To overcome the above problems we should go for BufferedWriter and BufferedReader concepts.**

BufferedWriter

- **We can use BufferedWriter to write character data to the file.**

- **Constructors :**
  BufferedWriter bw = new BufferedWriter (Writer w);
  BufferedWriter bw = new BufferedWriter (Writer w, int bufferSize);

- **Note :**
  BufferedWriter can't communicate directly to the file and it can communicate via some writer object.

Q: Which of the following are valid?

BufferedWriter bw = new BufferedWriter ("abc.txt"); → In valid
BufferedWriter bw = new BufferedWriter (new File ("abc.txt")); → In valid
BufferedWriter bw = new BufferedWriter (new FileWriter ("abc.txt")); → valid
BufferedWriter bw = new BufferedWriter (new BufferedWriter(new FileWriter ("abc.txt"))); → valid

- **Methods :**
  write(int ch);
  write(char[] ch);
  write(String s);
  flush();
  close();
  newLine(): To insert a line separator

Q: When compared with FileWriter which of the following capabilities is available extra in the method from in BufferedWriter.

- **Writing data to the file.**

- **Closing the file.**
- **flushing the file.**
- **inserting a newline character → valid**

Example :

```
package fileio;
import java.io.FileWriter;
import java.io.IOException;
public class FileWriterDemo2 {
        public static void main(String args[]) throws IOException {

                FileWriter fw= new FileWriter("D://Sri5/hello.txt");

                fw.write(68); // Adding Single
                fw.write('\n');
                fw.write("Srikanth, Venkat, Prasanthi and divya ");
                fw.write('\n');
                char[] ch1={'a', 'b' , 'c','d'};
                fw.write(ch1);

                fw.flush();
                fw.close();

        }
}
```

- **Note :**
- **Whenever we are closing, BufferedWriter automatically underlines that FileWriter will be closed. Hence we are not required to close explicitly.**
  bw.close | fw.close(); | bw.close(); fw.close();

BufferedReader :

- **We can use BufferedReader to read character data from the file.**

- **The main advantage of BufferedReader over FileReader is we can read the data line by line instead of character by char.**

  Constructors :

  ```
  BufferedReader  br = new BufferedReader (Reader r);
  BufferedReader  br = new BufferedReader (Reader r, int BufferSize );
  ```

  Note :

  BufferedReader can't communicate directly with the file and it can communicate via some reader object.

  Methods :

  ```
  int read();
  int read(char ch[]);

  void close();
  String readLine();
  ```

- **It attempts to read the next line in the file and returns it.**
  If the next line is not available then we will get null.

  Example:

```
package fileio;

import java.io.BufferedReader;
import java.io.FileReader;

public class BufferedReaderDemo {
        public static void main(String args[]) throws Exception {

                FileReader fr= new FileReader("D://Sri5/abc.txt");
                BufferedReader br = new BufferedReader(fr);

                String line = br.readLine();

                while(line != null){
                        System.out.println(line);
                        line = br.readLine();
                }
```

```
            br.close();
        }
    }
```

Note :

→ Whenever we are closing, the BufferedReader underlying FileReader object will be closed and we are not required to close explicitly.

br.close () | fr.close() | br.close() ; fr.close() ;

PrintWriter

- **It is the most enhanced writer to write character data to the file.**
- **The main advantage of PrintWriter over FileWriter is that we can write any type of primitive data directly to the File.**

Constructors :

```
PrintWriter  pw = new PrintWriter (String fname);
PrintWriter  pw = new PrintWriter (File f);
PrintWriter  pw = new PrintWriter (Writer w);
```

Note : Printwriter can communicate either directly to the file or via some writer object also.

Methods :

```
write(int ch);
write(Char[] ch);
write(String s);

flush();
close();

print(char ch);
print(int  i);
print(String s);
print(boolean b);
print(double d );

println(char ch);
println(int i);
println(String s);
println(boolean b);
```

println(double d);

Example :

```
package fileio;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriterDemo1 {
	public static void main(String[] args) throws IOException{

		FileWriter fw = new FileWriter("D://Sri5/abc.txt");
		PrintWriter out = new PrintWriter(fw);
//		PrintWriter out = new PrintWriter("D://Sri5/abc.txt");
		out.write(100);
		out.println(100);
		out.println(true);
		out.println('c');
		out.println("srikanth");
		out.flush();
		out.close();
	}
}
```

- What is the difference between write(100) and print(100)..?
- In the case of write(100) the corresponding char ‘d’ will be returned to the file.
- But in the case of print(100) the corresponding int value 100 will be written directly to the file.

- Note :
- The most enhanced writer to write character data to the file is PrintWriter whereas the most enhanced Reader to read character data from the file is BufferedReader .

- Note
- In general readers and writers handle character data whereas we can use streams we handle Binary data (images, audio files, video files ext).

- we can use below

FileOutputstream to write Binary Data to the file whereas
FileInputstream to read Binary data from the file.

-**End of the Files****

Collections framework

- **Introduction of Arrays**
- **An array is an indexed collection of fixed numbers of homogeneous data elements.**
- **The main advantage of arrays is we can represent multiple values by using a single variable so that readability of the code will be improved.**

- **limitations**

- **Arrays are fixed in size, hence once we create an array we can't increase or decrease size based on our requirement, hence to use arrays concept compulsory we should know the size in advance, which may not be possible.**
- **Arrays can hold only homogeneous data type elements.**

ex:

```
Student[] s = new Student[1000];
s[0]= new Student();
s[1]=new Customer();
// CE : incompatible types found: Customer required: Student
```

- **We can solve this problem by using objects for arrays hence predefined method support is not available.**

```
Object[] a = new Object[1000];
        a[0]= new Student();
        a[1]= new Customer();
```

- **There is no underlying data structure for arrays hence predefined method support is not available.**

Collections

- **Growable in nature.**
- **Can hold both homogeneous and heterogeneous elements.**

- **Every collection class is implemented based on some standard data structure hence for every requirement pre defined (readymade) method support available.**

- **Student [] s = new Student[50];**

Difference between arrays and collections:

Arrays	**Collections**
Arrays are fixed in size.	**Collections are dynamic in size.**
W.R.T Memory arrays are not good to use.	**W.R.T Memory Collections are good to use.**
W.R.T Performance it's better to use arrays.	**W.R.T Performance it's not good to use collections.**
Arrays can hold only homogeneous elements.	**collections can hold both homogeneous and heterogeneous elements.**
Arrays don't have predefined methods.	**collections have predefined data structures and methods.**
Arrays can hold both primitives and wrapper objects.	**collections can hold only objects.**

collection :
If you want to represent a group of individual objects AS A SINGLE ENTITY THEN WE SHOULD GO FOR COLLECTION .

Collection Framework :-
It defines several classes and interfaces which can be used to represent a group of individual objects as a single unit.

9 key interfaces :

1) **Collection**
- **Define the most common methods, which are applicable for any Collection object, root of collection framework.**

2) **List**
- **Where duplicates are allowed and insertion order must be preserved .**
- **List(I) → ArrayList (C), LinkedList(C), vector(C), stack(C).(2 are Legacy)**

3) **Set**
- **Where duplicates are not allowed and insertion order not preserved.**
- **Set(I) → HashSet(C), LinkedHashSet ©**

4) **SortedSet :**
- **Where duplicates are not allowed and all objects are required to insert according to some sorting order then we should go for SortedSet.**
- **Set(I) → SortedSet(I) → NavigableSet(I) → TreeSet (c)**

5) **NavigableSet :**
- **It defines several methods for navigable purposes.**
- **Set(I) → SortedSet(I) → NavigableSet(I) → TreeSet(C)**

6) **Queue :**
- **If we want to represent a group of individual objects prior to processing then we should go for Queue.**
- **Collection(I) → Queue (I)→ BlockingQueue →**
- **i)PriorityBlockingQueue (C) ii) LinkedBlockingQueue(C)**
  Collection(I) → Queue (I) → PriorityQueue (C)

7) **Map :**
- **Map is not a child interface of Collection.**
- **If we want to represent a group of objects as key value pairs then we should go for Map.**
- **Duplicate Keys are not allowed but values can be**
  Map (I) → HashMap (c)
- **LinkedHashMap(c)**
- **WeakHashMap(c)**
- **IdentityHashMap(c)**

8) **NavigableMap**
9) **SortedMap**

Note : Collections is utility class , to define several utility methods for collection objects.

1) **Collection :**

- **Represent a group of individual objects as a single entity then we should go for Collection.**
- **These are applicable for any Collection Object.**

```
boolean add(Object o)
boolean addAll(Collection c)

boolean remove(Object o)
boolean removeAll(Collection c)

boolean retainAll(Collection C)
void clear()

boolean contains(Object o)
boolean containsAll(Collection c)

boolean isEmpty()
int size();

Object[] toArray();
Iterator iterator();
```

- **There is no direct method in the collection interface to get Objects.**

2) **List :**

- **where duplicates are allowed and insertion order must be preserved .**

List(I) :::

ArrayList (C),
LinkedList(C),
vector(C) and stack(C).(2 are Legacy)

- **We can differentiate duplicate Objects and We can preserve insertion order by using indexes.**
- **Index plays a very important role in List Interface.**

```
void add(int index, Object o)
void addAll(int index, Collection c)

Object get (int index)
Object set (int index, Object  o)

int indexOf(Object)
int lastIndexOf(Object o)

ListIterator  listIterator();
Object remove(int index)
```

ArrayList : (C)

- **The underlying data structure for arrayList is a Resizable-Array or Growable-Array.**
- **Duplicates objects allowed.**
- **Insertion order is preserved.**
- **Null insertion is possible.**
- **Heterogeneous objects are allowed (except TreeSet and TreeMap) everywhere Heterogeneous objects are allowed.**

constructors :-

i) ArrayList l = new ArrayList();

- **create an empty ArrayList object with default initial capacity 10.**
- **Once ArrayList reaches its max capacity a new ArrayList object will be created with new capacity = (current capacity * 3/2) ; //**

ii) ArrayList l = new ArrayList(int initialcapacity);

iii) ArrayList l = new ArrayList(Collection c);

-

- *****Usually we can use collections
- - “to hold and transfer data from one location to another location”,
- to provide support for this requirement every collection class already implements Serializable and Cloneable interfaces.

- ArrayList and Vector classes implement RandomAccess i/f so that we can access any random element with the same speed. 0(1)

*Hence if our requirement is retrieval then ArrayList is the best Choice.

Difference between ArrayList and Vector .

- No method present in ArrayList is synchronized.
- Most of the methods present in Vector are synchronized.

- At a time multiple threads are allowed to operate on ArrayList objects simultaneously and hence the ArrayList object is not thread safe.
- At a time only one thread is allowed to operate on a vector object and hence the vector is thread safe.

- Threads are not required to wait to operate on ArrayList and hence relatively performance is high.
- Threads are required to wait to operate on vector object.hence relatively low performance.

- Not a legacy class and 1.2 version
- Legacy class and 1.0 version.

Note:

- **ArrayList is the best choice if the frequent operation is a retrieval operation, because ArrayList implements RandomAccess interface.
- ArrayList is the worst choice if the operation is insertion or deletion in the middle because several shift operations are required internally.

LinkedList :

- The underlying data Structure is a doubly-Linked List.
- Duplicates are allowed and insertion order preserved and heterogeneous objects are allowed.
- null insertion possible.

Note:

- **If our frequent operation is insertion or deletion in the middle, LL is the best option and worst choice if operation is retrieval.**

LL specific methods :
void addFirst(Object o);
void addLast(Object o);
Object getFirst();
Object getLast();

Object removeFirst();
Object removeLast();

Constructors :
LinkedList ll = new LinkedList();
LinkedList ll = new LinkedList(Collection c);

Vector :

Example :

```
import java.util.*;

public class VectorExample {

  public static void main(String args[]) {
    /* Vector of initial capacity(size) of 2 */
    Vector<String> vec = new Vector<String>(2);

    /* Adding elements to a vector*/
    vec.addElement("Apple");
    vec.addElement("Orange");
    vec.addElement("Mango");
    vec.addElement("Fig");

    /* check size and capacityIncrement*/
    System.out.println("Size is: "+vec.size());
```

```
    System.out.println("Default capacity increment is: "+vec.capacity());

    vec.addElement("fruit1");
    vec.addElement("fruit2");
    vec.addElement("fruit3");

    /*size and capacityIncrement after two insertions*/
    System.out.println("Size after addition: "+vec.size());
    System.out.println("Capacity after increment is: "+vec.capacity());

    /*Display Vector elements*/
    Enumeration en = vec.elements();
    System.out.println("\nElements are:");
    while(en.hasMoreElements())
      System.out.print(en.nextElement() + " ");
  }
}
```

Stack :

Example :

```
import java.util.Stack;
public class StackBasicExample {
   public static void main(String a[]){
      Stack<Integer> stack = new Stack<>();
      System.out.println("Empty stack : "  + stack);
      System.out.println("Empty stack : "  + stack.isEmpty());
      // Exception in thread "main" java.util.EmptyStackException
      // System.out.println("Empty stack : Pop Operation : "  + stack.pop());
      stack.push(1001);
      stack.push(1002);
      stack.push(1003);
      stack.push(1004);
      System.out.println("Non-Empty stack : "  + stack);
      System.out.println("Non-Empty stack: Pop Operation : "  + stack.pop());
      System.out.println("Non-Empty stack : After Pop Operation : "  + stack);
      System.out.println("Non-Empty stack : search() Operation : "  + stack.search(1002));
      System.out.println("Non-Empty stack : "  + stack.isEmpty());
   }
}
```

3 cursors of Java :

- **If we want to get objects one by one from a collection then we should go for a cursor.**

- **3 types of cursors..**

Enumeration
Iterator
ListIterator

Enumeration:

- **We can use it to get an object one by one from the legacy collection (vector and stack).**
- **Limitations:We can apply this concept only for legacy classes and it is not a universal cursor.**
- **By using this we can perform only read operation, we cannot perform remove operation.**

Vector
public Enumeration elements();
==========
Vector v = new Vector();
Enumeration e = v.elements();
public boolean hasMoreElements();
public Object nextElement();

- **To overcome these limitations we should go for an iterator.**

Iterator :

- **We can apply this concept to any collection object hence it is a universal cursor.**
- **By using an iterator we can perform both read and remove operations.**
- **We can create an iterator object by using the iterator() method.**

public boolean hasNext();
public Object next();
public void remove();
Limitations :
→ By using enumeration and iterators we can always move only towards the forward direction; we cannot move backward direction i.e.., these are single direction cursors.

- **By using an iterator we can perform read and remove operations and we cannot perform replacement and addition of new objects.**

To overcome these, we should go for a listiterator.

Listiterator:

- **By using this concept we can move either backward or forward direction(Bi-directional cursor).**
- **While iterating by listiterator we can perform replacement and addition of new objects, In addition to remove and read operations.**

```
public boolean hasNext();
public Object next();                          //forward operations
public void remove();

public boolean hasPrevious();
public Object previous(); add          // Backward operations
public int previousIndex();


public void remove()
public void set(Object new Object)
public void add (Object new Object )
```

Set :

→ Collection → Set→ HashSet(C) → LinkedHashSet(C)
→ Collection → Set→ SortedSet (I)→ NavigableSet(I) → TreeSet(C)

- **Set is the child interface of a collection.**
- **If we want to represent a group of individual objects where duplicates are not allowed and insertion order is not preserved then we should go for Set.**

- **Set interface doesn't define any new methods.**
- **We have to use collection interface methods only.**

HashSet :

- **The underlying data Structure is Hashtable.**
- **Duplicated objects are not allowed.**
- **If we are trying to add duplicate objects, we won't get any compile time or runtime error, and the add method simply returns false.**

- **Heterogeneous objects are allowed.**

- **Null insertion is possible only once.**
- **implements serializable and cloneable interfaces but not RandomAccess.**
- **HashSet is the best choice, if our frequent operation is search operation.**

constructors :

HashSet h = new HashSet();
HashSet h = new HashSet(int initialCapacity);
HashSet h = new HashSet(int initialCapacity, float fillRatio);
HashSet h = new HashSet(Collection c);

LinkedHashSet :

- **It is the child class of HashSet.**
- **The underlying data Structure is Linked List+Hash Table.**
- **Duplicated objects are not allowed.**
- **Insertion order must be preserved.**
- **If we are trying to add duplicate objects, we won't get any compile time or runtime error, and the add method simply returns false.**
- **Heterogeneous objects are allowed.**

SortedSet : (I)

- **Some sorting order.**

100,101,103,107,110,115

first() → 100
last() → 115
headSet(107) → [100,101,103] (Less Than)
tailSet(107) → [107, 110,115] (GT =)
subSet(101,110) → [101,103,107]
comparator() → null.

TreeSet :(C)

- **Balanced Tree.**
- **According to some sorting order it will print.**

- **Heterogeneous objects are not allowed, otherwise we will get runtime exceptions… CCE**

- **default natural sorting order → objects should be 'Homogeneous and Comparable' .**

Comparable i/f : java.lang

- **It contains only one method compareTo()**
  obj1.compareTo(obj2) →

return → -ve ⇐> obj1 has to come before obj2 // 100 to come before 200 → -1
return → +ve ⇐> obj1 has to come after obj2 // 200 to come after 100 → 1
return → 0 ⇐> obj1 and obj2 are equal // 200 equals to 200 → 0

S.o.p ("A".compareTo("Z")); → -ve AZ
S.o.p ("Z".compareTo("K")); → +ve KZ

Comparator → java.util

- **our own sorting order.**

compare()
equal()

Public int compare(Object obj1, Object obj2)

return → -ve ⇐> obj1 has to come before obj2
return → +ve ⇐> obj1 has to come after obj2
return → 0 ⇐> obj1 and obj2 are equal .

Map :

- **Map is not a child interface or Collection interface.**
- **If we want to represent a group of objects as Key value pairs then we go for Map.**
- **Duplicate keys are not allowed but values can be duplicated.**
- **Each key value pair is called "Entry".**

Important methods of Map Interface :

- **Object put(Object key, Object value)**
- **To add one key value pair to the map.**

- **If the key is already available then the old value will be replaced with a new value and the old value will be returned.**

void putAll(Map m).

Object get(Object key)
Object remove(Object key)

boolean containsKey(Object key)
boolean containsValue(Object value)

boolean isEmpty()
set keySet()
collection values()

set entrySet()

Entry interface :

- **In the map each key value pair is called an Entry..**
- **Without an existing map object there is no chance of an existing Entry object.**
- **Hence Entry interface is defined inside map interface.**

```
interface Map{
 interface Entry {
        {
        Object getKey()
        Object getValue()
        Object getValue(Object new)
}

}
```

HashMap :

- **The underlying data structure is HashTable.**
- **Duplicate keys are not allowed but values can be duplicated.**
- **Heterogeneous objects are allowed for both keys and values.**
- **“null” insertion is allowed for key (only once) and for value (any no of times).**
- **Insertion order is not preserved and it is based on the HashCode of the keys.**
- **Hashmap is the best choice if our frequent operation is search operation.**

Constructors :
→ HashMap m = new HashMap();
→ HashMap m = new HashMap(int initialCapacity);
→ HashMap m = new HashMap(int initialCapacity, float fill Ratio);
→ HashMap m = new HashMap(map m);

LinkedHashMap :

- **It is the child class of HashMap.**
- **It is exactly same as HashMap except the following diff ||.**

HashMap	LinkedHashMap
The underlying data structure is Hashtable	The underlying data structure is LinkedList + Hash Table.
Insertion order not preserved.	Insertion order is preserved.
Introduced in 1.2 version.	Introduced in the 1.4 version.

Note :

- **In general we can use LinkedHashSet and LinkedHashMap for developing cache based applications where duplicates are not allowed and insertion order must be preserved.**

IdentityHashMap :

Note :

- **"==" operator always meant for reference comparison (address comparison) whereas**
- **".equals()" method meant for content comparison.**

```
Integer i1= new Integer(10);
Integer i2= new Integer(10);

s.o.p(i1==i2);  //false
s.o.p(11.equals(i2)); true
```

- **IdentityHashMap is exactly the same as HashMap except the following diff .**

- **In the case of HashMap JVM will give the ".equals()" method to identify duplicate keys, which is meant for content comparison.**

- **But in the case of IdentityHashMap Jvm will use "==" operator to identify duplicate keys, which is meant for reference comparison.**

```
HashMap m = new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);

m.put(I1, "Pawan");
m.put(I2, "Kalyan");
s.o.p(m);  {10= kalyan}
```

- **If we replace hashMap with IdentityHashMap then I1 & I2 are not duplicate keys because I1==I2 returns false. In this case output is {10= pawan, 10 = kalyan}**

WeakHashMap :

- **It is exactly the same as HashMap except the following diff.**
- **In the case of HashMap even though the object doesn't have any reference still it is not eligible for 'gc' if it is associated with HashMap.**

Note : I.e HashMap dominates garbage Collector .

- **But in the case of WeakHashMap if the object doesn't have any reference then it is eligible for garbage collection even though it is associated with WeakHashMap**

Note : i.e garbage collector dominates WeakHashMap.

SortedMap :

- **It is the child interface of Map.**
- **If we want to represent a group of objects as key-value pairs according to some sorting order of keys then we should go for sortedMap.**
- **Sorting is only based on the key but not based on the value.**
- **The SortedMap interface defines the following methods.**

Object firstKey();
Object lastKey();
SortedMap headMap();
SortedMap tailMap();
SortedMap subMap(Object key1, Object key2);
Comparator comparator();

TreeMap :

- **Underlying data structure is the "RED-BLACK Tree".**
- **Duplicate keys are not allowed but values can be duplicated.**
- **Insertion Order is not preserved and it is based on some sorting order of keys.**
- **If we are depending on default natural sorting order then keys should be “homogeneous and comparable” otherwise we will get a RuntimeException saying classCastException.**
- **If we are defining our own sorting order by comparator.**
- **There are no restrictions for values that can be heterogeneous and non-comparable.**

null acceptance :

For an empty TreeMap as First Entry with a null key is allowed but after inserting that Entry if we are trying to insert any other entry we will get NPE.

For Non Empty TreeMap if we are trying to insert an Entry with a null key then we will get NPE.

For the values we can use "null" any no.of times and there are no restrictions.

TreeMap t= new TreeMap(); // For default sorting order.
TreeMap t= new TreeMap(Comparator c); //For customized Sorting order.
TreeMap t= new TreeMap(SortedMap m);
TreeMap t= new TreeMap(Map m);

Hashtable :

- **Underlying data structure is Hashtable.**
- **Duplicate keys are not allowed but values can be duplicated.**
- **Heterogeneous objects are allowed for both keys and values.**

- **Insertion order is not preserved and it is based on HashCode Keys.**
- **null is not allowed for both keys and values otherwise we will get NPE.**
- **Most of the methods present in Hash Table are Synchronized hence Hashtable object "thread-safe".**
- **This is Best choice for search.**
- **How to get synchronized version of hashmap object :**
- **By default HashMap is Non- synchronized but we can get a synchronized version of HashMap by using the Synchronized map method of Collection class.**

Ex :-
HashMap() m = new HashMap();
Map m1= Collections.synchronizedMap(m);

Queue :

- **Queue is the child interface of the collection.**
- **If we want to represent a group of individual objects prior to processing then we should go for 'Queue'.**

Ex:- Before Sending mail, we have to store all mail ids in some data structure in which order we saved in the order only mail has to deliver (FIFO) for this requirement Queue is the best Choice.

- **Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also.**
- **From 1.5 version onward**

PriorityQueue. → Same as SortedSet
Blocking Queue.

****End of Collections***

MultiThreading

Introduction :

"The ways to define threads

by extending Thread class" and
"By implementing Runnable Interface".

Multitasking :

"Executing several tasks simultaneously is the concept of multitasking."
There are two types of multitasking.

1) **PBM (Process based multitasking)**
2) **TBM (Thread Based multitasking)**

PBM (Process based multitasking)

- **Executing several tasks simultaneously where each task is a separate independent program(process) is Called PBM.**
- **These are OS level but not at Programmatic level.**

Ex: CPU, Browser.

TBM (Thread Based multitasking)

- **Executing several tasks simultaneously where each task is a separate independent part of the same program is called TBM and each independent part is called a Thread.**
- **TBM is best suitable at programmatic level.**
- **Whether it is PB or TB the main advantage of multitasking is to reduce response time and to improve the performance of the system.**

The main important application areas of multithreading are

- **To develop multimedia graphics.**
- **To implement Video animation.**
- **To develop video games.**
- **To develop a Server.**
- **To develop servers and web application servers.**
- **When comparing C++ developing multithreading programs in java is very easy because java provides inbuilt support by providing a RICH API (Thread, ThreadGroup, Runnable etc...)**

Defining a Thread :

- **We can define a Thread in the following 2 ways.**

- **By Extending the Thread class.**
- **By implementing the Runnable Interface.**

1) **Defining Thread by extending the Thread class.**

```
Class MyThread extends Thread {
public void run(){
  for(int i=0; i<10;i++){
  System.out.println("Child Thread");
}
}
Class ThreadDemo{
public static void main(String args[]){
        MyThread t= new MyThread();
        t.start();
for(int i =0; i<10 ;i++){
System.out.println("Main Thread");
}
}
```

Case 1 :

Thread Scheduler :

- **If multiple threads are waiting for execution then in which order threads will be executed is decided by Thread Scheduler.**

- **Threads scheduler is the part of JVM and we can't expect an exact algorithm followed by Thread scheduler, It varies from JVM to JVM and hence we can't expect exact output for multithreaded programs but we can provide several possible outputs.**

- **The following various possible outputs for the above programs.**

P1 :
main thread 10 times
Child thread 10 times.
P2 :
Child thread 10 times.
Main thread 10 times.
P3:
Main thread

Child thread
main thread
Child thread

P4:
Child thread
Main thread

Case 2:

Difference between t.start() and t.run() :

- **In the case of t.start() a new Thread will be created which is responsible for the execution of run().**
- **But in the case of t.run() no new Thread will be created and run() will be executed just like a normal method call by main Thread.**
- **In the above program if we replace t.start() with t.run() then the output is**
  Child thread 10 times
  Main thread 10 times
  Total output produced by main thread only.

Case 3 :

Importance of Thread class start() :

- **Thread class start() method is responsible for registering our Thread with Thread Scheduler and all other mandatory activities.**
  Class Thread {
  start()
  {
    - **Registering this Thread with Thread Scheduler**
    - **Perform all remaining mandatory activities.**
    - **JVM Invoke run();**

  }

- **Hence, Without executing the Thread class start() method , There is no chance of starting a new Thread, due to this Thread class start() is considered as the heart of multithreading.**

Case 4 :

Overloading of run() method :

- **Overloading of run() is possible but the Thread class start() method always calls no argument run() method.**

- **The other overloaded method, we have to call explicitly just like a normal method call.**

```
Class MyThread extends Thread
{
Public void run(){
System.out.println("no-arg");
}

Public void run(int i){
System.out.println("int arg");
}
}

Class Test {
Public static void main(String args[]){
MyThread t = new MyThread();
t.start();
t.run(10);
}
}
```

Case 5:

If we are not overriding run() method :

- **If we are not overriding run() then Thread class run() will be executed which has an empty implementation and hence we won't get any output.**

```
Class MyThread extends Threads{
}

Class Test {
```

```
Public static void main (String args[] ){
MyThread t = new MyThread();

t.start();
}
}
```

Note : It is highly recommended to override the run() method.
Otherwise don't go for multithreading.

Case 6 :

Overriding of start() method :

- **If we are overriding the start() method then our method will be executed just like a normal method call and no new Thread will be created.**

```
Class MyThread extends Threads{
        Public void start(){
                System.out.println("start ");
        }
        Public void run(){
        System.out.println("run method");
        }
}

Class Test{
Public static void main (String args[]){
MyThread t  = new MyThread();
t.start();
System.out.println("Main method");
}
}
```

- **start method**
- **main method**
- **Total output produced only by main Thread.**

Case 7

It is not recommended to override the start() method. Otherwise don't go for multithreading.

```
Class MyThread extends Thread
{
Public void start(){
Super.start();
System.out.println("Start method");
}

Public void run(){
System.out.println("run method");
}
}

Class Test {
Public static void main(String args[]){
MyThread t= new MyThread();
t.start();
}
```

Thread →
main: start method, main method
Child : run method

Output :
P1:
Start method
Main method
Run method

→ P2:
run method
Start method
Main method

→ **P3 :**
Start method
Run method
Main method

Case 8 :

Life Cycle of a Thread :

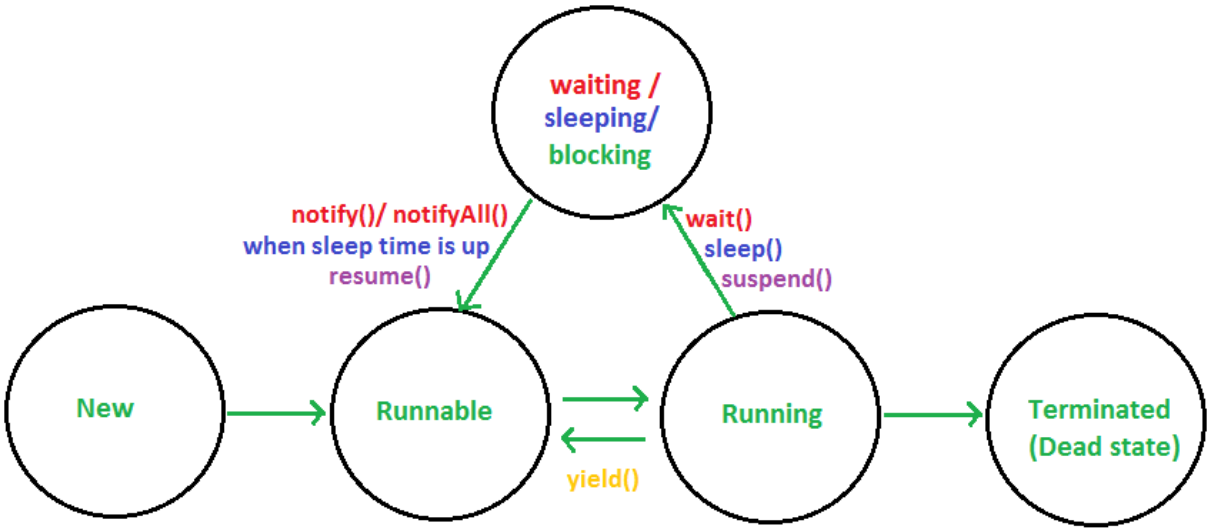


Fig. THREAD STATES

MyThread t = new MyThread();
new/born
Ready/Runnable
Running
Dead

new/born → (t.start()) → Ready/Runnable → (If TS allocates Processor) → Running → (If Run method completes) → Dead.

After starting a Thread, if we are trying to restart the same method once again then we will get a Runtime exception saying IllegalThreadStateException.

```
Thread t = new Thread();
t.start();
t.start(); // RE : IllegalThreadStateException
```

1st Approach :
MyThread→ Thread → MyRunnable

2nd Approach
Thread → Runnable → MyRunnable

- **Runnable interface present in java.lang package and defines only one abstract method i.e. run().**
- **We can consider Runnable as a Functional Interface.**

```
Public void run();

Class MyRunnable implements Runnable {
Public void run(){
for(int i=0;i<10;i++){
System.out.println("Child Thread");
}
}
}

Class ThreadDemo1{
Public static void main(String args[]){

MyRunnable r= new MyRunnable();
Thread t = new Thread(r);
t.start();

for(int i =0; i<10;i++){
System.out.println("Main Thread");
}
}
```

- **We can't tell the exact output but we will get mixed output.**

  MyRunnable r= new MyRunnable();

  Thread t1 = new Thread();
  Thread t2 = new Thread(r);

- **Case1 : t1.start(): A new thread will be created which is responsible for the execution of the Thread class run() method.**

- **case2 : t1.run(): No new Thread will be created and the Thread class run() method will be executed just like a normal method.**

- **Case3: t2.start() : A new Thread will be created which is responsible for the execution of the MyRunnable run() method.**

- **Case4: t2.run() : No new Thread will be created and MyRunnable run() method will be executed as a normal method call.**

- **Case5 : r.start() we will get a compile time error saying MyRunnable class does Not have a start capability.**

- **Case 6 : r.run() : No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.**

Q: In Which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method.
Ans: t2.start().

Q: In which of the above cases a new Thread will be created.
Ans: t1.start and t2.start()

Q: In Which of the cases MyRunnable class run() method will be executed.
And : t2.start() ,t2.run( ) and r.run().

Which is the best approach to define a Thread :

between two ways of Thread Creation ?,

- **implementing a Runnable approach is recommended.**
- **In the first approach as our class always extends Thread class, it is not possible to extend any other class due to this we are missing inheritance benefit.**
- **But in the second approach while implementing Runnable interface we can extend any other class and hence we won't miss any inheritance benefits.)**
- **Because of the above reason, implementing Runnable interface approach is recommended to use when comparing with extending Thread class.**

Example : MyThread.

Defining a Thread by implementing Runnable interface :

- **We can also define a Thread by implementing the Runnable interface.**

Thread class constructors :

- **Thread t - new Thread();**
- **Thread t = new Thread(Runnable r);**
- **Thread t = new Thread(String name);**

Getting and setting of name of the Thread :

- **Every Thread in java has some name, it may be the default name generated by JVM or Customized name provided by programmer.**

- **We can get and set the name of a Thread by using the following methods of Thread class.**

  Public final String getName();
  Public final void setname(String name);

  Example : MyThread.

- **We can get Current executing Thread object reference by using Thread.CurrentThread().**

⇒ Thread Priority :

- **Every Thread in java has some priority, it may be default provided by JVM or customized priority explicitly provided by programmer.**

- **The void range of Thread priorities is 1 to 10 but not 0 to 10.**

- **Where 1 is the least priority and 10 is the highest priority.**

- **Thread class already defines some constants to represent some standard priority.**

  Thread.MIN-PRIORITY→ 1
  Thread.MAX-PRIORITY → 10
  Thread.NORM-PRIORITY → 5

- *****Thread scheduler will use Thread priorities while allocating processors to the Thread which is having highest priority. We will get a chance for execution.**

- **If two Threads have the same priority then we cannot expect an exact execution order. It varies from JVM to JVM.**
- **We can get and set the priority of a Thread by using the following Thread class.**
  Public final int getPriority()
  Public final void setPriority(int n)
- **The allowed range is 1 to 10 by mistake if we are providing any other value we will get Runtimeexception saying IllegalArgumentException.**

Ex:
t.setPriority(10): valid
t.setPriority(100): Not valid i.e, RE.

Default priority :

- **The default priority only for the main() Thread is 5 but for all remaining Threads, the default priority inheriting from parent to child i.e, whatever priority parent Thread has the same priority will be there for the child Thread.**

Example :

```
Class MyThread extends Thread{
}

Class Test {
Public static void main(String args[]){
System.out.println(Thread.currentThred.getpriority());
Thread.currentThread.setPriority(7);//Line 2
MyThread t = new MyThread();
System.out.println(t.getPriority())
}
}
```

Example :

```
Class MyThread extends Thread
{
Public void run(){
for(int i =0; i<10;i++){
System.out.println("Child Thread ");
}
}
}

Class ThreadPriorityDemo{
Public static void main(String s){
MyThread t = new MyThread();
t.setPriority(10);// Line 1
t.start();
for(int i=0;i<10;i++){
System.out.println("main thread");
}
```

```
}
}
```

- **If we are commenting line 1 then both main and child Threads have the same priority 5 hence we cannot accept exact execution order and exact output.**
- **If we are not commenting line 1 then child Thread has priority 10 and main Thread has priority 5 hence child Thread will get the chance first followed by main Thread, in this case output is child Thread 10 times and main Thread 10 times.**
- **Note : Sometimes operating systems may not provide proper support for thread priorities.**

******* To Prevent Thread Execution :**

- **We can prevent a Thread execution by using the following methods**
- **1) yield() 2) join() 3) sleep().**

⇒ yield() :

- **This method causes a pause current executing Thread to give the chance for remaining waiting Threads of the same priority.**
- **If there is no waiting method or all waiting Threads having low priority then the same thread will continue its execution.**
- **If multiple waiting Threads have the same priority then which waiting Thread will get the chance, we cannot expect it depends on the Thread Scheduler.**
- **The Thread which is yielded when it will get a chance again we can't expect, it depends on the Thread Scheduler.**

new/Born → (t.start)Ready/ Runnable→ (If TS allocates (Thread.yield())) <– Running → Dead .

Example :

```
class MyThread extends Thread{

        Public void run(){
for(int i =0;i<10;i++){
Thread.yield(); // Line 1
System.out.println("Child Thread")
}
```

```
}
}

Class Thread Yield Demo{

Public static void main(String args[]){

MyThread t = new MyThread();
t.start();
for(int i =0;i<10;i++){
System.out.println("Main Thread");
}
}
}
```

- **If we are commenting line 1 then both Threads will be executed simultaneously and we cannot expect which Thread will be completed first.**
- **If we are not commenting on line 1 then child Threads are always called yield methods and hence main Thread will get a chance more number of times, hence completing main Thread first is always high , when compared with child Thread.**
- **Note : Some operating systems may not provide proper support for yield methods.**

⇒ join():

If a Thread(t3) wants to wait until completing some(t1) other thread then we should go for the t1.join() method.

- **For example if a Thread t1 wants to wait until completing Thread t2 and t1 has to call**

```
t2.join();
```

Public final void join() throws IE.
Public final join(long ms) throws IE.
Public final void join(long ms, int ns)throws IE

Note : Every join() throws InterruptedException, which is checked Exception, hence whenever we are using the join() method compulsory we should handle InterruptedException, either by try catch or by throws keyword otherwise we will get a compile time error.

new/born→ Ready/Runnable→ (
1.If T2 completes
2. If time expires,
3. If waits Thread got interrupted , if TS allocates processor (Waiting state(blocked state)))

→ Running → If run() method completion
→ Dead.

Case 1: Waiting for the main Thread until completing child Thread.
Case 2 : Waiting for child Thread until completing main Thread.

Case 3 : If main Thread calls join() method on child Thread object and child Thread calls join() method on main Thread object then the program will be changed like a deadlock.

Case 4 : If a Thread calls join() and the same Thread itself then the program will be changed just like a deadlock.

```
Thread.currentThread.join();
```

Sleep() :

- **If a Thread doesn't want to perform any operation for a particular amount of time i.e, just pausing is required then we should go for the sleep() method.**
-
- **public static native void sleep(long ms) throws IE**
- **public static void sleep(long ms, int ns) throws IE.**

new/born→ Ready/Runnable→ (
1.If T2 completes
2. If time expires,
3. If waits Thread got interrupted ,
if TS allocates processor (Waiting state(blocked state))) → (Thread.sleep()) ← Running → If run() method completion → Dead.

```
Class Test{
Public static void main(String args) throws IE

{
for(int i=0; i<10;i++){
System.out.println("slide " +i);
Thread.sleep(10000);
}
}
}
```

How a thread interrupts another Thread.

- **A thread can interrupt another sleeping or waiting Thread by using the interrupt() method of Thread class.**

```
public void interrupt();


class MyThread extends Thread{

Public void run(){
try{
for(int i=0;i<10;i++){
System.out.println("I am lazy Thread");
Thread.sleep(2000);
}
}catch(IE e){
System.out.println("I got interrupted");
}
}
}

class ThreadSleepDemo{
public static void main (String args[]){
MyThread t = new MyThread();
t.start();
t.interrupt(); //
System.out.println("End of main method");
}
}
```

- **In the above example main Thread interrupts child Thread**
  Output :
  End of main Thread
  I am lazy Thread
  I got interrupted

  ****Note :**

- **Whenever we are calling interrupt() method and if the target Thread not sleeping or waiting state then there is no impact of interrupt call immediately, interrupt call will wait until target thread entered into sleeping or waiting state , once target thread entered into Sleeping or waiting state, immediately interrupt will interrupt that Thread.**

- **If the target Thread never entered into a sleeping or waiting state in its lifetime then there is an impact of interrupt call, this is the only case where interrupt call will be wasted.**

```
Class MyThread extends Thread{
Public void run(){
for(int i=0;i<1000;i++){
System.out.println("I am lazy Thread " +i);
}
System.out.println("I am entering into sleep state");
try{
Thread.sleep(5000);
}
catch("InterruptedException e "){
System.out.println("I got Interrupted");
}
}
}

Class ThreadSleepDemo{
Public static void main (String args[]){
MyThread t = new MyThread();
t.start();
t.interrupt();
System.out.println("End of main Thread");
}
}
```

- **In the above example the interrupt call waited until the child Thread completed the loop 5000 times.**

Daemon Threads :

- **The Threads which are executing in the background or called Daemon Thread.**
- **Example : GarbageCollector.**

- **The main purpose of daemon Threads is to provide support for non-daemon threads.**
- **Usually daemon thread runs with low priority but based on our requirement it can run with high priority.**
- **We can check the daemon thread nature of a Thread by using isDaemon() method of Thread class.**

  public boolean isDaemon();
- **But we can change daemon nature before starting a Thread once Thread starts, We are not allowed to change daemon nature otherwise we will get a RuntimeException saying**

IllegalThreadStateException.

Default nature: by default main thread is allowed as a non-daemon thread and for all remaining Threads daemon nature will be inherited from parent to child. I.e, if the parent thread is daemon then child thread is also daemon and if the parentis non- daemon thread, child Thread is also non-daemon thread.

setDaemon(boolean status)

Synchronization :

- **MultiThreaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.**
- **So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.**
- **Java provides a way of creating threads and synchronizing their tasks by using synchronized blocks.**
- **Synchronized blocks in Java are marked with the synchronized keyword.**
- **A synchronized block in Java is synchronized on some object.**
- **All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.**
- **All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.**

```
// 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.
- The output is the same every-time we run the program.
- In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class.
- Alternatively, we could define the whole send() block as synchronized and it would produce the same result. Then we don't have to synchronize the Message object inside the run() method in the Thread Send class.

```
// An alternate implementation to demonstrate
// that we can use synchronized with the 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 makes this possible.

```
// One more alternate implementation to demonstrate
// that synchronized can be used with only a part of
// method

class Sender
{
   public void send(String msg)
   {
      synchronized(this)
      {
         System.out.println("Sending\t" + msg );
         try
         {
            Thread.sleep(1000);
         }
         catch (Exception e)
         {
            System.out.println("Thread interrupted.");
         }
         System.out.println("\n" + msg + "Sent");
      }
   }
}
```

```
// 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;
           private Thread t;
           Sender  sender;

           // Received a message object and a string
           // message to be sent
           ThreadedSend(String m,  Sender obj)
           {
              msg = m;
              sender = obj;
           }

           public void run()
           {
              // Only one thread can send a message
              // at a time.
```

```
            synchronized(sender)
            {
                // synchronizing the snd object
                sender.send(msg);
            }
        }
    }

    // Driver class
    class SyncDemo
    {
        public static void main(String args[])
        {
            Sender and = new Sender();
            ThreadedSend S1 =
                new ThreadedSend( " Hi " , snd );
            ThreadedSend S2 =
                new ThreadedSend( " Bye " , snd );

            // Start two threads of ThreadedSend type
            S1.start();
            S2.start();

            // wait for threads to end
            try
            {
                S1.join();
                S2.join();
          1)  Lambda Expressions
            }
            catch(Exception e)
            {
                System.out.println("Interrupted");
            }
        }
    }
```

Output:

```
Sending     Hi

 Hi Sent
Sending     Bye
```

Bye Sent

InterThreadCommunication.java

```
package com.multithreading;
class TestResource {
    int i = 0;
    boolean status = false;
    synchronized void put(int i) throws InterruptedException {
        while(status) {
            wait();
        }

        this.i = i;
        System.out.println("put : " + i);
        status = true;
        notify();
    }
    synchronized void get() throws InterruptedException {
        while(!status) {
            wait();
        }

        System.out.println("get : " + i);
        status = false;
        notify();
    }
}
class Producer implements Runnable {
    TestResource r1;
    Producer(TestResource r1) {
        this.r1 = r1;
```

```
        Thread t = new Thread(this, "Producer");
        t.start();
    }
    @Override
    public void run() {
        int i = 0;
        while (true) {
            try {
                r1.put(i++);
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Consumer implements Runnable {
    TestResource r1;
    Consumer(TestResource r1) {
        this.r1 = r1;
        Thread t1 = new Thread(this, "Consumer");
        t1.start();
    }
    @Override
    public void run() {
        while (true) {
            try {
                r1.get();
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class InterThreadCommunication {
    public static void main(String[] args) {
        System.out.println("main method strated !");
        TestResource r1 = new TestResource();
        Producer p1 = new Producer(r1);
        Consumer c1 = new Consumer(r1);
    }
}
```

Deadlock in java

- **Deadlock in java is a part of multithreading.**
- **Deadlock can occur in a situation when a thread is waiting for an object lock that is acquired by another thread and the second thread is waiting for an object lock that is acquired by the first thread.**
- **Since, both threads are waiting for each other to release the lock, the condition is called deadlock.**

Example of Deadlock in java

```
public class TestDeadlockExample1 {

public static void main(String[] args) {

        final String resource1 = "ratan jaiswal";

        final String resource2 = "vimal jaiswal";

        // t1 tries to lock resource1 then resource2

        Thread t1 = new Thread() {

          public void run() {

            synchronized (resource1) {

             System.out.println("Thread 1: locked resource 1");


             try { Thread.sleep(100);} catch (Exception e) {}
```

```
      synchronized (resource2) {
       System.out.println("Thread 1: locked resource 2");
      }
    }
  }
};

// t2 tries to lock resource2 then resource1
Thread t2 = new Thread() {
  public void run() {
    synchronized (resource2) {
      System.out.println("Thread 2: locked resource 2");

      try { Thread.sleep(100);} catch (Exception e) {}

      synchronized (resource1) {
        System.out.println("Thread 2: locked resource 1");
      }
    }
  }
};


t1.start();
t2.start();
```

```
    }
}
```

Output: Thread 1: locked resource 1
Thread 2: locked resource 2

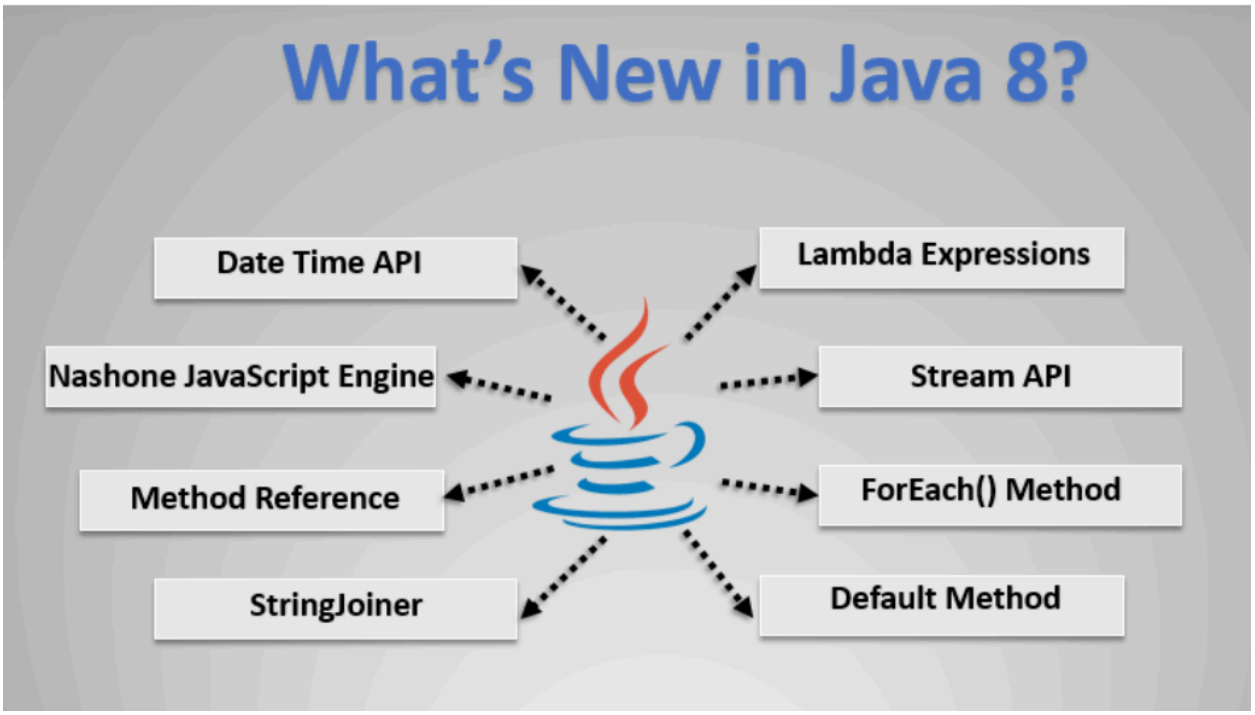
******End of Threads.******

Java 8 Features

Java 8 Introduction :

- **Java 8 introduced on 2014 March 18.**
- **The main advantage is concise coding and Enabling Functional Programming.**
  1) **Lambda Expression**
  2) **Functional Interfaces**
  3) **Default methods and static methods in interfaces**
  4) **Predefined Functional Interfaces (Predicate, Function, Consumer, Supplier…etc)**
  5) **Double colon Operator (::) - Method reference, Constructor Reference**
  6) **Streams API**
  7) **Date and Time API**
  8) **Optional Class**
  9) **Nashorn JavaScript Engine**

What's New in Java 8 ..?



Lambda Expressions

- **The main Objective of Lambda expression is : To Bring functional programming into java.**
- **What is Lambda Expression ? It is an Anonymous(Name less) function in Java**
- **What is Anonymous ? It means Nameless, without return type and without access modifier.**
- **It is Very Easiest Concept !!**

Lambda Syntax

- No arguments:
```
() -> System.out.println("Hello")
```
- One argument:
```
s -> System.out.println(s)
```
- Two arguments:
```
(x, y) -> x + y
```
- With explicit argument types:
```
(Integer x, Integer y) -> x + y
```
- Multiple statements:
```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```

6

```
//Ex1 : Normal Syntax to print hello
        public void m1() {
                System.out.println("hello");
        }
```

```
        //Lambda means no name, no return type, no modifier
Interf1 i1 = ()->System.out.println();
```

Note :If only one line No need of curly braces, if it is more lines we must need to have curly braces.

```
//Ex 2: This method can take two int values

Public void addition(int a, int b){
  System.out.println(a+b);
}

//compiler can guess automatically the Data types
(a, b) -> System.out.println(a+b);
```

```
//Ex 3

Public int squareit(int n){
  return n*n;

}

(int n)->{return n*n;};
n -> n*n;
```

Note : return statement is optional.

- If only one argument also, no need of Curly braces.

So below which one is valid..?

n-> return n*n; → Invalid (without curly braces)
n-> {return n*n;}; → valid
n-> {return n*n}; → Invalid(It must ends with semicolon)
n-> {n*n;}; → Invalid (If you use curly braces, it must be return statement)
n-> n*n; //Valid

- Without curly braces we can't use the return statement.
- Compiler will consider returned value automatically.

- Within Curly braces if we want to return some value, we must use a return statement.

Ex : 4

```
Public void hello(String s){

                return s.length();
}

s->  return s.length();
```

Functional Interfaces - FI :

Package (java.util.function)

- **Below are all interfaces only.**
- **And each interface contains Single Abstract methods (SAM),**
- **All single abstract methods of interface are Functional Interfaces.**
- **If you want to invoke Lambda Expression FI is compulsory.**
- **Inside Functional interfaces, default methods are also allowed.**
- **And static methods are also allowed.**
- **Functional Interfaces can contain only one abstract method but we can take any number of default and static methods.**

Runnable → run();
Comparable → compareTo();
Comparator → compare();
ActionListner → actionPerformed();
Callable → call();

```
package com.java8features;
@FunctionalInterface // It is an Optional
//When we have SAM then you can say it is --> FI
// It contains only one abstract methods
//but it can contains any number of default and static methods
public interface TestInterf1 {
        public void m1();
        default void m2() {
        }
        default void m5() {
        }
        static void m3() {
```

```
	}
}
```

Functional Interfaces wrt Inheritance ?

```
package com.java8features;
//@FunctionalInterface wrt Inheritance
@FunctionalInterface
public interface TestA {
	public void m1();
}
@FunctionalInterface
interface TestB extends TestA {
	public void m2();
}
```

LambdaExpression with the FI 01

```
package com.java8features;

interface interf1 {
	public void m1();
}
//To Implementing this m1 we are taking Test class.
//So with this functionality, we can implement with Lambda Expression also.
//class Test implements interf1 {
//
//	@Override
//	public void m1() {
//		System.out.println("Hello m1()");
//	}
//}
public class TestDemo01 {
	public static void main(String[] args) {
//		Test t = new Test();
//		t.m1();
		// Compiler can guess which method it is and it will call directly that m1
		// method because we have only one method
		// SO there is no need of having implemented classes to call methods from
		// Functional interfaces.
		// Note : If you want to call this LamdaExpression then FI must be required.
```

```
		// Functional Interface act as can be used to provide reference for FI.
		interf1 i1 = () -> System.out.println("Hello m1()");
		i1.m1();
	}
}
```

LambdaExpression with the FI 02

```
package com.java8features;
//LamdaExpreesion is a very Specific Concept, It's not Generic concept to use everywhere.
//We can use it only for Functional Interfaces !!
interface TestInterf {
	public void add(int a, int b);
}
//class Test2 implements TestInterf {
//
//	@Override
//	public void add(int a, int b) {
//		System.out.println("The sum of two numbers : " + (a + b));
//	}
//(int a, int b) -> System.out.println("The sum of two numbers : " + (a + b));
//}
public class TestDemo02 {
	public static void main(String[] args) {
//		TestInterf t = new Test2();
//		t.add(100, 300);
		TestInterf t = (a, b) -> System.out.println("The sum of two numbers : " + (a + b));
		t.add(100, 300);
		t.add(1000, 3000);
	}
}
```

LambdaExpression vs MultiThreading

- We can implement Thread in Two ways
- 1) Implementing Runnable interface
- 2) By Extending Thread class

```
package com.java8features;
//class MyRunnable implements Runnable {
//
```

```
//	@Override
//	public void run() {
//
//		for (int i = 0; i < 10; i++) {
//			System.out.println("Child Thread !!");
//		}
//	}
//}
//Both Thread will execute so we can't expect the exact output and we will get mixed output when we use multiple Threads
public class TestDemo04 {
	public static void main(String[] args) {
//		MyRunnable r = new MyRunnable();
		Runnable r = () -> {
			for (int i = 0; i < 10; i++) {
				System.out.println("Child Thread !!");
			}
		};
		Thread t = new Thread(r);
		t.start();
		for (int i = 0; i < 10; i++) {
			System.out.println("Main Thread !!");
		}
	}
}
```

LambdaExpression vs Collections

```
package com.java8features;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

//class TestComparator implements Comparator<Integer> {
//
//	@Override
//	public int compare(Integer o1, Integer o2) {
//
//		return (o1 < o2) ? -1 : (o1 > 02) ? 1 : 0;
//
```

```
////            if (o1 < o2) {
////                  return -1;
////            } else if (o1 > o2) {
////                  return +1;
////            } else {
////                  return 0;
////            }
//      }
//}

public class TestDemo05 {

      public static void main(String[] args) {
            ArrayList<Integer> al = new ArrayList<Integer>();

            al.add(20);
            al.add(10);
            al.add(25);
            al.add(5);
            al.add(30);
            al.add(0);
            al.add(15);

            System.out.println(al);

            Comparator<Integer> c = (o1, o2) -> (o1 < o2) ? -1 : (o1 > 02) ? 1 : 0;
//          Collections.sort(al, new TestComparator());
            Collections.sort(al, c);
            System.out.println(al);

            // In Comparator - we have only one method which is compare() method
            // int compare(Object o1, Object o2)
            // return -1 iff o1 has to come before o2
            // return 1 iff o1 has to come after o2
            // return 0 iff o1 and o2 are equal

      }

}
```

Anonymous Inner class vs Lambda Expression

- **Anonymous class means, A class without a name !!**

```
Thread t = new Thread();

Thread t= new Thread()

{

_ _ _ _ _ _ _ _ _ _ _ _ _

};
```

- **We are writing a class that extends Thread class which does not have any name**

```
Runnable r = new Runnable(); //It is Invalid

Runnable r = new Runnable()

{
  public void run(){

}

};
```

- **Lambda Expressions are not replaced for Anonymous Inner classes.**
- **Anonymous Inner class is more powerful than Lambda Expression.**
- **(Because If we have Two abstract methods in a interface we can not go and use Lambda Expressions)**
- **Anonymous Inner class != Lambda Expression.**
- **If an anonymous inner class implements an interface that contains a single abstract method then only we can replace that anonymous inner class with Lambda Expression.**
- **Anonymous Inner class can extend Normal class.**
- **Anonymous Inner class can extend Abstractclass.**
- **Anonymous Inner class can implement Interfaces which contain any number of abstract methods.**
- **Lambda Expression can implement an interface which contains a single abstract method.**

Q) Which of the following is true for Lambda Expressions !!

- **Lambda Expression contains multiple parameters, then these parameters are separated with commas.**
- **A Lambda Expression can have any number of arguments including zero number.**
- **The Lambda Expression body can contain multiple statements and we can enclose it with curly braces.**
- **If lambda Expression contains only one argument then parentheses are Optional.**

Q) Which of the valid concerns Functional Interface !!

- **Functional Interface should contain only one abstract method.**
- **Functional Interface should contain only one static method.**
- **Functional Interface can contain any number of abstract methods.**
- **Function Interface should contain only one default method.**

Q) Which of the following is true !!

- **The main Objective of Lambda expressions is to enable functional programming in java.**
- **Lambda Expressions concept applicable only for Java but not for other Languages.**
- **With Lambda expressions we can write very concise code so that readability of the application will be improved.**
- **Functional interface reference can be used to hold Lambda expressions.**

Q) Consider the code

```
interface interf{
                public int product(int a, int b);
        }
```

- **Which of the following are valid lambda expressions which implements interf interface.**

a) **Interf i = () -> return a*b;**
b) **Interf i = (a,b)-> return a*b;**
c) **Interf i = (int a, int b) -> return a*b;**
d) **Interf i = (a,b) -> a*b;**

Default methods and static methods

- **From 1.8 onwards we can take default methods in interface**
- **Until 1.7 every method present in the interface is always : public and abstract.**
- **Without any changes in implemented classes we can change or add new implementations in an interface then we can go for default.**
- **Without affecting implementation classes if we want to add new methods to the interface**

- (extending interface functionality) ⇒ default methods
- To define general utility methods inside interfaces with static.
- sum(int a, int b){
  sop(a+b);
  }

void m1();
public void m1();
abstract void m1();
public abstract void m1();

```
package com.java8features;
interface MyInterf {
	public abstract void method1();
	public abstract void method2();
	public abstract void method3();
//	void method4() {
//
//	}
	default void method5() {
		System.out.println("Hello method5 ");
	}
	static void method6() {
		System.out.println("Hello static method !!");
	}
}
public class TestDefaultDemo1 implements MyInterf {
	public static void main(String args[]) {
		TestDefaultDemo1 t = new TestDefaultDemo1();
		t.method1();
		t.method5();
		MyInterf m = new TestDefaultDemo1();
		m.method1();
		MyInterf.method6();
	}
	public static void method6() {
	}
	@Override
	public void method1() {
		System.out.println("Hello method 1");
	}
```

```
	@Override
	public void method2() {
		System.out.println("Hello method 2");
	}
	@Override
	public void method3() {
		System.out.println("Hello method 3");
	}
}
```

Related methods:

- 1.8 V : default methods + static methods
- 1.9 V : private methods

Variables :

- public static final int a =100;

Note : No changes for Variables in 1.8 !!

- Default methods | virtual Extension method | Defender method

Predefined Functional Interfaces :

Predicate

- Conditional check
- Public abstract boolean test(T t)

```
package com.java8features;
import java.util.function.Predicate;
public class TestDemo10 {
	public static void main(String[] args) {
		Predicate<Integer> p1 = i -> i % 2 == 0;
		System.out.println(p1.test(100));
		System.out.println(p1.test(101));
		// Predicate<Employee> e1 = e-> e.salary>10000 && e.age>=30;
//		System.out.println(e);
		String[] names = { "Sachin", "Dhoni", "Rohit", "Rahul", "Kohli", "Jadeja" };
		Predicate<String> p = s3 -> s3.length() > 5;
		for (String name : names) {
			if (p.test(name)) {
				System.out.print(name + " ");//sachin Jadeja
			}
		}
```

```
//Write a Predicate to check whether length of the String is > 5 or not ..?
                Predicate<String> p2 = s1 -> s1.length() > 5;
                System.out.println(p2.test("Hello"));
        }
}
```

Function

In Java 8, Function is a functional interface;
- It takes an argument (object of type T) and returns an object (object of type R).
- The argument and output can be a different type.

- apply()
- andThen()
- compose()
- identity()

```
package com.java8features;

import java.util.function.Function;
import java.util.function.Predicate;

class Student {
        String name;
        int marks;

        public Student(String name, int marks) {
                this.name = name;
                this.marks = marks;
        }
}

public class TestDemo12 {

        public static void main(String[] args) {
                // Function which takes in a number
                // and returns square of it
                Function<Integer, Integer> sq = a -> a * a;
                // Applying the function to get the result
                System.out.println(sq.apply(9));
```

```
//              // Function which takes in a number
//              // and returns half of it
                Function<Integer, Double> half = a -> a / 2.0;
                // Applying the function to get the result
                System.out.println(half.apply(25));
//
//              // Find the length of the String
                Function<String, Integer> f5 = s -> s.length();
                System.out.println(f5.apply("Srikanth"));// 8

                // Find the uppercase of the String
                Function<String, String> f1 = s -> s.toUpperCase();
                System.out.println(f1.apply("sri5java"));

                // Only the Students whose marks greater than 60
                Predicate<Student> p = s -> s.marks > 60;

                // Get all the Student with grades based on their marks !!
                Function<Student, String> f = s -> {

                        int marks = s.marks;
                        String grade = "";

                        if (marks >= 80)
                                grade = "A";
                        else if (marks >= 60)
                                grade = "B";
                        else if (marks >= 50)
                                grade = "C";
                        else if (marks >= 35)
                                grade = "D";
                        else
                                grade = "Failed";
                        return grade;
                };
//
                Student[] s = {
                                new Student("Srikanth", 100),
                                new Student("Srinu", 80),
                                new Student("Venkat", 50),
                                new Student("Nandu", 60),
                                new Student("bunny", 35),
```

```
                        new Student("Chinni", 25) };
//
            for (Student s1 : s) {
                if (p.test(s1)) {
                    System.out.println("Student name : " + s1.name);
                    System.out.println("Student marks : " + s1.marks);
                    System.out.println("Student Grade : " + f.apply(s1));
                }
            }

        }

}
```

```
package com.java8features;
import java.util.function.Function;
public class TestDemo13 {
    public static void main(String[] args) {
        // Function chaining
        Function<Integer, Integer> f1 = i -> 2 * i;
        Function<Integer, Integer> f2 = i -> i * i * i;
        System.out.println(f1.andThen(f2).apply(2));
        System.out.println(f1.compose(f2).apply(2));
    }
}
```

Consumer

- Accept some input and perform the required operation and not be required to return anything.

```
package com.java8features;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class TestDemo14 {

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

```
		Consumer<String> c1 = s -> System.out.println(s);
		c1.accept("Srikanth");
		c1.accept("SrikanthJava");

		// ***************************************************

		// Get all the Student with grades based on their marks !!
		Function<Student, String> f = s -> {
			int marks = s.marks;
			String grade = "";

			if (marks >= 80)
				grade = "A";
			else if (marks >= 60)
				grade = "B";
			else if (marks >= 50)
				grade = "C";
			else if (marks >= 35)
				grade = "D";
			else
				grade = "Failed";
			return grade;
		};

		// Only the Students whose marks greater than 60
		Predicate<Student> p = s -> s.marks > 60;

		Consumer<Student> c = s1 -> {
			System.out.println(s1.name);
			System.out.println(s1.marks);
			System.out.println(f.apply(s1));
			System.out.println();
		};

		Student[] s = { new Student("Srikanth", 100), new Student("Srinu", 80), new
Student("Venkat", 50),
				new Student("Nandu", 60), new Student("bunny", 35), new Student("bunny",
35) };
//
//		for (Student s1 : s) {
//			if (p.test(s1)) {
//				System.out.println("Student name : " + s1.name);
```

```
//                              System.out.println("Student marks : " + s1.marks);
//                              System.out.println("Student Grade : " + f.apply(s1));
//                      }
//              }

                for (Student s1 : s) {
                        if (p.test(s1)) {
                                c.accept(s1);
                        }
                }

        }

}
```

```
package com.java8features;
import java.util.function.Consumer;
class Movie {
        String name;
        Movie(String name) {
                this.name = name;
        }
}
public class TestDemo15 {
        public static void main(String[] args) {
                Consumer<Movie> c1 = m -> System.out.println(m.name + "ready to release");
                Consumer<Movie> c2 = m -> System.out.println(m.name + "Movie got flop");
                Consumer<Movie> c3 = m -> System.out.println(m.name + "Store the info in records ");
                Consumer<Movie> cc = c1.andThen(c2).andThen(c3);
                Movie m = new Movie("Hello");
                c1.accept(m);
        }
}
```

Supplier

- **Just supply my required objects and it won't take any input → Supplier**

```
package com.java8features;

import java.util.Date;
import java.util.function.Supplier;

public class TestDemo16 {

	public static void main(String[] args) {

		Supplier<Date> s = () -> new Date();
		System.out.println(s.get());

		// Which is responsible for generate Random OTP
		Supplier<String> s1 = () -> {
			String otp = "";

			for (int i = 0; i < 6; i++) {
				otp = otp + (int) (Math.random() * 10);
			}
			return otp;
		};

		System.out.println(s1.get());
	}

}
```

Summary :

These interfaces can take only one input.

Predicate → test();
Function → apply();
Consumer → accept();
Supplier → get();

Two Argument predefined Functional interfaces BiPredicate

Two input arguments

1) **Whether the given int is even or not → Predicate**
2) **Check if the sum of 2 given numbers are even or not ..? BiPredicate**

```
package com.java8features;
import java.util.function.BiPredicate;
public class BiPredicateTestDemo {
	public static void main(String[] args) {
		// Normal Predicate and will check some conditional check
		// If our requirement we have to take 2 input arguments and perform some
		// conditional check, for this requirement we should go for BiPredicate
		BiPredicate<Integer, Integer> bp = (a, b) -> (a + b) % 2 == 0;
		System.out.println(bp.test(10, 21));
		System.out.println(bp.test(10, 50));
	}
}
```

- BiFunction

```
package com.java8features;
import java.util.ArrayList;
import java.util.function.BiFunction;
class Employee3 {
	int eno;
	String name;
	Employee3(int eno, String name) {
		this.eno = eno;
		this.name = name;
	}
}
public class BiFunctionDemo {
	public static void main(String[] args) {
		ArrayList<Employee3> al = new ArrayList<Employee3>();
		BiFunction<Integer, String, Employee3> bf = (eno, name) -> new Employee3(eno, name);
		al.add(bf.apply(101, "Srikanth"));
		al.add(bf.apply(102, "Vishwa"));
		al.add(bf.apply(103, "Ganesh"));
		for (Employee3 e : al) {
			System.out.println("Employee Number:" + e.eno);
			System.out.println("Employee Name:" + e.name);
			System.out.println();
		}
	}
}
```

BiConsumer

If we want to provide two arguments and get the details.

```
package com.java8features;
import java.util.ArrayList;
import java.util.function.BiConsumer;
```

```
class Employee4 {
	String name;
	double salary;
	Employee4(String name, double salary) {
		this.name = name;
		this.salary = salary;
	}
}
public class BiConsumerDemo {
	public static void main(String[] args) {
		ArrayList<Employee4> al = new ArrayList<Employee4>();
		show(al);
		BiConsumer<Employee4, Double> bc = (e, d) -> e.salary = e.salary + d;
		for (Employee4 e : al) {
			bc.accept(e, 500.0);
		}
		for (Employee4 e : al) {
			System.out.println("Employee Name :" + e.name);
			System.out.println("Employee Salary :" + e.salary);
		}
	}
	public static void show(ArrayList<Employee4> al) {
		al.add(new Employee4("Srikanth", 10000));
		al.add(new Employee4("Venkat", 20000));
		al.add(new Employee4("Vishwa", 30000));
		al.add(new Employee4("Nag", 40000));
	}
}
```

Primitive Functional interfaces

- IntPredicate
- **Write code to check whether the given int value is even or not ?**

```
package com.java8features;
import java.util.function.IntPredicate;
//int -->(Auto Boxing) Integer(Auto UnBoxing) --> int
//You will get performance impact
public class IntPredicateTestDemo {
```

```
        public static void main(String[] args) {
                int x[] = { 0, 5, 10, 15, 20, 25, 30 };
//              Predicate<Integer> p = i -> i % 2 == 0;
                IntPredicate p = i -> i % 2 == 0;
                System.out.println(p.test(120));
                System.out.println(p.test(10));
                System.out.println(p.test(15));
                for (int x1 : x) {
                        if (p.test(x1)) {
                                System.out.println(x1);
                        }
                }
        }
}
```

- IntFunction

Can take input as int type and return any type

```
package com.java8features;
import java.util.function.IntFunction;
public class IntFunctionTestDemo {
        public static void main(String[] args) {
//              Function<Integer, Integer> f = i -> i * i;
                IntFunction<Integer> f = i -> i * i;
                System.out.println(f.apply(5));
        }
}
```

- DoubleToIntFunction :
- **Input type : double**
- **returnType : int**
- **Method : applyAsInt(double value);**

- DoubleToLongFunction :
- **Input type : double**

- **returnType : long**
- **Method : applyAsLong(double value);**

- IntToDoubleFunction :

  Method : applyAsDouble(int value)
- IntToLongFunction :

  Method : applyAsLong(int value)

  IntConsumer

- **void accept(int value) : performs this operation on a given argument.**

Note : Remaining concepts from Java8 will be covered in the classroom sessions.

Stream API

- Stream is an interface in the java.util.stream package.
- Java streams represent a pipeline through which data will flow and the functions to operate on the data.
- A pipeline in this instance consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.
- If we want to process objects from the collections then we can go for Streams.

- From the collection I want filter the elements then we can go for
  Stream s = c.stream();

- filter(Predicate <T> t); If We want to filter **some elements** from the Stream we can go for filter, It always takes a boolean condition so we can go for Predicate.

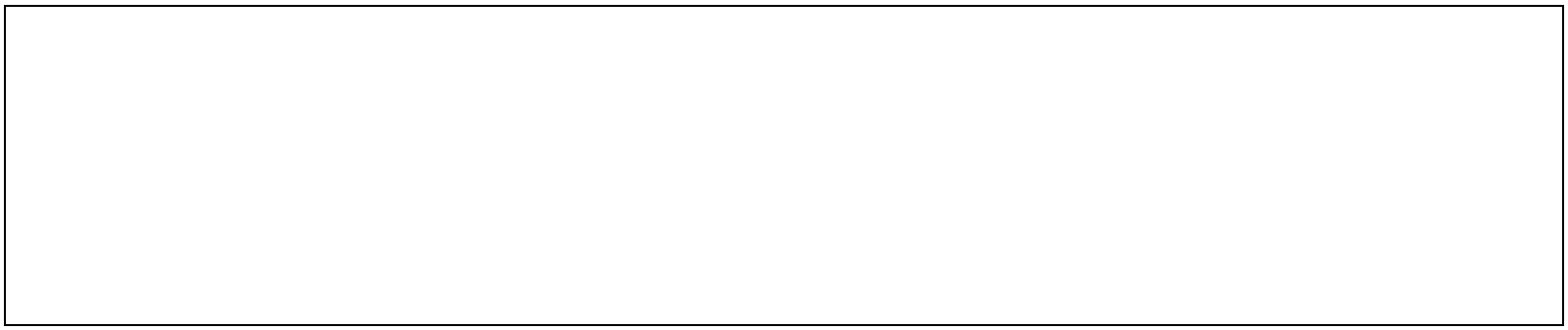
- map(Function<T,R> f); If we want to create a equivalent object for each Object from a stream then we can go for map and it always takes as input is Function<T,R>
- Once we configure it then we can go for some methods to process it.

  collect(); It is a method from a Stream
  count(); Returns the count of elements in this stream.
  distinct(); Returns a stream consisting of the distinct elements
  sorted();
  sorted(Comparator c);
  min(Comparator c);
  max(Comparator c);

  forEach();
  toArray();
  Stream.of();

  Examples :



**** End of Java 8 Features ****