

Sr. No.	Module	Detailed Contents	Hrs
1.	Fundamentals of Java Programming	Features of Object-oriented Programming, History of Java, Features of Java, JVM Architecture, Differences between C++ and Java, Data types, variable, expressions, operators, control structures, arrays	03
2.	Object and Classes	Classes, Instance variables, Methods, Constructors, Access Specifiers, Abstract Classes and Wrapper Classes, Autoboxing and Unboxing, Inheritance, Polymorphism, Method Overriding, Use of Static, final, super and this keyword, Garbage collection and finalize method, string and mutable string, Inner Classes	04
3.	Packages and Interfaces	Package concept, Creating user defined package, Access control protection, Defining interface, Implementing interface.	02
4.	Generics and Collections	Generics - Generic Class, Creating Generic Classes, Generic Methods, Bounded Type, Collections- Collections and Generics, Collection Classes-Links, Vector, Linked Lists, Maps, HashMap, WildCards, LambdaExpressions - Lambda Type Inference, Lambda Parameters, Lambda Function Body, Returning a Value From a Lambda Expression, Lambdas as Objects	05
5.	Exception Handling	Exception handling fundamentals, Exception types, Exception as objects, Exception hierarchy, Exception Keywords - Try, catch, finally, throw, throws, Creating User defined Exceptions, Assertion, Annotations	04
6.	Multi-threading	Java thread model, Life Cycle of Thread, Working with Thread class and the Runnable interface, Thread priorities, ThreadGroup class, Inter thread communication, Synchronization.	04
7.	File handling	Input streams and Output streams, FileInputStream and FileOutputStream, Binary and Character streams, Buffered Reader/ Writer, Object serialization and Deserialization.	04
8.	Event handling and GUI programming	Comparison of AWT and SWING, Applet class, Applet API hierarchy, Life cycle of Applet, Delegation Event Model, Event handling mechanisms, Swing components, Swing Component Hierarchy- Basic and Advanced Components, JApplet, Layout managers, Adapter class, Inner class.	05
9.	Database Programming	JDBC architecture, Types of drivers, Java.sql package, Establishing connectivity and working with connection interface, Working with statement interface, Working with PreparedStatement interface, Working with ResultSet interface, Working with ResultSetMetaData interface.	05
10.	Web development using Servlets	Introduction to servlets, Servlet vs CGI, Servlet API overview, Servlet Life cycle, Generic servlet, HttpServlet, ServletConfig, ServletContext, Handling HTTP Request and response –GET / POST method, request dispatching, Using cookies, Session tracking..	06
11.	Web development using JSP	Introduction to JSP, JSP Architecture, JSP Directives, JSP scripting elements, Default objects in JSP, JSP Actions, JSP with beans and JSP with Database, Error handling in JSP, Session tracking techniques in JSP, Introduction to custom tags, JSTL tags in detail	06
12.	Introduction to Spring Frameworks	Introduction to Spring Framework, Spring Architecture, Spring Aspect of Object Oriented Concepts – Join Point and Point Cuts.	04

UNIT 1 JAVA PROGRAMMING

Features of Object-oriented Programming

Object and Classes

Since Java is an object oriented language, complete java language is build on classes and object. Java is also known as a strong **Object oriented programming language(OOPS)**. OOPS is a programming approach which provides solution to problems with the help of algorithms based on real world. It uses real world approach to solve a problem. So object oriented technique offers better and easy way to write program then procedural programming model such as C, ALGOL, PASCAL.

Main Features of OOPS

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

As an object oriented language Java supports all the features given above. We will discuss all these features in detail later.

Class

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class. A class is declared using **class** keyword. A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**. Thus, the instance variables and methods are known as class members. **class** is also known as a user defined datatype.

A class and an object can be related as follows: Consider an ice tray(like of cube shape) as a class. Then ice cubes can be considered as the objects which is a blueprint of its class i.e of ice tray.

Rules for Java Class

- A class can have only public or default(no modifier) access specifier.
- It can be either abstract, final or concrete (normal class).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces **{ }**.
- Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

History of Java

James Gosling initiated Java language project in June 1991 for use in one of his many settop box projects. The language, initially called 'Oak' after an oak tree that stood outside Gosling's office, also went by the name 'Green' and ended up later being renamed as Java, from a list of random words. Sun released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms. On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL). On 8 May, 2007, Sun finished the process, making all of Java's core code free and open source, aside from a small portion of code to which Sun did not hold the copyright.

Features of Java,

The characteristics and features of java are as follows.

1) Simple

Java is a simple language because of its various features, Java Doesn't Support Pointers , Operator Overloading etc. It doesn't require unreferenced object because java support automatic garbage collection.

Java provides bug free system due to the strong memory management.

2) Object-Oriented

Object-Oriented Programming Language (OOPs) is the methodology which provide software development and maintenance by using object state, behavior , and properties.

Object Oriented Programming Language must have the following characteristics.

- 1)Encapsulation
- 2)Polymorphism
- 3)Inheritance
- 4)Abstraction

As the languages like Objective C, C++ fulfills the above four characteristics yet they are not fully object oriented languages because they are structured as well as object oriented languages.

In java everything is an Object. Java can be easily extended since it is based on the Object model

3) Secure

Java is Secure Language because of its many features it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption. Java does not support pointer explicitly for the memory.

All Program Run under the sandbox.

4) Robust

Java was created as a strongly typed language. Data type issues and problems are resolved at compile-time, and implicit casts of a variable from one type to another are not allowed.

Memory management has been simplified java in two ways. First Java does not support direct pointer manipulation or arithmetic. This make it possible for a java program to overwrite memory or corrupt data.

Second , Java uses runtime garbage collection instead of instead of freeing of memory. In languages like c++, it Is necessary to delete or free memory once the program has finished with it.

5) Platform-independent.

Java Language is platform-independent due to its hardware and software environment. Java code can be run on multiple platforms e.g. windows, Linux, sun Solaris, Mac/Os etc. Java code is compiled by the compiler and converted into byte code. This byte code is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

6) Architecture neutral

It is not easy to write an application that can be used on Windows , UNIX and a Macintosh. And its getting more complicated with the move of windows to non Intel CPU architectures.

Java takes a different approach. Because the Java compiler creates byte code instructions that are subsequently interpreted by the java interpreter, architecture neutrality is achieved in the implementation of the java interpreter for each new architecture.

7) Portable

Java code is portable. It was an important design goal of Java that it be portable so that as new architectures(due to hardware, operating system, or both) are developed, the java environment could be ported to them.

In java, all primitive types(integers, longs, floats, doubles, and so on) are of defined sizes, regardless of the machine or operating system on which the program is run. This is in direct contrast to languages like C and C++ that leave the sized of primitive types up to the compiler and developer.

Additionally, Java is portable because the compiler itself is written in Java.

8) Dynamic

Because it is interpreted , Java is an extremely dynamic language, At runtime, the java environment can extends itself by linking in classes that may be located on remote servers on a network(for example, the internet)

At runtime, the java interpreter performs name resolution while linking in the necessary classes. The Java interpreter is also responsible for determining the placement of object in memory. These two features of the Java interpreter solve the problem of changing the definition of a class used by other classes.

9) Interpreted

We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code.

The interpreter program reads the source code and translates it on the fly into computations. Thus, Java as an interpreted language depends on an interpreter program.

The versatility of being platform independent makes Java to outshine from other languages. The source code to be written and distributed is platform independent.

Another advantage of Java as an interpreted language is its error debugging quality. Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

10) High performance

For all but the simplest or most infrequently used applications, performance is always a consideration for most applications, including graphics-intensive ones such as are commonly found on the world wide web, the performance of java is more than adequate.

11) Multithreaded

Writing a computer program that only does a single thing at a time is an artificial constraint that we've lived with in most programming languages. With java, we no longer have to live with this limitation. Support for multiple, synchronized threads is built directly into the Java language and runtime environment.

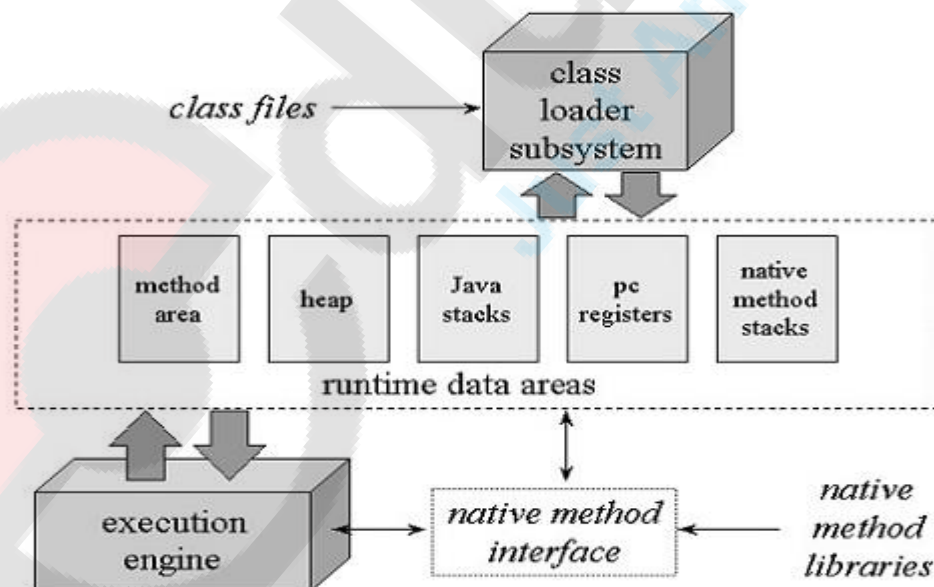
Synchronized threads are extremely useful in creating distributed, network-aware applications. Such as application may be communicating with a remote server in one thread while interacting with a user in a different thread.

12) Distributed.

Java facilitates the building of distributed application by a collection of classes for use in networked applications. By using java's URL (Uniform Resource Locator) class, an application can easily access a remote server. Classes also are provided for establishing socket-level connections.

JVM Architecture

JVM is a virtual machine or a program that provides run-time environment in which java byte code can be executed. JVMs are available for many hardware and software platforms. The use of the same byte code for all JVMs on all platforms make java platform independent.



JVM Diagram

Figure by javawithease

1. Class loader subsystem:

It is a part of JVM that care of finding and loading of class files.

2. Class/method area:

It is a part of JVM that contains the information of types/classes loaded by class loader. It also contain the static variable, method body etc.

3. Heap:

It is a part of JVM that contains object. When a new object is created, memory is allocated to the object from the heap and object is no longer referenced memory is reclaimed by garbage collector.

4. Java Stack:

It is a part of JVM that contains local variable, operands and frames. To perform an operation, Byte code instructions takes operands from the stack, operate and then return the result in to the java stack.

5. Program Counter:

For each thread JVM instance provide a separate program counter (PC) or pc register which contains the address of currently executed instruction.

6. Native method stack:

As java program can call native methods (A method written in other language like c). Native method stack contains these native methods.

7. Execution Engine:

It is a part JVM that uses Virtual processor (for execution), interpreter (for reading instructions) and JIT (Just in time) compiler (for performance improvement) to execute the instructions.

How JVM is created(Why JVM is virtual):

When JRE installed on your machine, you got all required code to create JVM. JVM is created when you run a java program, e.g. If you create a java program named FirstJavaProgram.java. To compile use – `java FirstJavaProgram.java` and to execute use – `java FirstJavaProgram`. When you run second command – `java FirstJavaProgram`, JVM is created. That's why it is virtual.

Lifetime of JVM:

When an application starts, a runtime instance is created. When application ends, runtime environment destroyed. If n no. of applications starts on one machine then n no. of runtime instances are created and every application run on its own JVM instance.

Main task of JVM:

- 1. Search and locate the required files.
- 2. Convert byte code into executable code.
- 3. Allocate the memory into ram
- 4. Execute the code.
- 5. Delete the executable code.

Differences between C++ and Java

C++	Java
C++ is a procedural and object oriented programming language.	Java is pure object oriented programming language.
C++ support structure, union, template, preprocessor, default arguments, operator overloading and pointers.	Java doesn't support all these features. Java has concept of "restricted pointers" that uses references which acts like pointers. But we can't perform arithmetic operations on it.
C++ support destructor, which is called to destroy the objects.	Java doesn't support destructor because it supports automatic garbage collection.
In C++ we can declare global variables and can define methods outside the class using scope resolution operator (::).	Java doesn't have scope resolution operator and we can declare global variables. The methods can only be defined inside the class.
C++ support goto statement. Use of goto is not considered good because it makes difficult to understand the program.	goto and const keywords are reserved in Java but they are not used.
C++ supports multiple inheritance and it can be implemented using class.	Java doesn't support multiple inheritance. Although it can be implemented using interface.
C++ is a platform dependent language. Write once, compile anywhere (WOCA).	Java is a platform independent language. Write once, run anywhere / everywhere (WORA / WORE).
C++ only uses compiler.	Java uses both interpreter and compiler.
C++ doesn't have built in thread support. We have to use third party libraries for thread support. In C++ 11, the thread support is added as a built in feature.	Java has built in thread support. There is a Thread class which is used to implement multithreading.

Data types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java –

- Primitive Data Types
- Reference/Object Data Types

1. Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the

language

and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

Byte data type is an 8-bit signed two's complement integer

Minimum value is -128 (-2^7)

Maximum value is 127 (inclusive) ($2^7 - 1$)

Default value is 0

Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.

Example: byte a = 100, byte b = -50

short

Short data type is a 16-bit signed two's complement integer

Minimum value is -32,768 (-2^{15})

Maximum value is 32,767 (inclusive) ($2^{15} - 1$)

Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer

Default value is 0.

Example: short s = 10000, short r = -20000

int

Int data type is a 32-bit signed two's complement integer.

Minimum value is -2,147,483,648 (-2^{31})

Maximum value is 2,147,483,647 (inclusive) ($2^{31} - 1$)

Integer is generally used as the default data type for integral values unless there is a concern about memory.

The default value is 0

Example: int a = 100000, int b = -200000

long

Long data type is a 64-bit signed two's complement integer

Minimum value is -9,223,372,036,854,775,808 (-2^{63})

Maximum value is 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$)

This type is used when a wider range than int is needed

Default value is 0L

Example: long a = 100000L, long b = -200000L

float

Float data type is a single-precision 32-bit IEEE 754 floating point

Float is mainly used to save memory in large arrays of floating point numbers

Default value is 0.0f

Float data type is never used for precise values such as currency

Example: float f1 = 234.5f

double

double data type is a double-precision 64-bit IEEE 754 floating point

This data type is generally used as the default data type for decimal values, generally the default choice

Double data type should never be used for precise values such as currency

Default value is 0.0d

Example: double d1 = 123.4

boolean

boolean data type represents one bit of information

There are only two possible values: true and false

This data type is used for simple flags that track true/false conditions

Default value is false

Example: boolean one = true

char

char data type is a single 16-bit Unicode character

Minimum value is '\u0000' (or 0)

Maximum value is '\uffff' (or 65,535 inclusive)

Char data type is used to store any character

Example: char letterA = 'A'

2. Reference Datatypes

Reference variables are created using defined constructors of the classes. They are used to access objects. These

variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.

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

Default value of any reference variable is null.

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

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

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without

any computation.

Literals can be assigned to any primitive type variable. For example –

`byte a = 68;`

`char a = 'A'`

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8)

number systems as well. Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using

these number systems for literals. For example –

`int decimal = 100;`

`int octal = 0144;`

`int hexa = 0x64;`

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are –

Example

`"Hello World"`

`"two\nlines"`

`"\"This is in quotes\""`

String and char types of literals can contain any Unicode characters. For example –

`char a = '\u0001';`

`String a = "\u0001";`

Java language supports few special escape sequences for String and char literals as well. They are –

Notation	Character represented
<code>\n</code>	Newline (0x0a)
<code>\r</code>	Carriage return (0x0d)
<code>\f</code>	Formfeed (0x0c)
<code>\b</code>	Backspace (0x08)
<code>\s</code>	Space (0x20)
<code>\t</code>	tab
<code>\"</code>	Double quote
<code>\'</code>	Single quote
<code>\\</code>	backslash
<code>\ddd</code>	Octal character (ddd)
<code>\uxxxx</code>	Hexadecimal UNICODE character (xxxx)

Variable

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. You must declare all variables before they can be used.

Following is the basic form of a variable declaration –

```
data type variable [ = value ][, variable [ = value ] ... ] ;
```

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

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

Example

`int a, b, c;` // Declares three ints, a, b, and c.

`int a = 10, b = 10;` // Example of initialization

`byte B = 22;` // initializes a byte type variable B.

`double pi = 3.14159;` // declares and assigns a value of PI.

`char a = 'a';` // the char variable a is initialized with value 'a'

This chapter will explain various variable types available in Java Language. There are three kinds

of variables in Java –

- Local variables
- Instance variables
- Class/Static variables

Local Variables

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

Instance Variables

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

- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Class/Static Variables

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

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

Expressions

Pending

Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –

Assume integer variable A holds 10 and variable B holds 20, then –

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –

Show Examples

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators –

Assume integer variable A holds 60 and variable B holds 13 then –

Show Examples

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language –

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –
(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	>() [] . (dot operator)	Left to right
Unary	>++ -- ! ~	Right to left
Multiplicative	>* /	Left to right
Additive	>+ -	Left to right
Shift	>>>>>><<	Left to right
Relational	>>>= <<=	Left to right
Equality	>== !=	Left to right
Bitwise AND	>&	Left to right
Bitwise XOR	>^	Left to right
Bitwise OR	>	Left to right

Logical AND	>&&	Left to right
Logical OR	>	Left to right
Conditional	?:	Right to left
Assignment	>= += -= *= /= %= >>= <<= &= ^= =	Right to left

Control Structures

The control statements are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic. [Java](#) contains the following types of control statements:

1-Selection Statements

2-Repetition Statements

3- Branching Statements

Selection statements:

1. If Statement:

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed .

Syntax:

```
if(conditional_expression){
    <statements>;
    ...;
    ...;
}
```

Example: If $n\%2$ evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence **"This is even number"** is printed on the screen.

```
int n = 10;
if(n%2 == 0){
    System.out.println("This is even number");
}
```

2. If-else Statement:

The **"if-else"** statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression){
    <statements>;
    ...;
    ...;
}
```

```
else{
<statements>;
....;
....;
}
```

Example: If $n\%2$ doesn't evaluate to 0 then else block is executed. Here $n\%2$ evaluates to 1 that is not equal to 0 so else block is executed. So **"This is not even number"** is printed on the screen.

```
int n = 11;
if(n%2 == 0){
    System.out.println("This is even number");
}
else{
    System.out.println("This is not even number");
}
```

3. Switch Statement:

This is an easier implementation to the if-else statements. The keyword **"switch"** is followed by an expression that should evaluate to byte, short, char or int primitive [data types](#), only. In a switch block there can be one or more labeled cases. The expression that creates labels for the case must be unique. The switch expression is matched with each case label. Only the matched case is executed, if no case matches then the default statement (if present) is executed.

Syntax:

```
switch(control_expression){
case expression 1:
<statement>;
case expression 2:
<statement>;
...
case expression n:
<statement>;
default:
<statement>;
} //end switch
```

Example: Here expression "day" in switch statement evaluates to 5 which matches with a case labeled "5" so code in case 5 is executed that results to output **"Friday"** on the screen.

```
int day = 5;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
```



```
break;
case 4:
System.out.println("Thrusday");
break;
case 5:
System.out.println("Friday");
break;
case 6:
System.out.println("Saturday");
break;
case 7:
System.out.println("Sunday");
break;
default:
System.out.println("Invalid entry");
break;
}
```

Repetition Statements:

1. while loop statements:

This is a looping or repeating statement. It executes a block of code or statements till the given condition is true. The expression must be evaluated to a boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

Syntax:

```
while(expression){
<statement>;
...;
...;
}
```

Example: Here expression $i \leq 10$ is the condition which is checked before entering into the loop statements. When i is greater than value 10 control comes out of loop and next statement is executed. So here i contains value "1" which is less than number "10" so control goes inside of the loop and prints current value of i and increments value of i . Now again control comes back to the loop and condition is checked. This procedure continues until i becomes greater than value "10". So this loop prints values 1 to 10 on the screen.

```
inti = 1;
//print 1 to 10
while (i<= 10){
System.out.println("Num " + i);
i++;
}
```

2. do-while loop statements:

- This is another looping statement that tests the given condition past so you can say that the do-while looping statement is a past-test loop statement. First the **do** block statements are executed then the condition given in **while** statement is checked. So in this case, even the condition is false in the first attempt, do block of code is executed at least once.

Syntax:

```
do{
    <statement>;
    ...;
    ...;
}while (expression);
```

Example: Here first do block of code is executed and current value "1" is printed then the condition $i \leq 10$ is checked. Here "1" is less than number "10" so the control comes back to do block. This process continues till value of i becomes greater than 10.

```
inti = 1;
do{
    System.out.println("Num: " + i);
    i++;
}while(i<= 10);
```

3. for loop statements:

4. This is also a loop statement that provides a compact way to iterate over a range of values. From a user point of view, this is reliable because it executes the statements within this block repeatedly till the specified conditions is true .

Syntax:

```
for (initialization; condition; increment or decrement){
    <statement>;
    ...;
    ...;
}
```

initialization: The loop is started with the value specified.

condition: It evaluates to either 'true' or 'false'. If it is false then the loop is terminated.

increment or decrement: After each iteration, value increments or decrements.

Example: Here num is initialized to value "1", condition is checked whether $\text{num} \leq 10$. If it is so then control goes into the loop and current value of num is printed. Now num is incremented and checked again whether $\text{num} \leq 10$. If it is so then again it enters into the loop. This process continues till $\text{num} > 10$. It prints values 1 to 10 on the screen.

```
for (intnum = 1; num<= 10; num++){
    System.out.println("Num: " + num);
}
```

Branching Statements:

1. Break statements:

The break statement is a branching statement that contains two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for) . It also terminates the switch statements.

Syntax:

```
break; // breaks the innermost loop or switch statement.
break label; // breaks the outermost loop in a series of nested loops.
```

Continue statements:

This is a branching statement that are used in the looping statements (while, do-while and for) to skip the current iteration of the loop and resume the next iteration .

Syntax:

continue;

Return statements:

It is a special branching statement that transfers the control to the caller of the method.

This statement is used to return a value to the caller method and terminates execution of method. This has two forms: one that returns a value and the other that can not return. the returned value type must match the return type of method.

Syntax:

return;

return values;

return; //This returns nothing. So this can be used when method is declared with void return type.

return expression; //It returns the value evaluated from the expression.

Arrays

pending

UNIT 2

OBJECTS AND CLASSES

INTRODUCTION

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts –

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

2.1 CLASSES IN JAVA

A class is a blueprint from which individual objects are created.

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **fields**
- **methods**
- **constructors**
- **blocks**
- **nested class and interface**

Syntax to declare a class:

A *class* is a template for manufacturing objects. You declare a class by specifying the **class** keyword followed by a non-reserved identifier that names it. A pair of matching open and close brace characters (**{** and **}**) follow and delimit the class's body. This syntax appears below:

1. **class** <class_name>{
2. // class body
3. field;
4. method;
5. }

Following is a sample of a class.

Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
    void barking(){  
    }  
    void hungry(){  
    }  
    void sleeping(){  
    }  
}
```

A class can contain any of the following variable types.

Local variables – Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables – Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

- **Class variables** – Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

2.1.2 OBJECTS IN JAVA

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

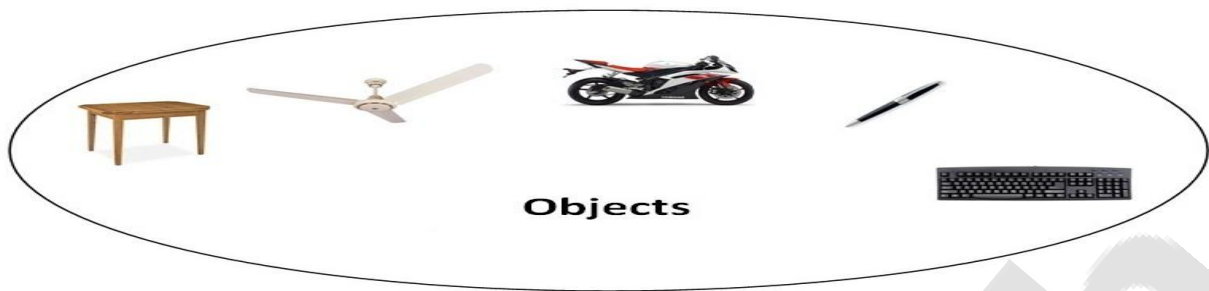
If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.



An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Object Definitions:

- Object is *a real world entity*.
- Object is *a run time entity*.
- Object is *an entity which has state and behavior*.
- Object is *an instance of a class*.

new keyword in Java

The new keyword is used to allocate memory at run time. All objects get memory in Heap memory area.

Object and Class Example: main within class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

Here, we are creating main() method inside the class.

File: Student.java

```
class Student{
    int id;//field or data member or instance variable
    String name;
    public static void main(String args[]){
        Student s1=new Student();//creating an object of Student
        System.out.println(s1.id);//accessing member through reference variable
        System.out.println(s1.name);
    }
}
```

Test it Now

Output:

0

Null

Object and Class Example: main outside class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
class Student{
    int id;
    String name;
}
class TestStudent1{
    public static void main(String args[]){
        Student s1=new Student();
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

Test it Now

Output:

0

null

3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

File: TestStudent2.java

```
class Student{
    int id;
    String name;
}
class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="Sonoo";
        System.out.println(s1.id+" "+s1.name);//printing members with a white space
    }
}
```

Test it Now

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{
    int id;
    String name;
}
class TestStudent3{
    public static void main(String args[]){
        //Creating objects
        Student s1=new Student();
        Student s2=new Student();
        //Initializing objects
        s1.id=101;
        s1.name="Sonoo";
        s2.id=102;
        s2.name="Amit";
        //Printing data
        System.out.println(s1.id+" "+s1.name);
        System.out.println(s2.id+" "+s2.name);
    }
}
```

Test it Now

Output:

101 Sonoo

102 Amit

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

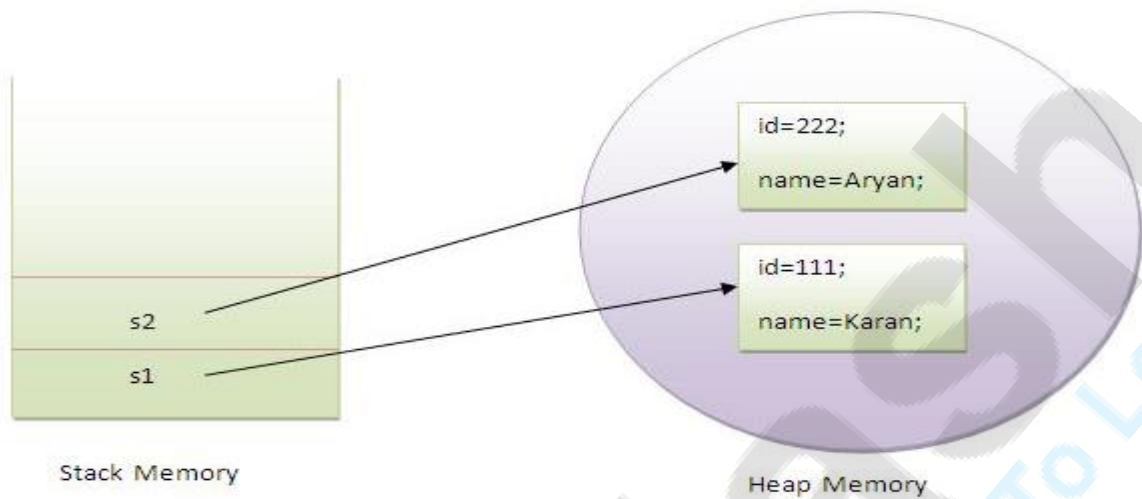
```
class Student{
    int rollno;
    String name;
    void insertRecord(int r, String n){
        rollno=r;
        name=n;
    }
    void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
    public static void main(String args[]){
        Student s1=new Student();
        Student s2=new Student();
        s1.insertRecord(111,"Karan");
        s2.insertRecord(222,"Aryan");
        s1.displayInformation();
        s2.displayInformation();
    }
}
```

Test it Now

Output:

111 Karan

222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through constructor

We will learn about constructors in java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

public class TestEmployee {
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
        e1.insert(101,"ajeet",45000);
        e2.insert(102,"irfan",25000);
        e3.insert(103,"nakul",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}
```

```
}
}
```

Test it Now

Output:

101 ajeet 45000.0

102 irfan 25000.0

103 nakul 55000.0

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
class Rectangle{
    int length;
    int width;
    void insert(int l, int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
    public static void main(String args[]){
        Rectangle r1=new Rectangle();
        Rectangle r2=new Rectangle();
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
        r2.calculateArea();
    }
}
```

Test it Now

Output:

55

45

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, anonymous object is a good approach. For example:

1. **new** Calculation();//**anonymous object**

Calling method through reference:

1. Calculation c=**new** Calculation();
2. c.fact(**5**);
Calling method through anonymous object
1. **new** Calculation().fact(**5**);
Let's see the full example of anonymous object in java.
1. **class** Calculation{
2. **void** fact(**int** n){
3. **int** fact=**1**;
4. **for**(**int** i=**1**;i<=n;i++){
5. fact=fact*i;
6. }
7. System.out.println("factorial is "+fact);
8. }
9. **public static void** main(String args[]){
10. **new** Calculation().fact(**5**);**//calling method with anonymous object**
11. }
12. }

Output:

Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

int a=**10**, b=**20**;

Initialization of reference variables:

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();**//creating two objects**

Let's see the example:

```
class Rectangle{
    int length;
    int width;
    void insert(int l,int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
        r2.calculateArea();
    }
}
```

Test it Now

Output:

55

45

Real World Example: Account

File: TestAccount.java

```
class Account{
    int acc_no;
    String name;
    float amount;
    void insert(int a,String n,float amt){
        acc_no=a;
        name=n;
        amount=amt;
    }
    void deposit(float amt){
        amount=amount+amt;
        System.out.println(amt+" deposited");
    }
    void withdraw(float amt){
        if(amount<amt){
            System.out.println("Insufficient Balance");
        }else{
            amount=amount-amt;
            System.out.println(amt+" withdrawn");
        }
    }
    void checkBalance(){System.out.println("Balance is: "+amount);}
    void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
```

```
class TestAccount{
    public static void main(String[] args){
        Account a1=new Account();
        a1.insert(832345,"Ankit",1000);
        a1.display();
        a1.checkBalance();
        a1.deposit(40000);
        a1.checkBalance();
        a1.withdraw(15000);
        a1.checkBalance();
    }
}
```

Test it Now

Output:

832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

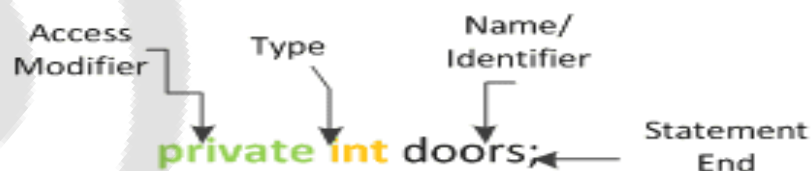
Balance is: 26000.0

2.2 INSTANCE VARIABLES

A variable which is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object (instance) is created. That is why, it is known as instance variable.

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

DECLARATION OF INSTANCE VARIABLES :



Example

```
import java.io.*;
public class Employee {
    // this instance variable is visible for any child class.
    public String name;
```

```
// salary variable is visible in Employee class only.
privatedouble salary;
// The name variable is assigned in the constructor.
publicEmployee(String empName){
    name = empName;
}
// The salary variable is assigned a value.
publicvoid setSalary(double empSal){
    salary = empSal;
}
// This method prints the employee details.
publicvoid printEmp(){
    System.out.println("name : "+ name );
    System.out.println("salary :"+ salary);
}
publicstaticvoid main(String args[]){
    Employee empOne =newEmployee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
}
}
```

This will produce the following result –

Output

name : Ransika

salary :1000.0

Rules for Instance variable

- Instance variables can use any of the four access levels
- They can be marked final
- They can be marked transient
- They cannot be marked abstract
- They cannot be marked synchronized
- They cannot be marked strictfp
- They cannot be marked native
- They cannot be marked static

2.3 METHODS IN JAVA

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

A method is like function i.e. used to expose behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

Creating Method

Considering the following example to explain the syntax of a method –

Syntax

```
public static int methodName(int a, int b) {  
    // body  
}
```

Here,

- **public static** – modifier
- **int** – return type
- **methodName** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax –

Syntax

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

The syntax shown above includes –

- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **min()**. This method takes two parameters **num1** and **num2** and returns the maximum between the two –

/** the snippet returns the minimum between two numbers */

```
public static int minFunction(int n1, int n2){  
    int min;  
    if(n1 > n2)  
        min = n2;  
    else  
        min = n1;  
    return min;  
}
```

Method Calling

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

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

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

The methods returning void is considered as call to a statement. Lets consider an example –
`System.out.println("This is tutorialspoint.com!");`

The method returning value can be understood by the following example –
`int result = sum(6,9);`

Following is the example to demonstrate how to define a method and how to call it –

Example

```
public class ExampleMinNumber {

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

This will produce the following result –

Output

Minimum value = 6

The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

Example

```

public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
            System.out.println("Rank:A1");
        } else if (points >= 122.4) {
            System.out.println("Rank:A2");
        } else {
            System.out.println("Rank:A3");
        }
    }
}

```

This will produce the following result –

Output

Rank:A1

Passing Parameters by Value

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

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

Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```

public class swappingExample {

    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b);

        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping values will be same here**");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }

    public static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

        // Swap n1 with n2
        int c = a;

```

```

    a = b;
    b = c;
    System.out.println("After swapping(Inside), a = " + a + " b = " + b);
}
}

```

This will produce the following result –

Output

Before swapping, a = 30 and b = 45
 Before swapping(Inside), a = 30 b = 45
 After swapping(Inside), a = 45 b = 30
 Now, Before and After swapping values will be same here:
 After swapping, a = 30 and b is 45

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same –

Example

```

public class ExampleOverloading {
    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        double c = 7.3;
        double d = 9.4;
        int result1 = minFunction(a, b);
        // same function name with different parameters
        double result2 = minFunction(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
    }
    // for integer
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
    // for double
    public static double minFunction(double n1, double n2) {
        double min;
        if (n1 > n2)

```

```
        min = n2;
else
    min = n1;
return min;
}
```

This will produce the following result –

Output

Minimum Value = 6
Minimum Value = 7.3

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments

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

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

Example

The following program displays all of the command-line arguments that it is called with –

```
public class CommandLine {
    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}
```

Try executing this program as shown here –

```
$java CommandLine this is a command line 200 -100
```

This will produce the following result –

Output

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

2.4 CONSTRUCTORS IN JAVA

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Types of Java Constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

1. `<class_name>(){ }`

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

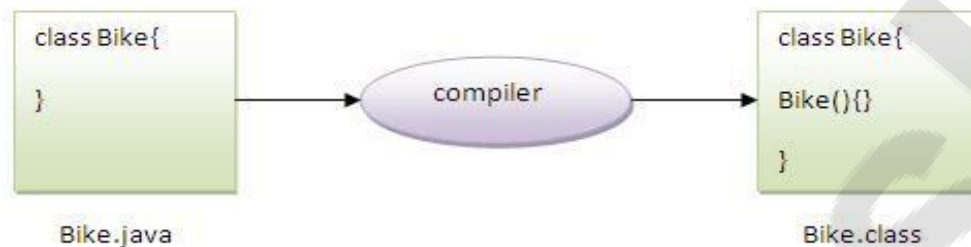
```
class Bike1{  
    Bike1(){System.out.println("Bike is created");}  
    public static void main(String args[]){
```

```
Bike1 b=new Bike1();
}
```

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of default constructor?

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

Example of default constructor that displays the default values

```
class Student3{
int id;
String name;
void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

Output:

0 null

0 null

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that has parameters is known as a parameterized constructor.

Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

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

```

class Student4{
    int id;
    String name;
    Student4(int i,String n){
        id = i;
        name = n;
    }
    void display(){System.out.println(id+ " "+name);}
    public static void main(String args[]){
        Student4 s1 = new Student4(111,"Karan");
        Student4 s2 = new Student4(222,"Aryan");
        s1.display();
        s2.display();
    }
}

```

Output:

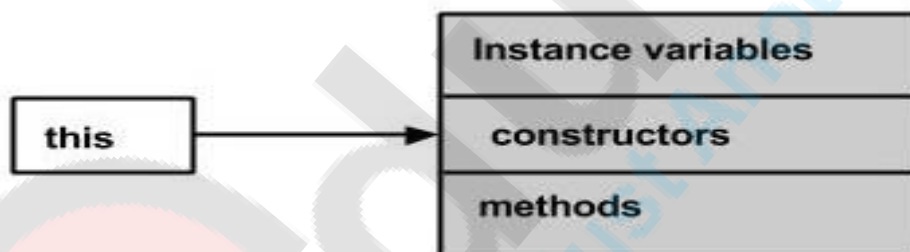
111 Karan

222 Aryan

The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

Note – The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to –

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```

class Student{
    int age;
    Student(int age){
        this.age = age;
    }
}

```

- Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```

class Student{
    int age
    Student(){

```



```
this(20);  
}  
Student(int age){  
this.age = age;  
}  
}
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1. class Student5{  
2.     int id;  
3.     String name;  
4.     int age;  
5.     Student5(int i,String n){  
6.         id = i;  
7.         name = n;  
8.     }  
9.     Student5(int i,String n,int a){  
10.        id = i;  
11.        name = n;  
12.        age=a;  
13.    }  
14.    void display(){System.out.println(id+" "+name+" "+age);}  
15.    public static void main(String args[]){  
16.        Student5 s1 = new Student5(111,"Karan");  
17.        Student5 s2 = new Student5(222,"Aryan",25);  
18.        s1.display();  
19.        s2.display();  
20.    }  
21. }
```

Output:

```
111 Karan 0  
222 Aryan 25
```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
        id = i;
        name = n;
    }
    Student6(Student6 s){
        id = s.id;
        name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student6 s1 = new Student6(111,"Karan");
        Student6 s2 = new Student6(s1);
        s1.display();
    }
}
```

```

        s2.display();
    }
}

```

Test it Now

Output:

111 Karan

111 Karan

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```

class Student7{
    int id;
    String name;
    Student7(int i,String n){
        id = i;
        name = n;
    }
    Student7(){ }
    void display(){System.out.println(id+ " "+name);}
    public static void main(String args[]){
        Student7 s1 = new Student7(111,"Karan");
        Student7 s2 = new Student7();
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
    }
}

```

Test it Now

Output:

111 Karan

111 Karan

Q) Does constructor return any value?

Ans:yes, that is current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

2.5 ACCESS SPECIFIERS or MODIFIERS

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

There are 4 types of java access modifiers:

1. default
2. public
3. private
4. protected

1) Default Access Modifier - No Keyword

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
    void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

2) Public Access Modifier - Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control –

```
public static void main(String[] arguments){  
    // ...  
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Example of public access modifier

```
//save by A.java  
package pack;  
public class A{  
    public void msg(){System.out.println("Hello");}  
}  
//save by B.java  
package mypack;  
import pack.*;  
class B{  
    public static void main(String args[]){  
        A obj = new A();  
        obj.msg();  
    }  
}
```

Output:Hello

3) Private Access Modifier – Private

The private access modifier is accessible only within class.

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

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

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
```

```
public class Simple{
public static void main(String args[]){
A obj=new A();
System.out.println(obj.data);//Compile Time Error
obj.msg();//Compile Time Error
}
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){} //private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
public static void main(String args[]){
A obj=new A();//Compile Time Error
}
}
```

Note: A class cannot be private or protected except nested class.

4) Protected Access Modifier – Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

```
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;
class B extends A{
public static void main(String args[]){
    B obj = new B();
    obj.msg();
}
}
```

Output:Hello

Understanding all java access modifiers

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

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

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");}
}
public class Simple extends A{
void msg(){System.out.println("Hello java");} //C.T.Error
public static void main(String args[]){
    Simple obj=new Simple();
    obj.msg();
}
}
```

The default modifier is more restrictive than protected. That is why there is compile time error.

2.6) ABSTRACT AND WRAPPER CLASSES

2.6.1 ABSTRACT CLASS IN JAVA

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Definition of Abstract Class:-

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example of abstract class

1. `abstract class A{ }`

abstract method

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

Example abstract method

1. `abstract void printStatus();//no body and abstract`

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
    abstract void run();
}
class Honda4 extends Bike{
    void run(){System.out.println("running safely..");}
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

```
}
}
```

Test it Now

running safely..

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}
```

//In real scenario, implementation is provided by others i.e. unknown by end user

```
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
```

```
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}
```

//In real scenario, method is called by programmer or user

```
class TestAbstraction1{
    public static void main(String args[]){
        Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape()
        method
        s.draw();
    }
}
```

Test it Now

drawing circle

Another example of abstract class in java

File: TestBank.java

```
abstract class Bank{
    abstract int getRateOfInterest();
}
class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}
```

```
class TestBank{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    }
}
```

```
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Test it Now

Rate of Interest is: 7 %

Rate of Interest is: 8 %

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

//example of abstract class that have method body

```
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}

bike is created
running safely..
gear changed
```

Rule: If there is any abstract method in a class, that class must be abstract.

```
class Bike12{
    abstract void run();
}
```

compile time error

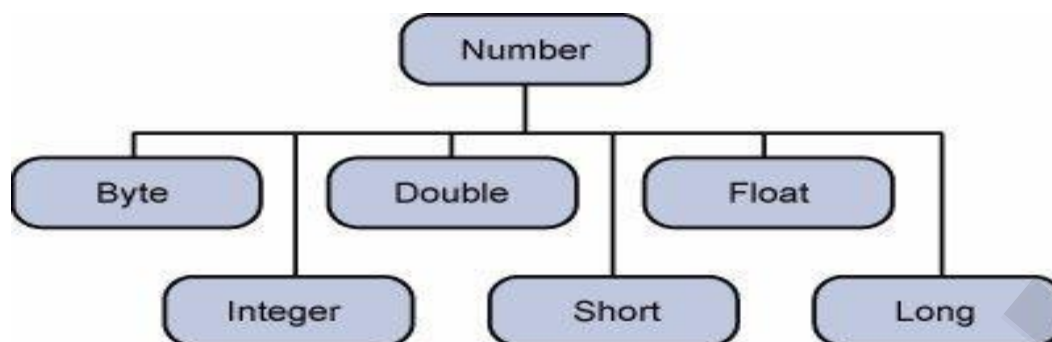
Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

2.6.2 WRAPPER CLASS IN JAVA

Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

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



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the `java.lang` package.

The eight classes of `java.lang` package are known as wrapper classes in java. The list of eight wrapper classes are given below:

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

Wrapper class Example: Primitive to Wrapper

```

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

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

```

Output:

20 20 20

Wrapper class Example: Wrapper to Primitive

```

public class WrapperExample2{

```

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

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

Output:

3 3 3

2.7) AUTOBOXING AND UNBOXING

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

- Autoboxing and Unboxing features was added in Java5.
- **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
- **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

Advantage of Autoboxing and Unboxing:

- No need of conversion between primitives and Wrappers manually so less coding is required.
- Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.
- We don't have to perform Explicit **typecasting**.
- It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.
- Auto-unboxing also allows you to mix different types of numeric objects in an expression. When the values are unboxed, the standard type conversions can be applied.

Simple Example of Autoboxing in java:

```
class BoxingExample1 {  
    public static void main(String args[]){  
        int a=50;  
        Integer a2=new Integer(a);//Boxing  
        Integer a3=5;//Boxing  
        System.out.println(a2+" "+a3);  
    }  
}
```

Output:50 5

Simple Example of Unboxing in java:

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

```
class UnboxingExample1 {  
    public static void main(String args[]){
```



```
Integer i=new Integer(50);
int a=i;

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

Test it Now

Output:50

Autoboxing and Unboxing with comparison operators

Autoboxing can be performed with comparison operators. Let's see the example of boxing with comparison operator:

```
class UnboxingExample2{
public static void main(String args[]){
Integer i=new Integer(50);

if(i<100){           //unboxing internally
System.out.println(i);
}
}
```

Test it Now

Output:50

Autoboxing and Unboxing with method overloading

In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:

- **Widening beats boxing**
- **Widening beats varargs**
- **Boxing beats varargs**

1) Example of Autoboxing where widening beats boxing

If there is possibility of widening and boxing, widening beats boxing.

```
class Boxing1{
static void m(int i){System.out.println("int");}
static void m(Integer i){System.out.println("Integer");}

public static void main(String args[]){
short s=30;
m(s);
}
}
```

Test it Now

Output:int

2) Example of Autoboxing where widening beats varargs

If there is possibility of widening and varargs, widening beats var-args.

```
class Boxing2{
    static void m(int i, int i2){System.out.println("int int");}
    static void m(Integer... i){System.out.println("Integer...");}

    public static void main(String args[]){
        short s1=30,s2=40;
        m(s1,s2);
    }
}
```

Test it Now

Output:int int

3) Example of Autoboxing where boxing beats varargs

Let's see the program where boxing beats variable argument:

```
class Boxing3{
    static void m(Integer i){System.out.println("Integer");}
    static void m(Integer... i){System.out.println("Integer...");}

    public static void main(String args[]){
        int a=30;
        m(a);
    }
}
```

Test it Now

Output:Integer

Method overloading with Widening and Boxing

Widening and Boxing can't be performed as given below:

```
class Boxing4{
    static void m(Long l){System.out.println("Long");}

    public static void main(String args[]){
        int a=30;
        m(a);
    }
}
```

Output:Compile Time Error

2.8) INHERITANCE IN JAVA

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

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

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

Why use inheritance in java

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

Syntax of Java Inheritance

1. `class` Subclass-name `extends` Superclass-name
2. {
3. `//methods and fields`
4. }

The extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {  
    ....  
}  
class Sub extends Super {  
    ....  
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using `extends` keyword, the `My_Calculation` inherits the methods `addition()` and `Subtraction()` of `Calculation` class.

Copy and paste the following program in a file with name `My_Calculation.java`

Example

```
class Calculation {  
    int z;  
    public void addition(int x, int y) {  
        z = x + y;  
        System.out.println("The sum of the given numbers:" + z);  
    }  
}
```

```

public void Subtraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given numbers:" + z);
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:" + z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}

```

Compile and execute the above code as shown below.

```
javac My_Calculation.java
```

```
java My_Calculation
```

After executing the program, it will produce the following result –

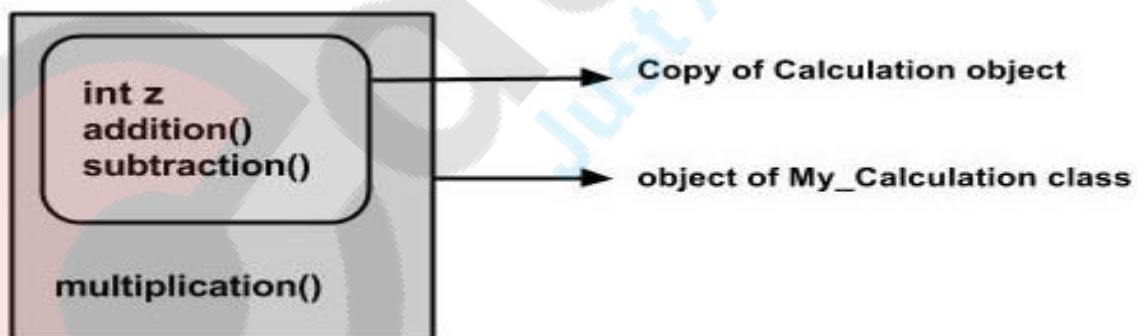
Output

The sum of the given numbers:30

The difference between the given numbers:10

The product of the given numbers:200

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass **My_Calculation**.

```
Calculation cal = new My_Calculation();
```

```
demo.addition(a, b);  
demo.Subtraction(a, b);
```

Note – A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a string value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

```
class Superclass{  
    int age;  
  
    Superclass(int age){  
        this.age = age;  
    }  
  
    public void getAge(){  
        System.out.println("The value of the variable named age in super class is: "+age);  
    }  
}  
  
public class Subclass extends Superclass{  
    Subclass(int age){
```

```
super(age);
}
public static void main(String argd[]){
Subclass s = new Subclass(24);
    s.getAge();
}
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
java Subclass
```

On executing the program, you will get the following result –

Output

The value of the variable named age in super class is: 24

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say –

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal{
}

class Mammal extends Animal{
}
```



```

class Reptile extends Animal {
}
public class Dog extends Mammal {
    public static void main(String args[]){
        Animal a = new Animal();
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}

```

This will produce the following result –

Output

```

true
true
true

```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```

public interface Animal {
}
public class Mammal implements Animal {
}
public class Dog extends Mammal {
}

```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```

interface Animal {}
class Mammal implements Animal {}

public class Dog extends Mammal {
    public static void main(String args[]){
        Mammal m = new Mammal();
        Dog d = new Dog();
        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}

```

This will produce the following result –

Output

true
true
true

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

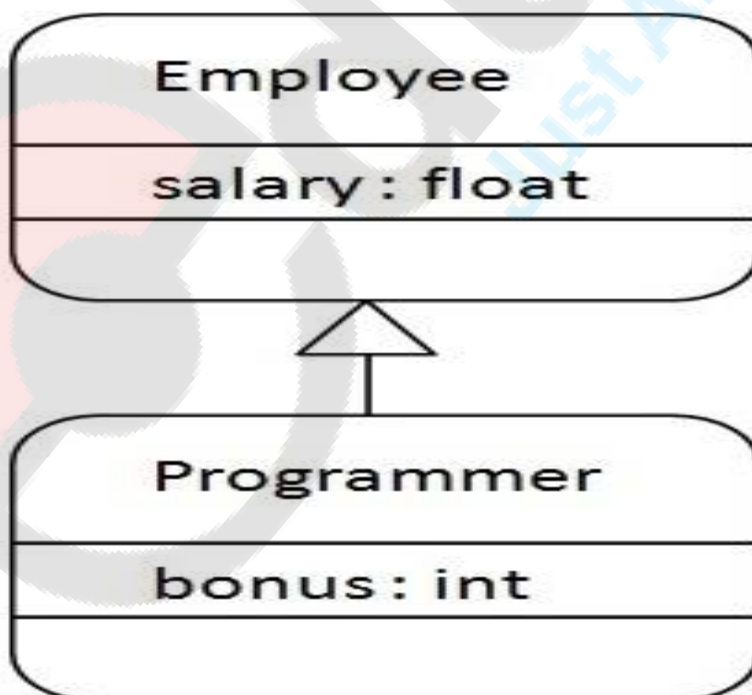
Lets look into an example –

Example

```
public class Vehicle {}  
public class Speed {}  
public class Van extends Vehicle {  
    private Speed sp;  
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Java Inheritance Example

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

```
class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```

Test it Now

Programmer salary is:40000.0

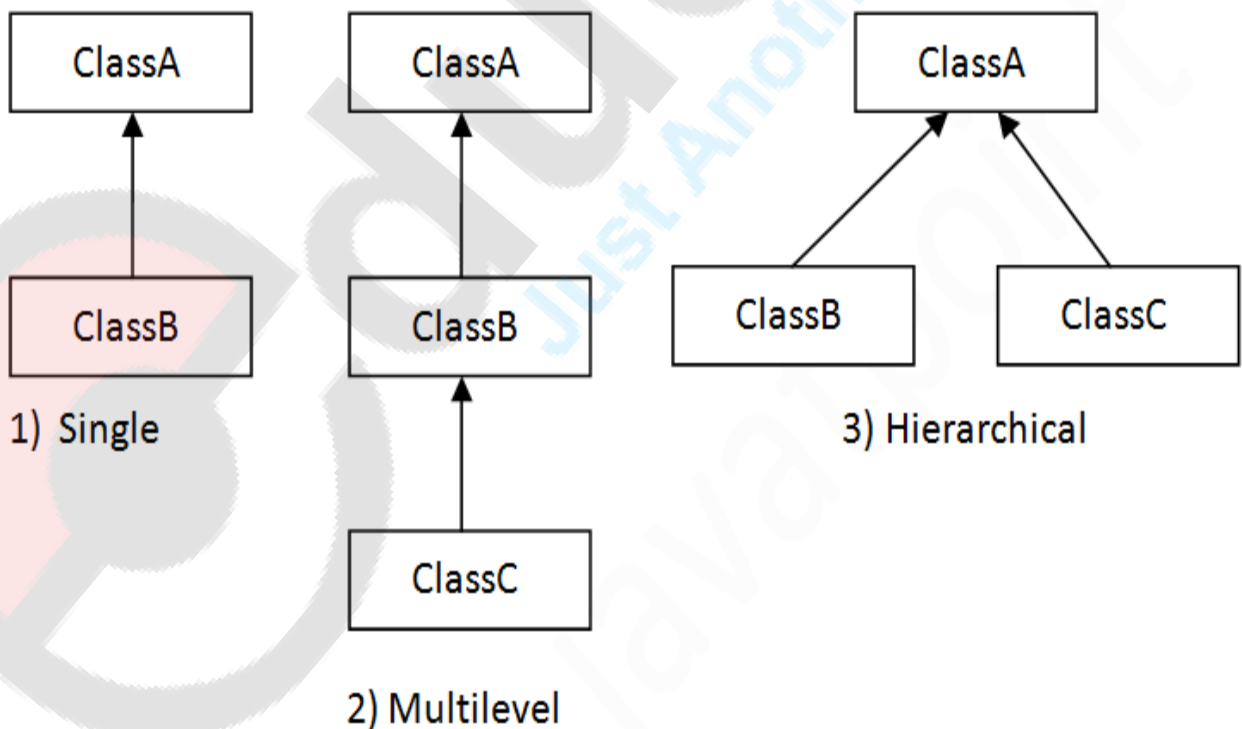
Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

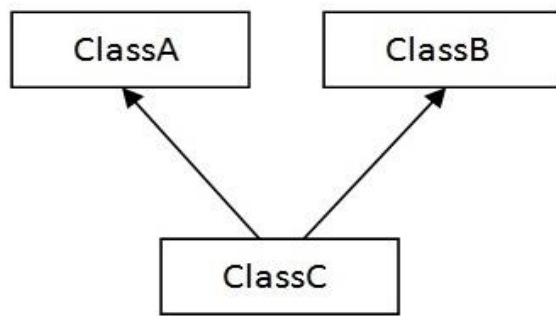
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

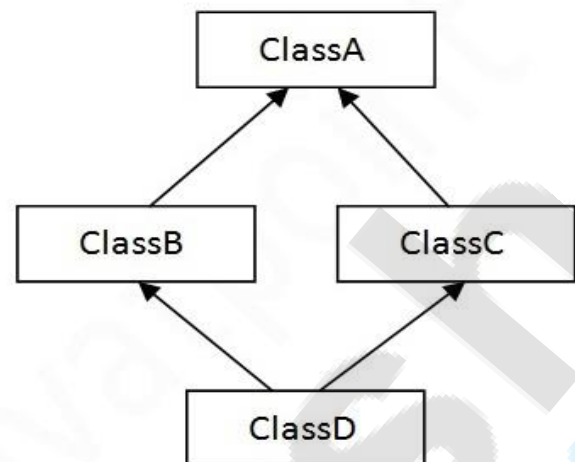


Note: Multiple inheritance is not supported in java through class.

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



4) Multiple



5) Hybrid

Single Inheritance Example

File: TestInheritance.java

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
  
```

Output:
barking...
eating...

Multilevel Inheritance Example

File: TestInheritance2.java

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
  
```

```
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

File: TestInheritance3.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

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

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

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
```

```
Public Static void main(String args[]){
C obj=new C();
obj.msg();//Now which msg() method would be invoked?
```

```
}  
}
```

OUTPUT:-

Compile Time Error

2.9) POLYMORPHISM IN JAVA

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms. So it is the ability of an object to take on many forms.

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

There are two types of polymorphism in java: **compile time polymorphism** and **runtime polymorphism**. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
    void run(){System.out.println("running");}
}
class Splender extends Bike{
    void run(){System.out.println("running safely with 60km");}

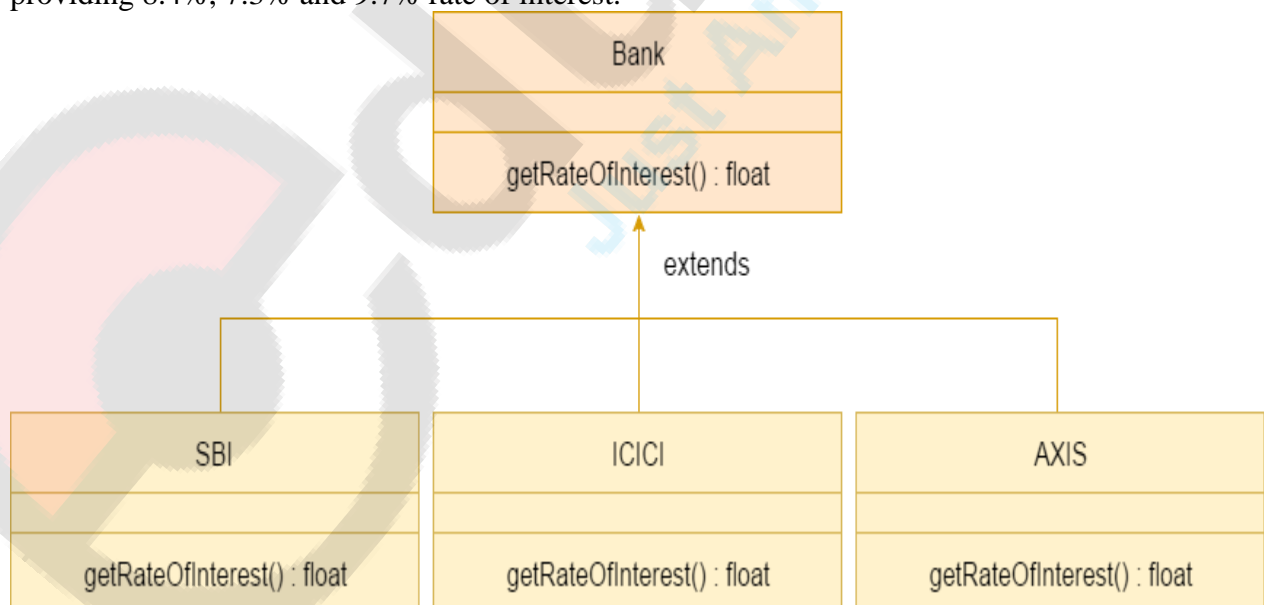
    public static void main(String args[]){
        Bike b = new Splender();//upcasting
        b.run();
    }
}
```

Test it Now

Output:running safely with 60km.

Java Runtime Polymorphism Example: Bank

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks are providing 8.4%, 7.3% and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Test it Now

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

Java Runtime Polymorphism Example: Shape

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
```

```
}
}
```

Test it Now

Output:

drawing rectangle...

drawing circle...

drawing triangle...

Java Runtime Polymorphism Example: Animal

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
class Cat extends Animal{
void eat(){System.out.println("eating rat...");}
}
class Lion extends Animal{
void eat(){System.out.println("eating meat...");}
}
class TestPolymorphism3{
public static void main(String[] args){
Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();
}}
```

Test it Now

Output:

eating bread...

eating rat...

eating meat...

Java Runtime Polymorphism with Data Member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```
class Bike{
int speedlimit=90;
}
class Honda3 extends Bike{
int speedlimit=150;
```

```
public static void main(String args[]){  
    Bike obj=new Honda3();  
    System.out.println(obj.speedlimit);//90  
}
```

Test it Now

Output:
90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{  
    void eat(){System.out.println("eating");}  
}  
class Dog extends Animal{  
    void eat(){System.out.println("eating fruits");}  
}  
class BabyDog extends Dog{  
    void eat(){System.out.println("drinking milk");}  
    public static void main(String args[]){  
        Animal a1,a2,a3;  
        a1=new Animal();  
        a2=new Dog();  
        a3=new BabyDog();  
        a1.eat();  
        a2.eat();  
        a3.eat();  
    }  
}
```

Test it Now

Output:
eating
eating fruits
drinking Milk

2.10) METHOD OVERRIDING

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level.
For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.

- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

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

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

Test it Now

Output:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```

class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
}

```

```

public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}

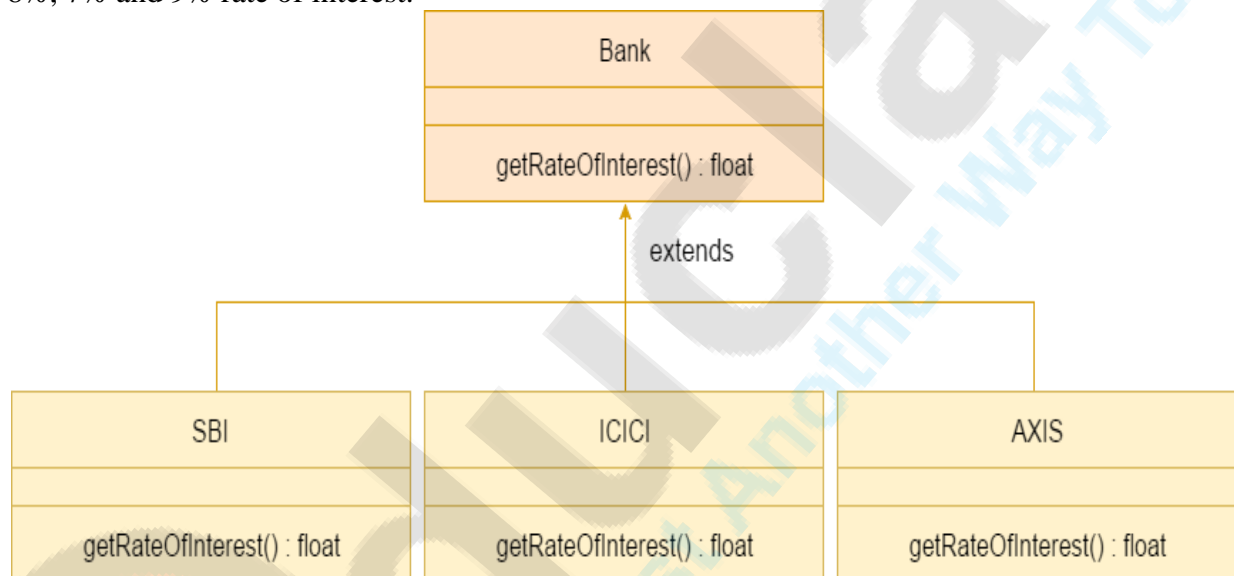
```

Test it Now

Output: Bike is running safely

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



```

class Bank{
int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
}
}

```



```

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}

```

Test it Now

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

Can we override static method?

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

Why we cannot override static method?

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

Can we override java main method?

No, because main is a static method.

Difference between method Overloading and Method Overriding in java

No.	Method Overloading	Method Overriding
1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

2.11.1) STATIC KEYWORD IN JAVA

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

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

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

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

Advantage of static variable

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

Understanding problem without static variable

1. `class Student{`
2. `int rollno;`
3. `String name;`
4. `String college="ITS";`
5. `}`

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

Java static property is shared to all objects.

Example of static variable

//Program of static variable

```
class Student8{
    int rollno;
    String name;
    static String college = "ITS";

    Student8(int r,String n){
        rollno = r;
        name = n;
    }

    void display (){System.out.println(rollno+" "+name+" "+college);}

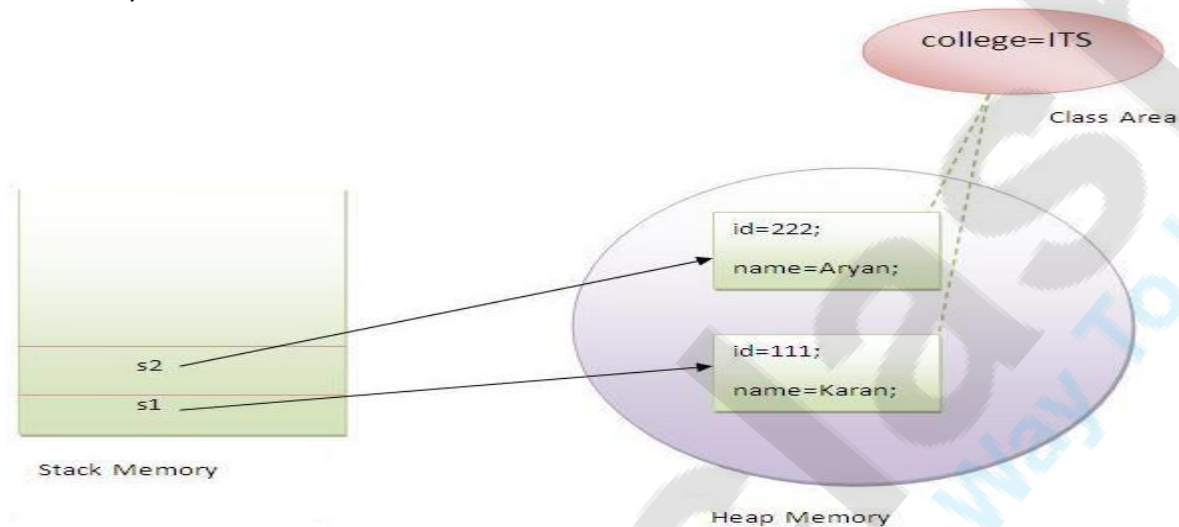
    public static void main(String args[]){
        Student8 s1 = new Student8(111,"Karan");
    }
}
```

```
Student8 s2 = new Student8(222,"Aryan");
```

```
s1.display();
s2.display();
}
}
```

Test it Now

Output:111 Karan ITS
222 Aryan ITS



Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
class Counter{
    int count=0;//will get memory when instance is created
```

```
    Counter(){
        count++;
        System.out.println(count);
    }
```

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

```
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
```

```
    }
}
```

Test it Now

Output:1

1
1

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter2{
static int count=0;//will get memory only once and retain its value
```

```
Counter2(){
count++;
System.out.println(count);
}
```

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

```
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
```

```

}
```

Test it Now

Output:1

2

3

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

//Program of changing the common property of all objects(static field).

```
class Student9{
int rollno;
String name;
static String college = "ITS";
static void change(){
college = "BBDIT";
}
Student9(int r, String n){
rollno = r;
name = n;
}
void display (){System.out.println(rollno+" "+name+" "+college);}
public static void main(String args[]){
Student9.change();
Student9 s1 = new Student9 (111,"Karan");
Student9 s2 = new Student9 (222,"Aryan");
```

```

Student9 s3 = new Student9 (333, "Sonoo");
s1.display();
s2.display();
s3.display();
}
}

```

Test it Now

Output:111 Karan BBDIT
 222 Aryan BBDIT
 333 Sonoo BBDIT

Another example of static method that performs normal calculation

//Program to get cube of a given number by static method

```

class Calculate{
    static int cube(int x){
        return x*x*x;
    }
    public static void main(String args[]){
        int result=Calculate.cube(5);
        System.out.println(result);
    }
}

```

Test it Now

Output:125

Restrictions for static method

There are two main restrictions for the static method. They are:

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

```

class A{
    int a=40;//non static
    public static void main(String args[]){
        System.out.println(a);
    }
}

```

Test it Now

Output:Compile Time Error

Q) why java main method is static?

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

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
class A2{
    static{System.out.println("static block is invoked");}
    public static void main(String args[]){
        System.out.println("Hello main");
    }
}
```

Test it Now

Output:static block is invoked
Hello main

Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
class A3{
    static{
        System.out.println("static block is invoked");
        System.exit(0);
    }
}
```

Test it Now

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

Output>Error: Main method not found in class A3, please define the main method as:

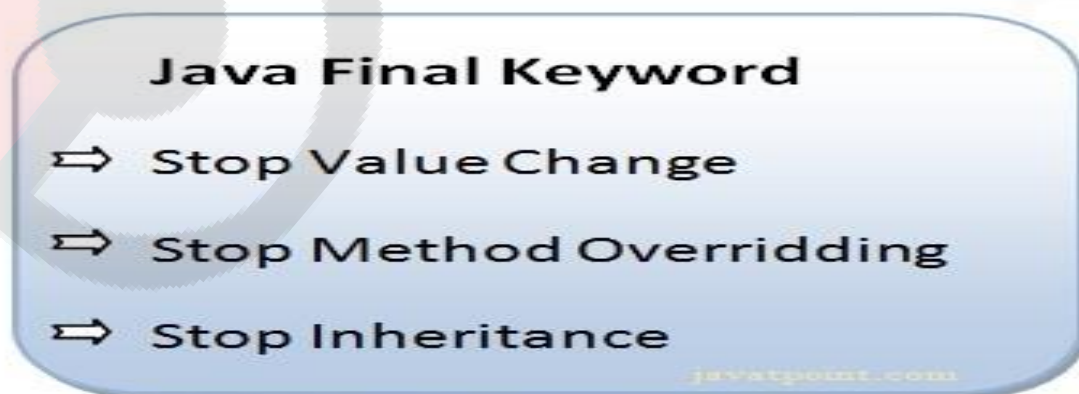
public static void main(String[] args)

2.11.2) FINAL KEYWORD IN JAVA

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

1. variable
2. method
3. class

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



1) Java final variable

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

Example of final variable

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

```

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

```

Test it Now

Output:Compile Time Error

2) Java final method

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

Example of final method

```

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

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

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

```

Test it Now

Output:Compile Time Error

3) Java final class

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

Example of final class

```

final class Bike{}

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

    public static void main(String args[]){
        Honda1 honda= new Honda1();
        honda.run();
    }
}

```

Test it Now

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
    final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}
```

Test it Now

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}
```

Test it Now

Output:70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
    static final int data;//static blank final variable
    static{ data=50;}
    public static void main(String args[]){
        System.out.println(A.data);
    }
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
    int cube(final int n){
        n=n+2;//can't be changed as n is final
        n*n*n;
    }
    public static void main(String args[]){
        Bike11 b=new Bike11();
        b.cube(5);
    }
}
```

Test it Now

Output:Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited.

2.11.3 SUPER KEYWORD IN JAVA

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
    String color="white";
}
class Dog extends Animal{
    String color="black";
    void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
```

```

}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}

```

Test it Now

Output:

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}

```

Test it Now

Output:

eating...

barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```

class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{

```

```

Dog(){
    super();
    System.out.println("dog is created");
}
}
class TestSuper3{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}

```

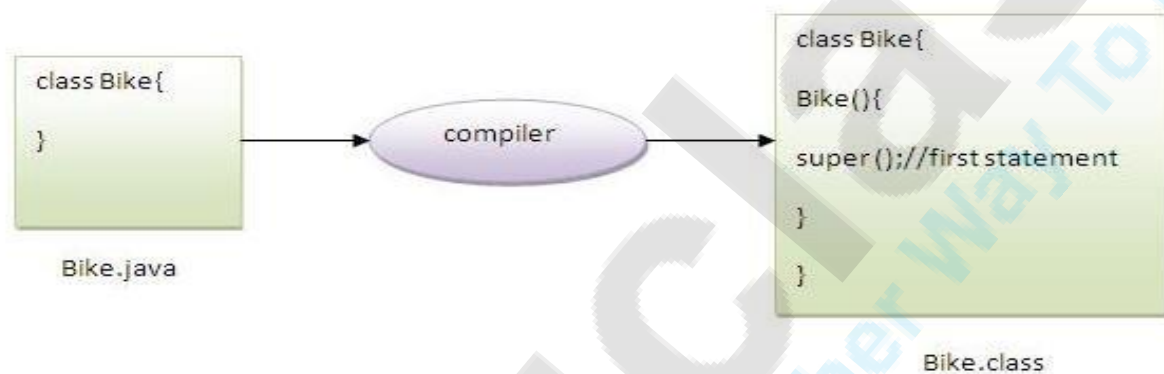
Test it Now

Output:

animal is created

dog is created

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```

class Animal{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
        System.out.println("dog is created");
    }
}
class TestSuper4{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}

```

Test it Now

Output:

animal is created

dog is created

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are

using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}
class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+ " "+name+ " "+salary);}
}
class TestSuper5{
    public static void main(String[] args){
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
    }
}
```

Test it Now

Output:

1 ankit 45000

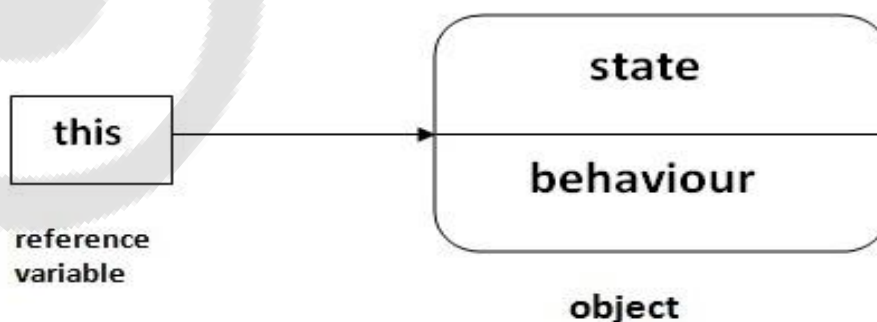
2.11.4 THIS KEYWORD IN JAVA

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.



1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        rollno=rollno;
        name=name;
        fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Test it Now

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
    }
}
```

```
s2.display();
}}
```

Test it Now

Output:

111 ankit 5000

112 sumit 6000

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int r,String n,float f){
        rollno=r;
        name=n;
        fee=f;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis3{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Test it Now

Output:

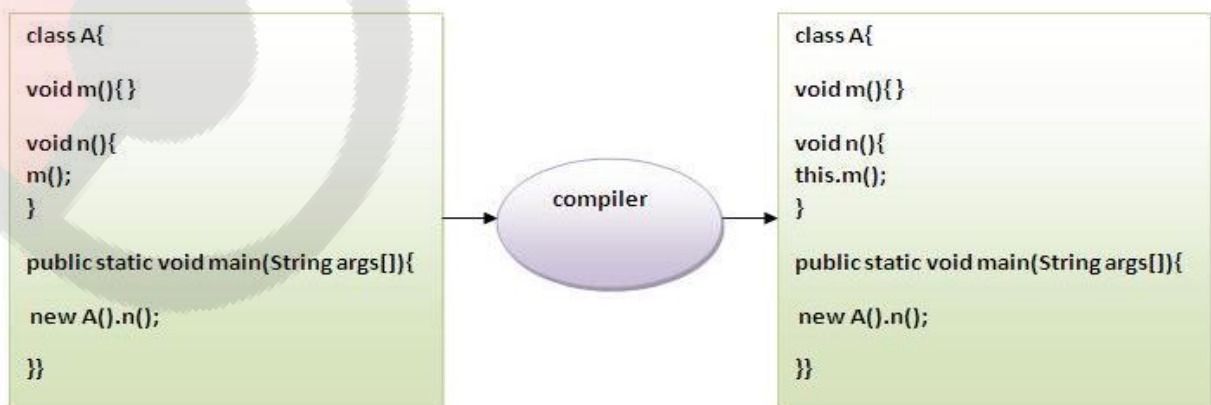
111 ankit 5000

112 sumit 6000

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Test it Now

Output:

hello n

hello m

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

hello a

10

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Test it Now

Output:

5

hello a

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
    int rollno;
    String name,course;
    float fee;
    Student(int rollno,String name,String course){
        this.rollno=rollno;
        this.name=name;
        this.course=course;
    }
    Student(int rollno,String name,String course,float fee){
        this(rollno,name,course);//reusing constructor
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit","java");
        Student s2=new Student(112,"sumit","java",6000f);
        s1.display();
        s2.display();
    }
}
```

Test it Now

Output:

111 ankit java null

112 sumit java 6000

Rule: Call to this() must be the first statement in constructor.

```
class Student{
    int rollno;
    String name,course;
    float fee;
    Student(int rollno,String name,String course){
        this.rollno=rollno;
        this.name=name;
        this.course=course;
    }
    Student(int rollno,String name,String course,float fee){
        this.fee=fee;
        this(rollno,name,course);//C.T.Error
    }
    void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
```

```
public static void main(String args[]){
    Student s1=new Student(111,"ankit","java");
    Student s2=new Student(112,"sumit","java",6000f);
    s1.display();
    s2.display();
}
```

Test it Now

Compile Time Error: Call to this must be first statement in constructor

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
    void m(S2 obj){
        System.out.println("method is invoked");
    }
    void p(){
        m(this);
    }
    public static void main(String args[]){
        S2 s1 = new S2();
        s1.p();
    }
}
```

Test it Now

Output:
method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
    A4 obj;
    B(A4 obj){
        this.obj=obj;
    }
    void display(){
        System.out.println(obj.data);//using data member of A4 class
    }
}
class A4{
    int data=10;
    A4(){
        B b=new B(this);
        b.display();
    }
    public static void main(String args[]){
        A4 a=new A4();
    }
}
```

```
}
}
```

Test it Now

Output:10

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){
    return this;
}
```

Example of this keyword that you return as a statement from the method

```
class A{
    A getA(){
        return this;
    }
    void msg(){System.out.println("Hello java");}
}
class Test1{
    public static void main(String args[]){
        new A().getA().msg();
    }
}
```

Test it Now

Output:

Hello java

2.12) JAVA GARBAGE COLLECTION

In java, garbage means unreferenced objects.

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

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

How Garbage Collection Really Works

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:

- Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 2.1). The next allocation starts at this offset and claims the next portion of the array.
- When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

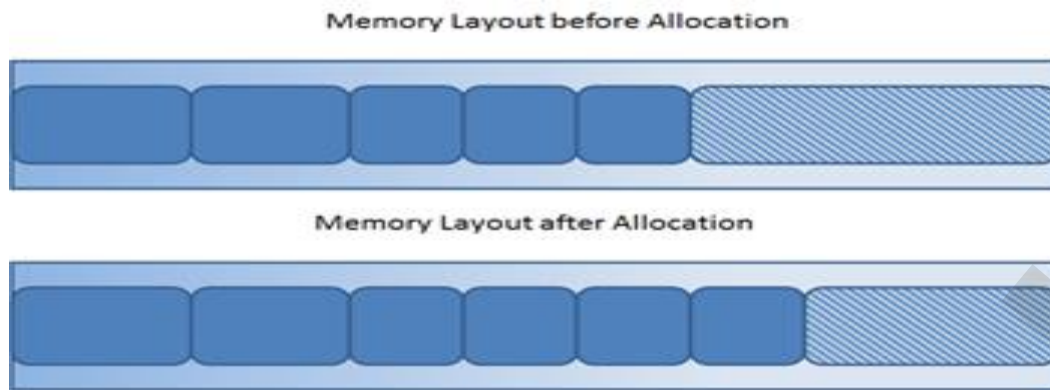


Figure 2.1: New objects are simply allocated at the end of the used heap.

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive.

Advantage of Garbage Collection

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

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

1. **new** Employee();

2.13.2) JAVA FINALIZE METHOD

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){ }

Garbage Collector Method :-**gc() method**

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

1. `public static void gc(){ }`

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{
    public void finalize(){System.out.println("object is garbage collected");}
    public static void main(String args[]){
        TestGarbage1 s1=new TestGarbage1();
        TestGarbage1 s2=new TestGarbage1();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

OUTPUT:-

```
object is garbage collected
object is garbage collected
```

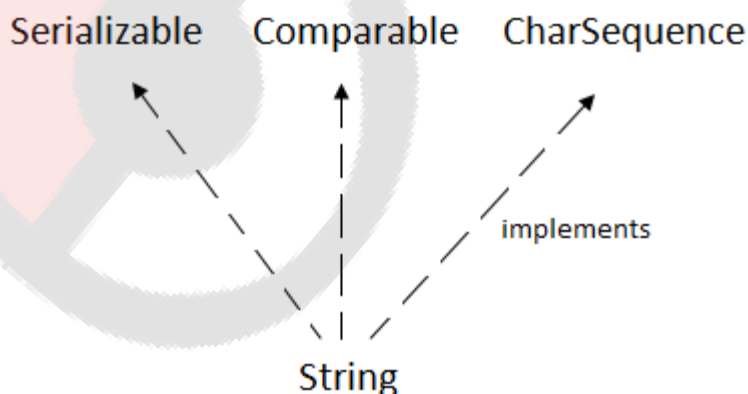
2.14 STRING AND MUTABLE STRING**1) STRING IN JAVA**

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

1. `char[] ch={'j','a','v','a','t','p','o','i','n','t'};`
2. `String s=new String(ch);`
is same as:
1. `String s="javatpoint";`

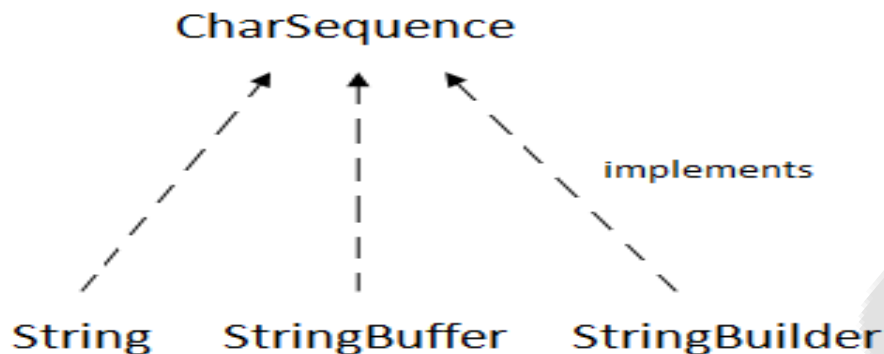
Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.Stringclass implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

We will discuss about immutable string later. Let's first understand what is string in java and how to create the string object.

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

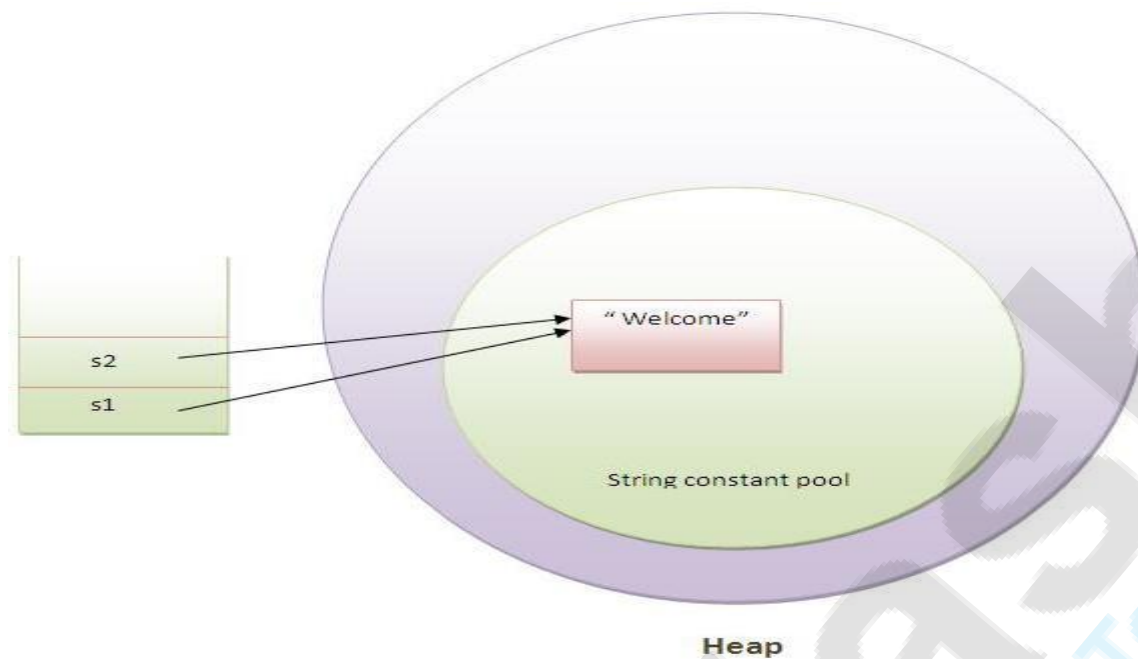
1) String Literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as string constant pool.

Why java uses concept of string literal?

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

2) By new keyword

1. `String s=new String("Welcome");` //creates two objects and one reference variable

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

Java String Example

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

Test it Now

```
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	<u>char charAt(int index)</u>	returns char value for the particular index
2	<u>int length()</u>	returns string length
3	<u>static String format(String format, Object... args)</u>	returns formatted string
4	<u>static String format(Locale l, String format, Object... args)</u>	returns formatted string with given locale
5	<u>String substring(int beginIndex)</u>	returns substring for given begin index
6	<u>String substring(int beginIndex, int endIndex)</u>	returns substring for given begin index and end index
7	<u>boolean contains(CharSequence s)</u>	returns true or false after matching the sequence of char value
8	<u>static String join(CharSequence delimiter, CharSequence... elements)</u>	returns a joined string
9	<u>static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)</u>	returns a joined string
10	<u>boolean equals(Object another)</u>	checks the equality of string with object
11	<u>boolean isEmpty()</u>	checks if string is empty
12	<u>String concat(String str)</u>	concatinates specified string
13	<u>String replace(char old, char new)</u>	replaces all occurrences of specified char value
14	<u>String replace(CharSequence old, CharSequence new)</u>	replaces all occurrences of specified CharSequence
15	<u>static String equalsIgnoreCase(String another)</u>	compares another string. It doesn't check case.
16	<u>String[] split(String regex)</u>	returns splitted string matching regex

17	<u>String[] split(String regex, int limit)</u>	returns splitted string matching regex and limit
18	<u>String intern()</u>	returns interned string
19	<u>int indexOf(int ch)</u>	returns specified char value index
20	<u>int indexOf(int ch, int fromIndex)</u>	returns specified char value index starting with given index
21	<u>int indexOf(String substring)</u>	returns specified substring index
22	<u>int indexOf(String substring, int fromIndex)</u>	returns specified substring index starting with given index
23	<u>String toLowerCase()</u>	returns string in lowercase.
24	<u>String toLowerCase(Locale l)</u>	returns string in lowercase using specified locale.
25	<u>String toUpperCase()</u>	returns string in uppercase.
26	<u>String toUpperCase(Locale l)</u>	returns string in uppercase using specified locale.
27	<u>String trim()</u>	removes beginning and ending spaces of this string.
28	<u>static String valueOf(int value)</u>	converts given type into string. It is overloaded.

2) MUTABLE STRING

A string that can be modified or changed is known as mutable string. **StringBuffer** and **StringBuilder** classes are used for creating mutable string.

[A] Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer class :

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Methods of the StringBuffer class:-**1) StringBuffer append() method**

The append() method concatenates the given argument with this string.

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

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello ");
        sb.insert(1,"Java");//now original string is changed
        System.out.println(sb);//prints HJavaello
    }
}
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

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

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello");
        sb.delete(1,3);
        System.out.println(sb);//prints Hlo
    }
}
```

5) StringBuffer reverse() method

The reverse() method of StringBuffer class reverses the current string.

```

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

```

2.15 INNER CLASSES IN JAVA:-

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```

class Java_Outer_class{
    //code
    class Java_Inner_class{
        //code
    }
}

```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

- 1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.
- 2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- 3) **Code Optimization:** It requires less code to write.

Inner Classes Example:-

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

Example :-

```

class Outer_Demo{
    int num;

    // inner class
    private class Inner_Demo{
        public void print(){
            System.out.println("This is an inner class");
        }
    }

    // Accessing the inner class from the method within
    void display_Inner(){

```

```
Inner_Demo inner = new Inner_Demo();
    inner.print();
}
}

public class My_class {

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

        // Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```

Here you can observe that **Outer_Demo** is the outer class, **Inner_Demo** is the inner class, **display_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

If you compile and execute the above program, you will get the following result –

Output

This is an inner class.

UNIT 3

PACKAGES AND INTERFACES

3.1) PACKAGE CONCEPT

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Here, we will have the detailed learning of creating and using user-defined packages.

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are –

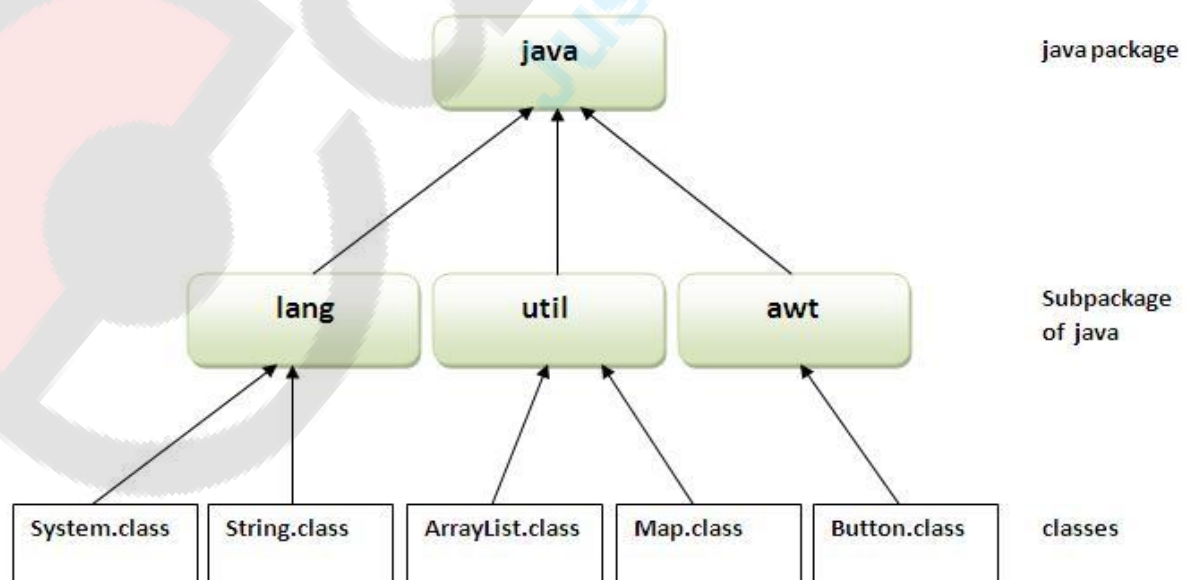
- **java.lang** – bundles the fundamental classes
- **java.io** – classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

//save as Simple.java

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

How to compile java package

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

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

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

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

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

How to access package from another package?

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

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

1) Using packagename.*

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

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
Output:Hello
```

2) Using package.name.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
Output:Hello
```

3) Using fully qualified name

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

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

Example of package by import fully qualified name

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```


Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created to **categorize the package further**.

Let's take an example, Sun Microsystems has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

package com.javatpoint.core;

class Simple{

public static void main(String args[]){

System.out.println("Hello subpackage");

}

}

To Compile: javac -d . Simple.java

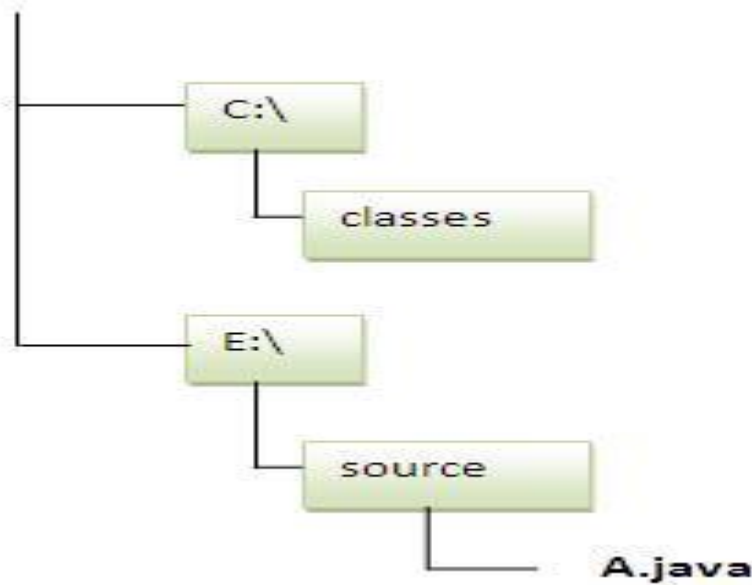
To Run: java com.javatpoint.core.Simple

Output:Hellosubpackage

3.2) CREATING USER DEFINED PACKAGE

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

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



//save as Simple.java

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

To Compile:

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

To Run:

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

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

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
 - By setting the classpath in the command prompt
 - By -classpath switch
- Permanent
 - By setting the classpath in the environment variables

- By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compile Time Error

```
class A{}  
class B{}  
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

//save as A.java
package javatpoint;
public class A{
//save as B.java

```
package javatpoint;  
public class B{
```

Simple example of user defined package :-

Q. Write a program to create a user defined package in Java.

Answer:

A package is a mechanism to group the similar type of classes, interfaces and sub-packages and provide access control. It organizes classes into single unit.

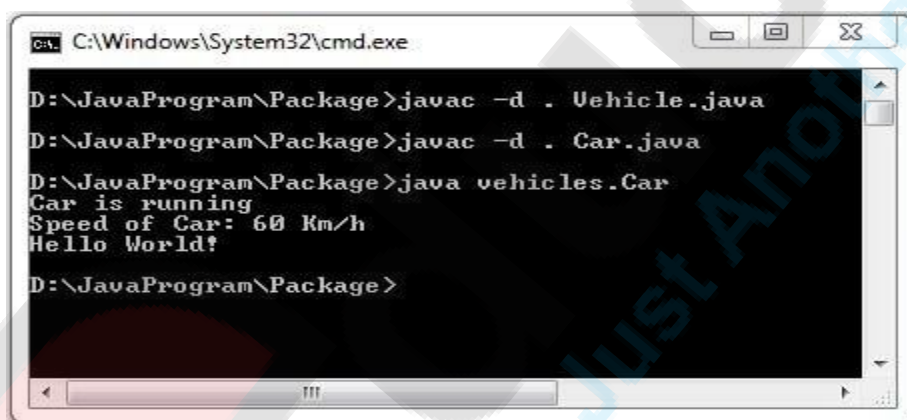
Vehicle.java

```
package vehicles;  
interface Vehicle  
{  
    public void run();  
    public void speed();  
}
```

Car.java

```
package vehicles;  
public class Car implements Vehicle  
{  
    public void run()
```

```
{
    System.out.println("Car is running.");
}
public void speed()
{
    System.out.println("Speed of Car: 50 Km/h");
}
public static void main(String args[])
{
    Car Car = new Car();
    Car.run();
    Car.speed();
    System.out.println("Hello World!");
}
}
```

Output:

```
C:\Windows\System32\cmd.exe
D:\JavaProgram\Package>javac -d . Vehicle.java
D:\JavaProgram\Package>javac -d . Car.java
D:\JavaProgram\Package>java vehicles.Car
Car is running
Speed of Car: 60 Km/h
Hello World!
D:\JavaProgram\Package>
```

3.3) ACCESS CONTROL PROTECTION IN JAVA

Packages adds another dimension to the access control. As you will see, Java provides many levels of protection to allow fine-grained control over visibility of the variables and methods within classes, subclasses, and packages.

Classes and packages are means of encapsulating and containing the name space and scope of the variables and methods.

Packages behaves as containers for classes and other subordinate packages.

Classes act as containers for data and code.

Class Members Visibility

The Java's smallest unit of abstraction is class. Because of the interplay between the classes and packages, Java addresses the following four categories of visibility for class members :

- Subclasses in same package
- Non-subclasses in same package
- Subclasses in different packages
- Classes that are neither in same package nor in subclasses

The three [access modifiers](#) are :

- public
- private
- protected

provides a variety of ways to produce many levels of access required by these categories. The upcoming table sums up the interaction

While the access control mechanism of Java may seem complicated, we can simplify it as follows.

Anything declared as **public** can be accessed from anywhere.

Anything declared as **private** can't be seen outside of its class.

Class Member Access

When a member doesn't have an explicit access specification, then it is visible to the subclasses as well as to the other classes in the same package. This is the default access. And If you want to allow an element to be seen outside your current package, but only to the classes that subclass your class directly, then declare that element protected.

	Private	Protected	Public	No Modifier
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	Yes	Yes	No
Different package non-subclass	No	No	Yes	No

This table applies only to the members of classes. A non-nested class has only two possible access levels i.e., **default** and **public**.

When a class is declared as **public**, then it is accessible by any other code. If a class has default access, then it can only be accessed by the other code within its same package. When

a class is public, it must be only the public class declared in the file that must have the same name as the class.

3.4) DEFINING INTERFACE IN JAVA

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An **interface in java** is a blueprint of a class. It has static constants and abstract methods. The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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

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

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

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

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface –

Example

Following is an example of an interface –

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
// Any number of final, static fields
// Any number of abstract method declarations\
}
```

Interfaces have the following properties –

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

Example

```
/* File name : Animal.java */
interface Animal {
public void eat();
public void travel();
}
```

3.5 IMPLEMENTING INTERFACE

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

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

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {
public void eat(){
System.out.println("Mammal eats");
}
public void travel(){
System.out.println("Mammal travels");
}
public int noOfLegs(){
return 0;
}
```

```

public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
}
}

```

This will produce the following result –

Output

Mammal eats

Mammal travels

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

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

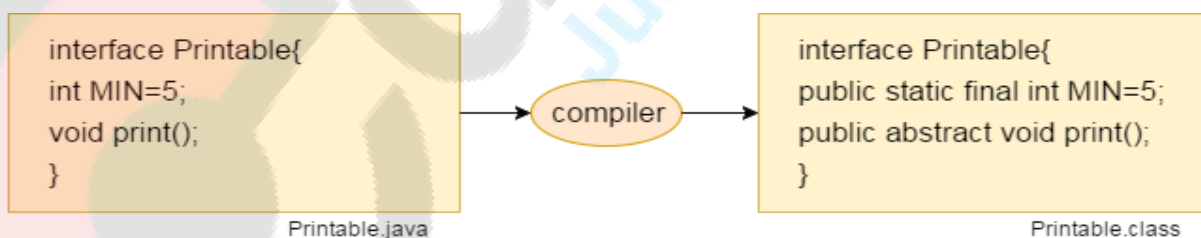
When implementing interfaces, there are several rules –

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Internal addition by compiler

The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

```

interface printable{
    void print();
}

class A6 implements printable{
    public void print(){System.out.println("Hello");}
}

```

```

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}

```

Test it Now

Output:
Hello

Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

File: TestInterface1.java

//Interface declaration: by first user

```

interface Drawable{
void draw();
}

```

//Implementation: by second user

```

class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

```

//Using interface: by third user

```

class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}

```

Test it Now

Output:
drawing circle

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```

interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
}
}

```

```
System.out.println("ROI: "+b.rateOfInterest());
}}
```

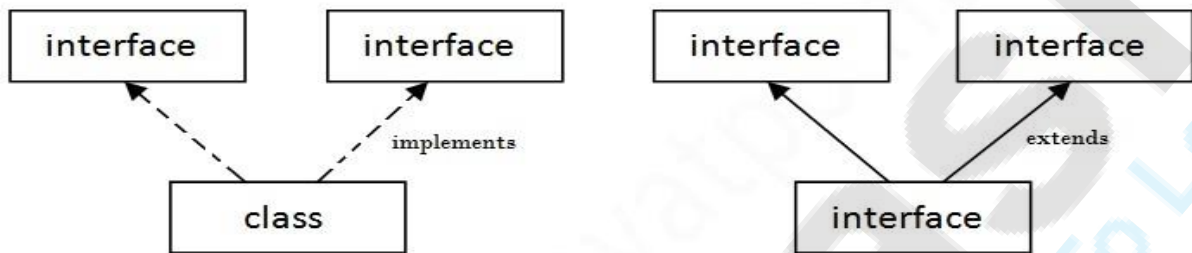
Test it Now

Output:

ROI: 9.15

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

Test it Now

Output:Hello
Welcome

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
```

```

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}

```

Test it Now

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```

interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

```

```

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
}
}

```

Test it Now

Output:

Hello

Welcome

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method.

Let's see an example:

File: *TestInterfaceDefault.java*

```

interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
}
}

```

```
d.msg();  
}}
```

Test it Now

Output:
drawing rectangle
default method

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{  
    void draw();  
    static int cube(int x){return x*x*x;}  
}  
class Rectangle implements Drawable{  
    public void draw(){System.out.println("drawing rectangle");}  
}
```

```
class TestInterfaceStatic{  
    public static void main(String args[]){  
        Drawable d=new Rectangle();  
        d.draw();  
        System.out.println(Drawable.cube(3));  
    }  
}
```

Test it Now

Output:
drawing rectangle
27

UNIT 4

GENERICS AND COLLECTION

4.1) INTRODUCTION TO GENERIC IN JAVA

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

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

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

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects

4.1.1) Generic Methods

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

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{
    public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T', 'P', 'O', 'I', 'N', 'T' };
        System.out.println( "Printing Integer Array" );
        printArray( intArray );
        System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Output:

```
Printing Integer Array
10
20
30
40
50
Printing Character Array
J
A
V
A
T
P
O
I
N
T
```

ADVANTAGE OF GENERICS IN JAVA :-

There are mainly 3 advantages of generics. They are as follows:

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

2) Type casting is not required: There is no need to typecast the object.
Before Generics, we need to type cast.

1. List list = new ArrayList();
 2. list.add("hello");
 3. String s = (String) list.get(0); //typecasting
- After Generics, we don't need to typecast the object.

1. List<String> list = new ArrayList<String>();
2. list.add("hello");

3. String s = list.get(0);

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

1. List<String> list = new ArrayList<String>();
2. list.add("hello");
3. list.add(32); //Compile Time Error

Syntax to use generic collection

1. ClassOrInterface<Type>
Example to use Generics in java
1. ArrayList<String>

Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32); //compile time error
```

```
String s=list.get(1); //type casting is not required
System.out.println("element is: "+s);
```

```
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

```
Output:element is: jai
        rahul
        jai
```

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");
```

```
//Now use Map.Entry for Set and Iterator
```

```
Set<Map.Entry<Integer,String>> set=map.entrySet();
```

```
Iterator<Map.Entry<Integer,String>> itr=set.iterator();
```

```
while(itr.hasNext()){
```

```
Map.Entry e=itr.next();//no need to typecast
```

```
System.out.println(e.getKey()+" "+e.getValue());
```

```
}
```

```
}}
```

```
Output:1 vijay
```

```
2 ankit
```

```
4 umesh
```

4.1.2) GENERIC CLASS

- A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.
- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

4.1.3) CREATING THE GENERIC CLASS

```
class MyGen<T>{
```

```
T obj;
```

```
void add(T obj){this.obj=obj;}
```

```
T get(){return obj;}
```

```
}
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{
```

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

```
MyGen<Integer> m=new MyGen<Integer>();
```

```
m.add(2);
```

```
//m.add("vivek");//Compile time error
```

```
System.out.println(m.get());
```

```
}}
```

```
Output:2
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

```
import java.util.*;
abstract class Shape{
    abstract void draw();
}
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
    void draw(){System.out.println("drawing circle");}
}
class GenericTest{
    //creating a method that accepts only child class of Shape
    public static void drawShapes(List<? extends Shape> lists){
        for(Shape s:lists){
            s.draw();//calling method of Shape class by child class instance
        }
    }
    public static void main(String args[]){
        List<Rectangle> list1=new ArrayList<Rectangle>();
        list1.add(new Rectangle());
        List<Circle> list2=new ArrayList<Circle>();
        list2.add(new Circle());
        list2.add(new Circle());
        drawShapes(list1);
        drawShapes(list2);
    }
}
OUTPUT:-
drawing rectangle
drawing circle
drawing circle
```

4.1.4) BOUNDED TYPE PARAMETERS

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter.
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects –

```
public class MaximumTest {
    // determines the largest of three Comparable objects

    public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
        T max = x; // assume x is initially the largest

        if (y.compareTo(max) > 0) {
            max = y; // y is the largest so far
        }
        if (z.compareTo(max) > 0) {
            max = z; // z is the largest now
        }
        return max; // returns the largest object
    }

    public static void main(String args[]) {
        System.out.printf("Max of %d, %d and %d is %d\n",
            3, 4, 5, maximum(3, 4, 5));
        System.out.printf("Max of %.1f, %.1f and %.1f is %.1f\n",
            6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
        System.out.printf("Max of %s, %s and %s is %s\n", "pear",
            "apple", "orange", maximum("pear", "apple", "orange"));
    }
}
```

This will produce the following result –

Output

Max of 3, 4 and 5 is 5

Max of 6.6, 8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

4.2) COLLECTIONS IN JAVA :-

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

The collections framework was designed to meet several goals, such as –

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList**, **HashSet**, and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following –

- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

What is Collection in java

Collection represents a single unit of objects i.e. a group.

What is framework in java

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

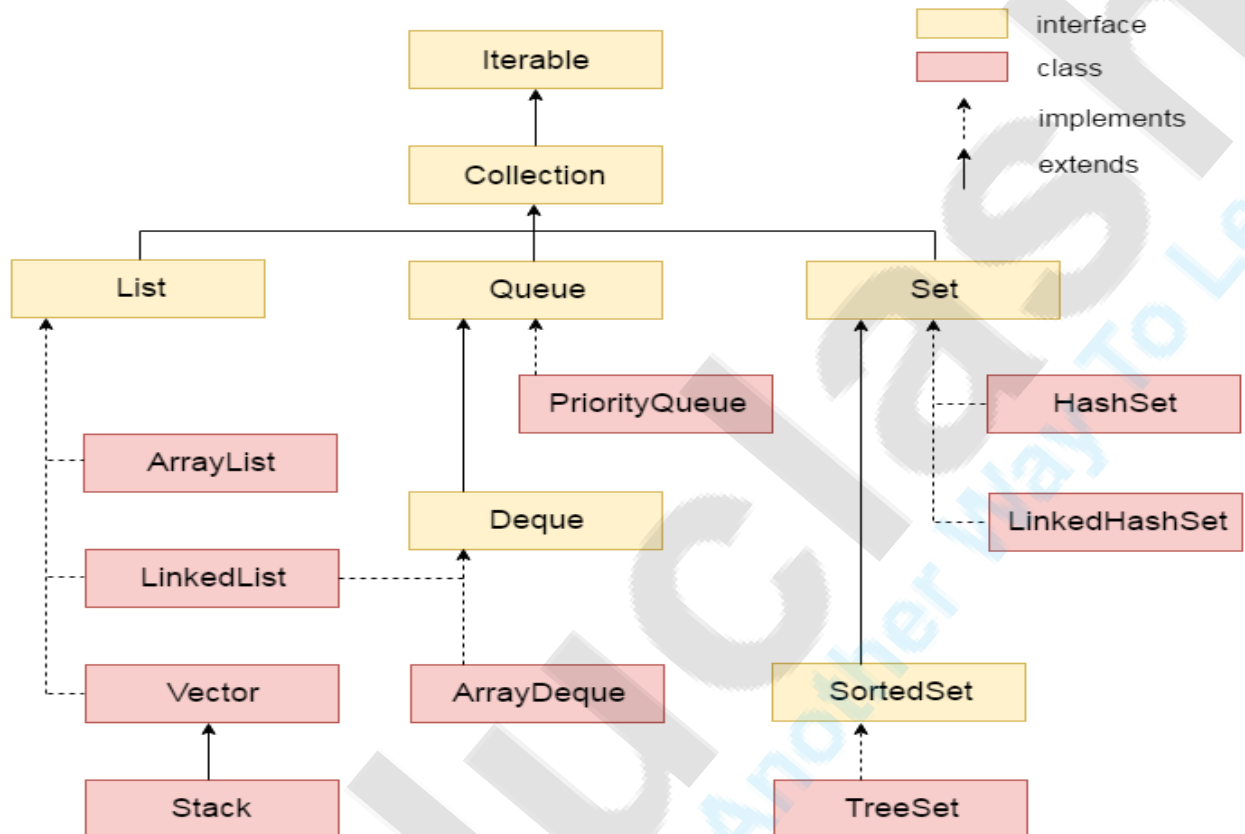
What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces
2. Implementations i.e. classes
3. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



4.2.1 COLLECTION INTERFACES

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following table.

Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**.

The collections framework defines several interfaces. This section provides an overview of each interface –

Sr.No.	Interface & Description
1	The Collection Interface This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	The List Interface

	This extends Collection and an instance of List stores an ordered collection of elements.
3	The Set This extends Collection to handle sets, which must contain unique elements.
4	The SortedSet This extends Set to handle sorted sets.
5	The Map This maps unique keys to values.
6	The Map.Entry This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	The SortedMap This extends Map so that the keys are maintained in an ascending order.
8	The Enumeration This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean	is used to search the specified collection in this

	<code>containsAll(Collection c)</code>	collection.
10	<code>public Iterator iterator()</code>	returns an iterator.
11	<code>public Object[] toArray()</code>	converts collection into array.
12	<code>public boolean isEmpty()</code>	checks if collection is empty.
13	<code>public boolean equals(Object element)</code>	matches two collection.
14	<code>public int hashCode()</code>	returns the hashcode number for collection.

4.3) COLLECTION CLASSES:-

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table –

Sr.No.	Class & Description
1	<code>AbstractCollection</code> Implements most of the Collection interface.
2	<code>AbstractList</code> Extends <code>AbstractCollection</code> and implements most of the List interface.
3	<code>AbstractSequentialList</code> Extends <code>AbstractList</code> for use by a collection that uses sequential rather than random access of its elements.
4	<code>LinkedList</code> Implements a linked list by extending <code>AbstractSequentialList</code> .
5	<code>ArrayList</code> Implements a dynamic array by extending <code>AbstractList</code> .
6	<code>AbstractSet</code> Extends <code>AbstractCollection</code> and implements most of the Set interface.
7	<code>HashSet</code> Extends <code>AbstractSet</code> for use with a hash table.
8	<code>LinkedHashSet</code> Extends <code>HashSet</code> to allow insertion-order iterations.
9	<code>TreeSet</code> Implements a set stored in a tree. Extends <code>AbstractSet</code> .

10	AbstractMap Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents.

The *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

4.3.1) LINKS

A LINK or symbolic link contains a reference to another file or directory.

The file referenced by a symbolic link is known as the target file for the symbolic link.

Operations on a symbolic link are transparent to the application. We can work with symbolic links using the `java.nio.file.Files` class.

`isSymbolicLink(Path p)` method checks if the file specified by the specified path is a symbolic link.

`createSymbolicLink()` method of the `Files`, which may not be supported on all platforms, creates a symbolic link.

```
import java.nio.file.Files;
```

```
import java.nio.file.Path;
```

```
import java.nio.file.Paths;
```

```
/*www.jav a2s.com*/
```

```
publicclass Main {
```

```
    publicstaticvoid main(String[] args) throws Exception {
```

```
        Path existingFilePath = Paths.get("C:\\Java_Dev\\test1.txt");
```

```
        Path symLinkPath = Paths.get("C:\\test1_link.txt");
```

```
        Files.createSymbolicLink(symLinkPath, existingFilePath);
```

```
    }
```

```
}
```

The Java NIO API follows the symbolic link by default. We can specify whether we want to follow a symbolic link or not. The option not to follow a symbolic link is indicated by using the enum constant `LinkOption.NOFOLLOW_LINKS`.

The `LinkOption` enum is declared in the `java.nio.file` package. Methods supporting this option let we pass an argument of the `LinkOption` type.

We can use the `createLink(Path newLink, Path existingPath)` method of the `Files` class to create a hard link.

4.3.2 VECTOR

Vector implements a dynamic array. It is similar to ArrayList, but with two differences –

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	Vector() This constructor creates a default vector, which has an initial size of 10.
2	Vector(int size) This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	Vector(int size, int incr) This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	Vector(Collection c) This constructor creates a vector that contains the elements of collection c.

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

Sr.No.	Method & Description
1	void add(int index, Object element) Inserts the specified element at the specified position in this Vector.
2	boolean add(Object o) Appends the specified element to the end of this Vector.
3	boolean addAll(Collection c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	boolean addAll(int index, Collection c) Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	void addElement(Object obj) Adds the specified component to the end of this vector, increasing its size by one.
6	int capacity() Returns the current capacity of this vector.

7	<code>void clear()</code> Removes all of the elements from this vector.
8	<code>Object clone()</code> Returns a clone of this vector.
9	<code>boolean contains(Object elem)</code> Tests if the specified object is a component in this vector.
10	<code>boolean containsAll(Collection c)</code> Returns true if this vector contains all of the elements in the specified Collection.
11	<code>void copyInto(Object[] anArray)</code> Copies the components of this vector into the specified array.
12	<code>Object elementAt(int index)</code> Returns the component at the specified index.
13	<code>Enumeration elements()</code> Returns an enumeration of the components of this vector.
14	<code>void ensureCapacity(int minCapacity)</code> Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	<code>boolean equals(Object o)</code> Compares the specified Object with this vector for equality.
16	<code>Object firstElement()</code> Returns the first component (the item at index 0) of this vector.
17	<code>Object get(int index)</code> Returns the element at the specified position in this vector.
18	<code>int hashCode()</code> Returns the hash code value for this vector.
19	<code>int indexOf(Object elem)</code> Searches for the first occurrence of the given argument, testing for equality using the equals method.
20	<code>int indexOf(Object elem, int index)</code> Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	<code>void insertElementAt(Object obj, int index)</code> Inserts the specified object as a component in this vector at the specified index.
22	<code>boolean isEmpty()</code> Tests if this vector has no components.

23	<code>Object lastElement()</code> Returns the last component of the vector.
24	<code>int lastIndexOf(Object elem)</code> Returns the index of the last occurrence of the specified object in this vector.
25	<code>int lastIndexOf(Object elem, int index)</code> Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	<code>Object remove(int index)</code> Removes the element at the specified position in this vector.
27	<code>boolean remove(Object o)</code> Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.
28	<code>boolean removeAll(Collection c)</code> Removes from this vector all of its elements that are contained in the specified Collection.
29	<code>void removeAllElements()</code> Removes all components from this vector and sets its size to zero.
30	<code>boolean removeElement(Object obj)</code> Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	<code>void removeElementAt(int index)</code> <code>removeElementAt(int index).</code>
32	<code>protected void removeRange(int fromIndex, int toIndex)</code> Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	<code>boolean retainAll(Collection c)</code> Retains only the elements in this vector that are contained in the specified Collection.
34	<code>Object set(int index, Object element)</code> Replaces the element at the specified position in this vector with the specified element.
35	<code>void setElementAt(Object obj, int index)</code> Sets the component at the specified index of this vector to be the specified object.
36	<code>void setSize(int newSize)</code> Sets the size of this vector.
37	<code>int size()</code> Returns the number of components in this vector.
38	<code>List subList(int fromIndex, int toIndex)</code> Returns a view of the portion of this List between fromIndex, inclusive, and toIndex,

	exclusive.
39	Object[] toArray() Returns an array containing all of the elements in this vector in the correct order.
40	Object[] toArray(Object[] a) Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
41	String toString() Returns a string representation of this vector, containing the String representation of each element.
42	void trimToSize() Trims the capacity of this vector to be the vector's current size.

Example

The following program illustrates several of the methods supported by this collection –

```
import java.util.*;
public class VectorDemo {
    public static void main(String args[]) {
        // initial size is 3, increment is 2
        Vector v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: " + v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
        System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Float(9.4));
        v.addElement(new Integer(10));
        System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Integer(11));
        v.addElement(new Integer(12));
        System.out.println("First element: " + (Integer)v.firstElement());
        System.out.println("Last element: " + (Integer)v.lastElement());
        if (v.contains(new Integer(3)))
            System.out.println("Vector contains 3.");
        // enumerate the elements in the vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while (vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();
    }
}
```

```
}
}
```

This will produce the following result –

Output

Initial size: 0

Initial capacity: 3

Capacity after four additions: 5

Current capacity: 5

Current capacity: 7

Current capacity: 9

First element: 1

Last element: 12

Vector contains 3.

Elements in vector:

1 2 3 4 5.45 6.08 7 9.4 10 11 12

4.3.3) LINKED LIST CLASS:-

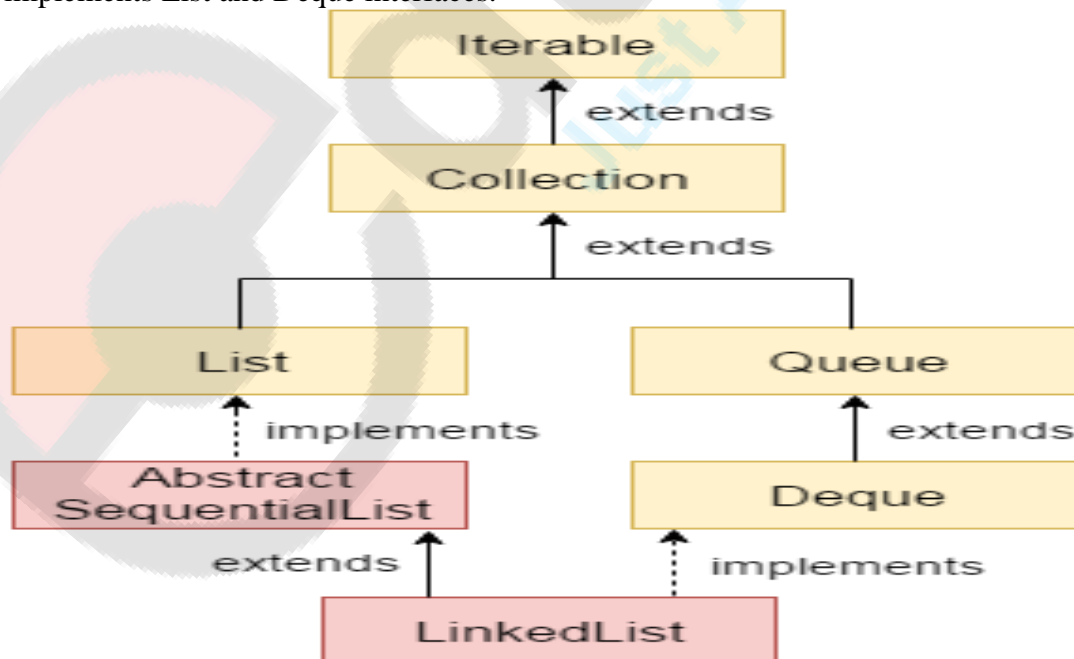
Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

Hierarchy of LinkedList class

As shown in below diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.



Hierarchy of LinkedList

Doubly Linked List

In case of doubly linked list, we can add or remove elements from both side.



fig- doubly linked list

LinkedList class declaration

Let's see the declaration for java.util.LinkedList class.

Ex:-

public class LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection c)	It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods of Java LinkedList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurrence of the specified

	element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Java LinkedList Example

```
import java.util.*;
public class TestCollection7{
    public static void main(String args[]){
        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

Output:Ravi
Vijay
Ravi
Ajay

Java LinkedList Example: Book

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
    public Book(int id, String name, String author, String publisher, int quantity) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.publisher = publisher;
        this.quantity = quantity;
    }
}
public class LinkedListExample {
```



```
public static void main(String[] args) {  
    //Creating list of Books  
    List<Book> list=new LinkedList<Book>();  
    //Creating Books  
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);  
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);  
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);  
    //Adding Books to list  
    list.add(b1);  
    list.add(b2);  
    list.add(b3);  
    //Traversing list  
    for(Book b:list){  
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);  
    }  
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8  
102 Data Communications & Networking Forouzan Mc Graw Hill 4  
103 Operating System Galvin Wiley 6
```

4.3.4) JAVA MAP INTERFACE :-

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.
- A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.
- Map is useful if you have to search, update or delete elements on the basis of key.

Useful methods of Map interface

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.
void putAll(Map map)	It is used to insert the specified map in this map.

Object remove(Object key)	It is used to delete an entry for the specified key.
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.

Map.Entry Interface

Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

Methods of Map.Entry interface

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.

Java Map Example: Generic (New Style)

```
import java.util.*;
class MapInterfaceExample{
    public static void main(String args[]){
        Map<Integer,String> map=new HashMap<Integer,String>();
        map.put(100,"Amit");
        map.put(101,"Vijay");
        map.put(102,"Rahul");
        for(Map.Entry m:map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output:

```
102 Rahul
100 Amit
101 Vijay
```

Java Map Example: Non-Generic (Old Style)

```
//Non-generic
import java.util.*;
public class MapExample1 {
    public static void main(String[] args) {
        Map map=new HashMap();
        //Adding elements to map
        map.put(1,"Amit");
        map.put(5,"Rahul");
        map.put(2,"Jai");
        map.put(6,"Amit");
    }
}
```

```
//Traversing Map
Set set=map.entrySet();//Converting to Set so that we can traverse
Iterator itr=set.iterator();
while(itr.hasNext()){
    //Converting to Map.Entry so that we can get key and value separately
    Map.Entry entry=(Map.Entry)itr.next();
    System.out.println(entry.getKey()+" "+entry.getValue());
}
}
```

Output:

```
1 Amit
2 Jai
5 Rahul
6 Amit
```

4.3.5) JAVA HASHMAP :-

The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get() and put(), to remain constant even for large sets.

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

The important points about Java HashMap class are:

- A HashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.

Hierarchy of HashMap class

As shown in the figure below, HashMap class extends AbstractMap class and implements Map interface.

HashMap class declaration

Let's see the declaration for java.util.HashMap class.

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable
```

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K:** It is the type of keys maintained by this map.
- **V:** It is the type of mapped values.

Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initialize the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float fillRatio)	It is used to initialize both the capacity and fill ratio of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
Object put(Object key, Object value)	It is used to associate the specified value with the specified key in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

Java HashMap Example

```
import java.util.*;
class TestCollection13{
    public static void main(String args[]){
        HashMap<Integer,String> hm=new HashMap<Integer,String>();
        hm.put(100,"Amit");
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");
        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output:102 Rahul

100 Amit

101 Vijay

Java HashMap Example: remove()

```
import java.util.*;
public class HashMapExample {
    public static void main(String args[]) {
        // create and populate hash map
        HashMap<Integer, String> map = new HashMap<Integer, String>();
        map.put(101,"Let us C");
        map.put(102, "Operating System");
        map.put(103, "Data Communication and Networking");
        System.out.println("Values before remove: "+ map);
        // Remove value for key 102
        map.remove(102);
        System.out.println("Values after remove: "+ map);
    }
}
```

Output:

Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

Values after remove: {103=Data Communication and Networking, 101=Let us C}

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains entry(key and value).

Java HashMap Example: Book

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
    public Book(int id, String name, String author, String publisher, int quantity) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.publisher = publisher;
    }
}
```

```

    this.quantity = quantity;
}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new HashMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill"
,4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(1,b1);
    map.put(2,b2);
    map.put(3,b3);

    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}

```

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

4.3.6 WILDCARDS IN JAVA

Bounded and unbounded wildcards in Generics are two types of wildcard available on Java.

Any Type can be bounded either upper or lower of the class hierarchy in Generics by using bounded wildcards. In short <? extends T> and <? super T> represent bounded wildcards while <?> represent an unbounded wildcard in generics .

What are bounded and unbounded wildcards in Generics

- bounded and unbounded wildcards in generics are used to bound any Type. Type can be upper bounded by using <? extends T> where all Types must be sub-class of T or lower bounded using <? super T> where all Types required to be the super class of T, here T represent the lower bound. Single <?> is called an unbounded wildcard in generic and it can represent any type, similar to Object in Java.

- For example `List<?>` can represent any `List` e.g. `List<String>` or `List<Integer>` its provides highest level of flexibility on passing method argument.
- On the other hand, bounded wildcards provide limited flexibility within bound. Any Type with bounded wildcards can only be instantiated within bound and any instantiation outside bound will result in compiler error.
- One of the important benefits of using bounded wildcard is that it not only restrict the number of Types can be passed to any method as an argument it also provides access to methods declared by bound.

4.4) LAMBDA EXPRESSIONS IN JAVA

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection. Before lambda expression, anonymous inner class was the only option to implement the method.

In other words, we can say it is a replacement of java inner anonymous class. Java lambda expression is treated as a function, so compiler does not create .class file.

Syntax

A lambda expression is characterized by the following syntax –

```
parameter -> expression body
```

Following are the important characteristics of a lambda expression –

- **Optional type declaration** – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Scope

Using lambda expression, you can refer to final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

Scope Example

Create the following Java program using editor and save in some folder like C:\>JAVA.

Java8Tester.java

```

public class Java8Tester {
    final static String salutation = "Hello! ";
    public static void main(String args[]) {
        GreetingService greetService1 = message ->
        System.out.println(salutation + message);
        greetService1.sayMessage("Mahesh");
    }
    interface GreetingService {
        void sayMessage(String message);
    }
}

```

Verify the Result

Compile the class using **javac** compiler as follows –

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
Hello! Mahesh
```

Functional Interface

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an annotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java lambda expression is consisted of three components.

- 1) **Argument-list:** It can be empty or non-empty as well.
- 2) **Arrow-token:** It is used to link arguments-list and body of expression.
- 3) **Body:** It contains expressions and statements for lambda expression.

Let's see a scenario. If we don't implement Java lambda expression. Here, we are implementing an interface method without using lambda expression.

Java Example without Lambda Expression

```

interface Drawable {
    public void draw();
}

public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;
        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){

```

```

        public void draw(){System.out.println("Drawing "+width);}
    };
    d.draw();
}
}

```

Output:

Drawing 10

Java Example with Lambda Expression

Now, we are implementing the above example with the help of lambda expression.

@FunctionalInterface //It is optional

```

interface Drawable{
    public void draw();
}

public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}

```

Output:

Drawing 10

4.4.1) LAMDA TYPE INFERENCE

Type Inference means that the data type of any expression (e.g. method return type or parameter type) can be deduced automatically by the compiler.

Similarly, Java 8 Lambda expressions also support Type inference. Let's understand how it works with a few examples.

Suppose we have to calculate the sum of two numbers. As you know, we need an interface method that consumes two parameters of say Number type and returns the output of the same type. Let's create an interface and provide its inline implementation using Lambda expressions.

```

public class LambdaTypeInferenceExample {
    public static void main(String[] args) {
        BiFunction<Integer,Integer,Integer> summer = (Integer number1,Integer
        number2) -> { return number1 + number2;};

        Integer number1 = 10;
        Integer number2 = 20;

        System.out.println(number1+" + "+number2+" =

```

```

        "+summer.apply(number1,number2));
    }
}
interface BiFunction<K,V,R>{
    R apply(K k ,V v);
}

```

Observe that we have declared an interface called BiFunction which has a method, 'apply' accepting two parameters of any type K & V respectively and returns another type of value i.e. R. These types could be same or different as you provide their types during declaration. Here, we are summing up two integers and returning the output as an integer. For now, don't worry about any edge cases like sum of the maximum of two integers can't be equal to the integer as it exceeds the maximum limit. By using type inference feature, you could omit, the type of number1 and number2. The compiler would now automatically deduce the type, on the basis of input values as:

Here, we are summing up two integers and returning the output as an integer. For now, don't worry about any edge cases like sum of the maximum of two integers can't be equal to the integer as it exceeds the maximum limit. By using type inference feature, you could omit, the type of number1 and number2. The compiler would now automatically deduce the type, on the basis of input values as:

1. If you haven't specified generic types during reference variable declaration, type of number1 and number2 would be deduced as Object class instances.
2. If specified the generic types, it will automatically deduce the type and operation available for that type could be used.

The above code thus looks crisper after removing types from parameters:

```

public class LambdaTypeInferenceExample {
    public static void main(String[] args) {
        BiFunction<Integer,Integer,Integer> summer = (number1,number2) -> {
            return number1 + number2;};
        Integer number1 = 10;
        Integer number2 = 20;
        System.out.println(number1+" "+number2+" =
        "+summer.apply(number1,number2));
    }
}

```

```
interface BiFunction<K,V,R>{
    R apply(K k , V v);
}
```

This is an example of type inference here. Let's go one more step ahead. We are saying return the following:

```
1    number1 + number 2
```

Here, we could omit 'return from statement' as there is only one statement and hence could easily be inferred by the compiler. As you know, if there is only one statement, we don't need parenthesis [around lambda expression](#) statements.

Hence, the above code will finally look like:

```
public class LambdaTypeInferenceExample {
    public static void main(String[] args) {
        BiFunction<Integer,Integer,Integer> summer = (number1,number2) -
        > number1 + number2;
        Integer number1 = 10;
        Integer number2 = 20;
        System.out.println(number1+" "+number2+" =
        "+summer.apply(number1,number2));
    }
}

interface BiFunction<K,V,R>{
    R apply(K k , V v);
}
```

Hope now we have an understanding of how Type Inference works in lambda expressions.

Though Java supports this, it is still a strict type-checking language.

4.4.2) LAMBDA PARAMETERS

A lambda expression can have zero or any number of arguments. Let's see the examples:

Java Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}

public class LambdaExpressionExample{
    public static void main(String[] args) {
        Sayable s=()->{
            return "I have nothing to say.";
        }
    }
}
```

```
};  
System.out.println(s.say());  
}  
}
```

Output:

I have nothing to say.

Java Lambda Expression Example: Single Parameter

```
interface Sayable{  
    public String say(String name);  
}
```

```
public class LambdaExpressionExample{  
    public static void main(String[] args) {  
        // Lambda expression with single parameter.  
        Sayable s1=(name)->{  
            return "Hello, "+name;  
        };  
        System.out.println(s1.say("Sonoo"));  
  
        // You can omit function parentheses  
        Sayable s2= name ->{  
            return "Hello, "+name;  
        };  
        System.out.println(s2.say("Sonoo"));  
    }  
}
```

Output:

Hello, Sonoo

Hello, Sonoo

Java Lambda Expression Example: Multiple Parameters

```
interface Addable{  
    int add(int a,int b);  
}
```

```
public class LambdaExpressionExample{  
    public static void main(String[] args) {  
  
        // Multiple parameters in lambda expression  
        Addable ad1=(a,b)->(a+b);  
        System.out.println(ad1.add(10,20));  
  
        // Multiple parameters with data type in lambda expression  
        Addable ad2=(int a,int b)->(a+b);  
        System.out.println(ad2.add(100,200));  
    }  
}
```

Output:

30

300

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

```
package lambdaExample;
interface Addable{
    int add(int a,int b);
}
public class lambdaExpression {
    public static void main(String[] args) {
        // Lambda expression without return keyword.
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));
        // Lambda expression with return keyword.
        Addable ad2=(int a,int b)->{
            return (a+b);
        };
        System.out.println(ad2.add(100,200));
    }
}
```

Output:

30
300

Java Lambda Expression Example: Foreach Loop

```
import java.util.*;
public class LambdaExpressionExample{
    public static void main(String[] args) {
        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");

        list.forEach(
            (n)->System.out.println(n)
        );
    }
}
```

Output:

ankit
mayank
irfan
jai

Java Lambda Expression Example: Multiple Statements

```
@FunctionalInterface
interface Sayable{
    String say(String message);
}
public class LambdaExpressionExample{
    public static void main(String[] args) {
```

```
// You can pass multiple statements in lambda expression
Sayable person = (message)-> {
    String str1 = "I would like to say, ";
    String str2 = str1 + message;
    return str2;
};
System.out.println(person.say("time is precious."));
}
```

Output:

I would like to say, time is precious.

Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

```
public class LambdaExpressionExample{
    public static void main(String[] args) {
        //Thread Example without lambda
        Runnable r1=new Runnable(){
            public void run(){
                System.out.println("Thread1 is running...");
            }
        };
        Thread t1=new Thread(r1);
        t1.start();
        //Thread Example with lambda
        Runnable r2=()->{
            System.out.println("Thread2 is running...");
        };
        Thread t2=new Thread(r2);
        t2.start();
    }
}
```

Output:

Thread1 is running...

Thread2 is running...

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

Java Lambda Expression Example: Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
```

```

    super();
    this.id = id;
    this.name = name;
    this.price = price;
}
}
public class LambdaExpressionExample{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();
        //Adding Products
        list.add(new Product(1,"HP Laptop",25000f));
        list.add(new Product(3,"Keyboard",300f));
        list.add(new Product(2,"Dell Mouse",150f));
        System.out.println("Sorting on the basis of name...");
        // implementing lambda expression
        Collections.sort(list,(p1,p2)->{
            return p1.name.compareTo(p2.name);
        });
        for(Product p:list){
            System.out.println(p.id+" "+p.name+" "+p.price);
        }
    }
}

```

Output:

Sorting on the basis of name...

2 Dell Mouse 150.0

1 HP Laptop 25000.0

3 Keyboard 300.0

Java Lambda Expression Example: Filter Collection Data

```

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class LambdaExpressionExample{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();
        list.add(new Product(1,"Samsung A5",17000f));
        list.add(new Product(3,"Iphone 6S",65000f));
        list.add(new Product(2,"Sony Xperia",25000f));
    }
}

```

```
list.add(new Product(4,"Nokia Lumia",15000f));
list.add(new Product(5,"Redmi4 ",26000f));
list.add(new Product(6,"Lenevo Vibe",19000f));
```

// using lambda to filter data

```
Stream<Product> filtered_data = list.stream().filter(p -> p.price > 20000);
```

// using lambda to iterate through collection

```
filtered_data.forEach(
    product -> System.out.println(product.name+": "+product.price)
);
}
```

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

Java Lambda Expression Example: Event Listener

```
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
```

```
import javax.swing.JTextField;
```

```
public class LambdaEventListenerExample {
```

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

```
        JTextField tf=new JTextField();
```

```
        tf.setBounds(50, 50,150,20);
```

```
        JButton b=new JButton("click");
```

```
        b.setBounds(80,100,70,30);
```

// lambda expression implementing here.

```
b.addActionListener(e-> {tf.setText("hello swing");});
```

```
JFrame f=new JFrame();
```

```
f.add(tf);f.add(b);
```

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
f.setLayout(null);
```

```
f.setSize(300, 200);
```

```
f.setVisible(true);
```

```
}
```

```
}
```

UNIT 5

EXCEPTION HANDLING

5.1) EXCEPTION HANDLING FUNDAMENTALS

The **exception handling in java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

What is exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Advantage of Exception Handling

The core advantage of exception handling is to **maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; *//exception occurs*
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

5.2) EXCEPTION TYPES IN JAVA

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked exceptions –

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a `FileNotFoundException` occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FileNotFoundException_Demo {

    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FileNotFoundException_Demo.java
```

```
FileNotFoundException_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
```

```
FileReader fr = new FileReader(file);
```

```
^
```

1 error

Note – Since the methods **read()** and **close()** of `FileReader` class throws `IOException`, you can observe that the compiler notifies to handle `IOException`, along with `FileNotFoundException`.

2) Unchecked exceptions –

- An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an `ArrayIndexOutOfBoundsException` occurs.

Example

```
public class Unchecked_Demo {

    public static void main(String args[]) {
        int num[] = {1, 2, 3, 4};
        System.out.println(num[5]);
    }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

3) Errors –

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

5.3) EXCEPTION AS OBJECTS

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however -- only those objects whose classes descend from `Throwable`. `Throwable` serves as the base class for an entire family of classes, declared in `java.lang`, that your program can instantiate and throw. A small part of this family is shown in Figure 1.

As you can see in Figure 1, `Throwable` has two direct subclasses, `Exception` and `Error`. Exceptions (members of the `Exception` family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread. Errors (members of the `Error` family) are usually thrown for more serious problems, such as `OutOfMemoryError`, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

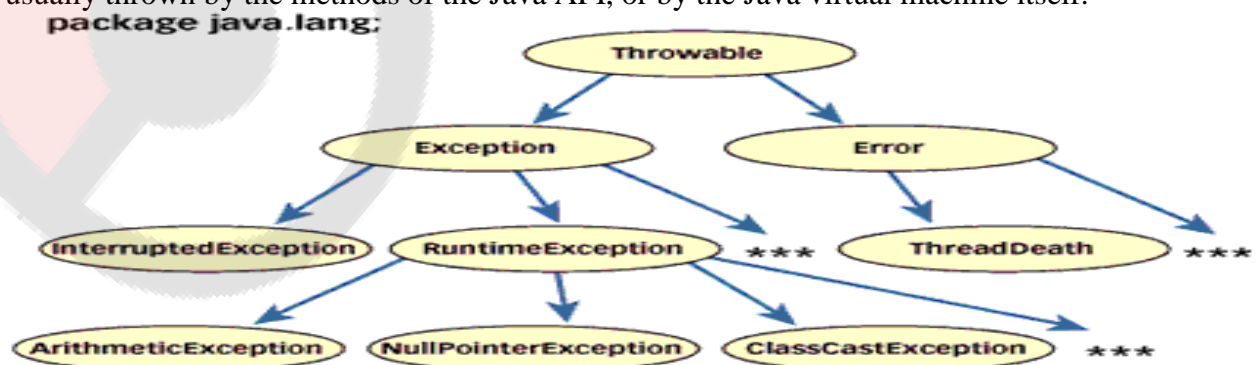


Figure 1. A partial view of the `Throwable` family

In addition to throwing objects whose classes are declared in `java.lang`, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of the **Throwable** family. In general, however, the throwable classes you define should extend class **Exception**. They should be "exceptions." The reasoning behind this rule will be explained later in this article.

Whether you use an existing exception class from `java.lang` or create one of your own depends upon the situation. In some cases, a class from `java.lang` will do just fine. For example, if one of your methods is invoked with an invalid argument, you could throw **IllegalArgumentException**, a subclass of **RuntimeException** in `java.lang`.

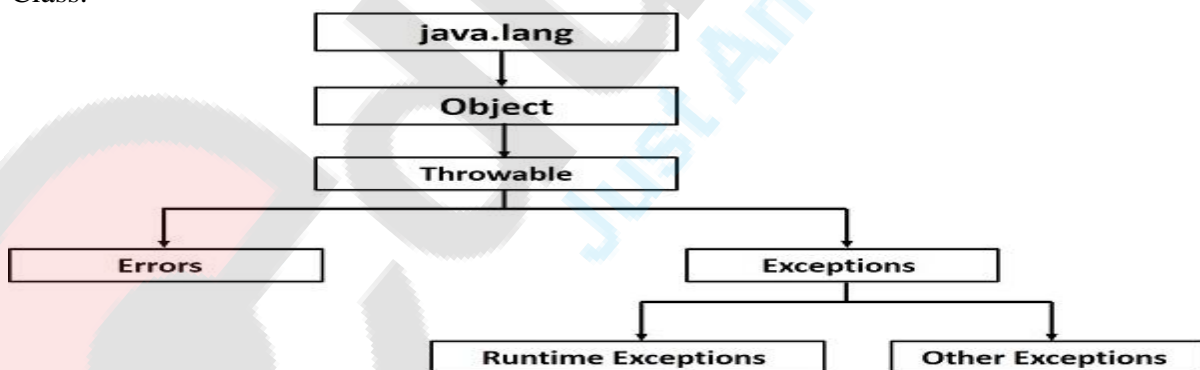
Other times, however, you will want to convey more information about the abnormal condition than a class from `java.lang` will allow. Usually, the class of the exception object itself indicates the type of abnormal condition that was encountered. For example, if a thrown exception object has class **IllegalArgumentException**, that indicates someone passed an illegal argument to a method. Sometimes you will want to indicate that a method encountered an abnormal condition that isn't represented by a class in the **Throwable** family of `java.lang`.

5.4) EXCEPTION HIERARCHY

All exception classes are subtypes of the `java.lang.Exception` class. The exception class is a subclass of the **Throwable** class. Other than the exception class there is another subclass called **Error** which is derived from the **Throwable** class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The **Exception** class has two main subclasses: **IOException** class and **RuntimeException** Class.



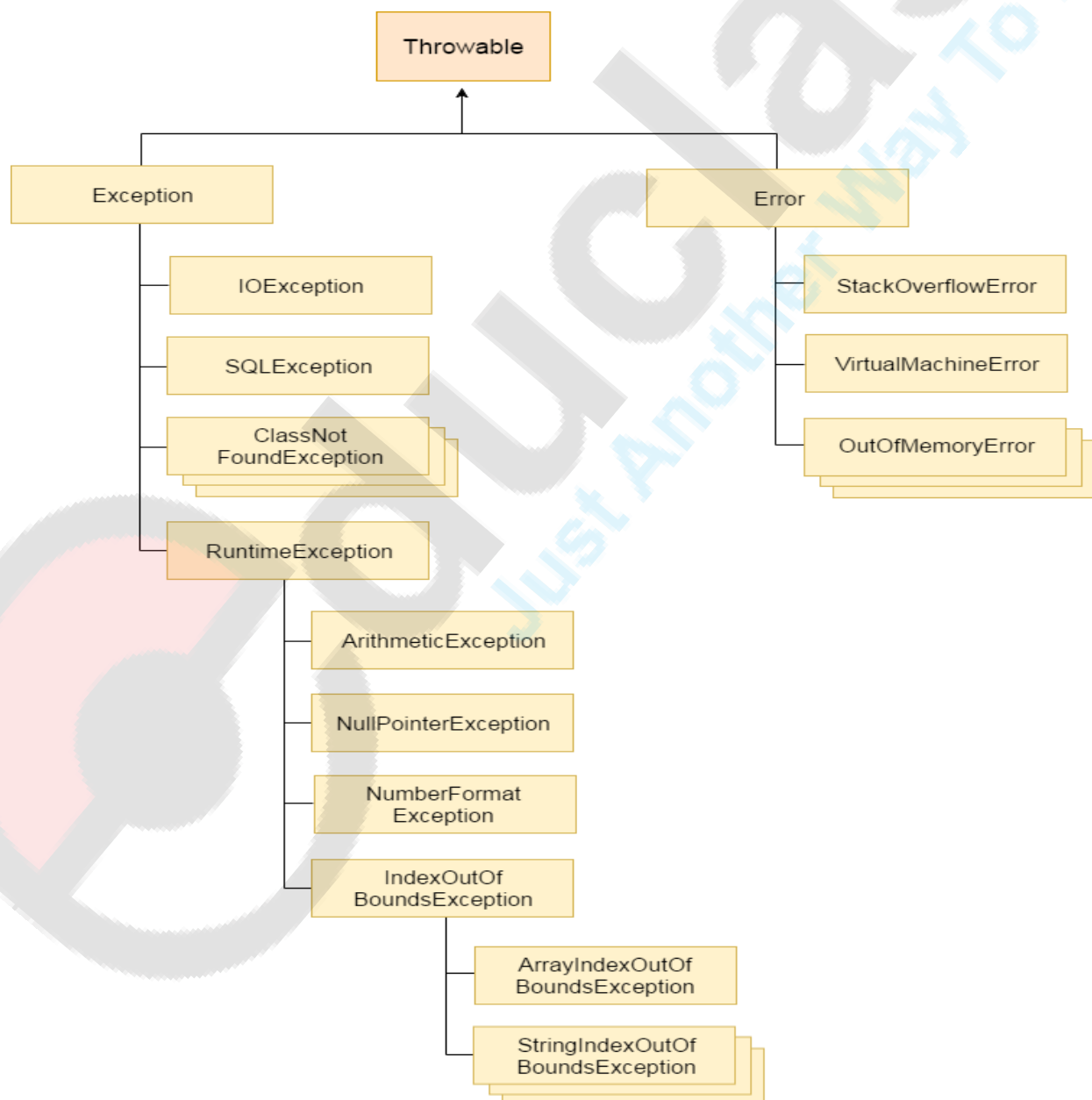
Exceptions Methods

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

Sr.No.	Method & Description
1	public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	public Throwable getCause() Returns the cause of the exception as represented by a Throwable object.

3	<code>public String toString()</code> Returns the name of the class concatenated with the result of <code>getMessage()</code> .
4	<code>public void printStackTrace()</code> Prints the result of <code>toString()</code> along with the stack trace to <code>System.err</code> , the error output stream.
5	<code>public StackTraceElement [] getStackTrace()</code> Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	<code>public Throwable fillInStackTrace()</code> Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Hierarchy of Java Exception classes



Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. `int a=50/0;//ArithmeticException`

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

1. `String s=null;`
2. `System.out.println(s.length());//NullPointerException`

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

1. `String s="abc";`
2. `int i=Integer.parseInt(s);//NumberFormatException`

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

1. `int a[]=new int[5];`
2. `a[10]=50; //ArrayIndexOutOfBoundsException`

JAVA BUILT-IN EXCEPTIONS:-

Java defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Sr.No.	Exception & Description
1	ArithmeticException Arithmetic error, such as divide-by-zero.
2	ArrayIndexOutOfBoundsException Array index is out-of-bounds.
3	ArrayStoreException Assignment to an array element of an incompatible type.
4	ClassCastException Invalid cast.

5	IllegalArgumentException Illegal argument used to invoke a method.
6	IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.
7	IllegalStateException Environment or application is in incorrect state.
8	IllegalThreadStateException Requested operation not compatible with the current thread state.
9	IndexOutOfBoundsException Some type of index is out-of-bounds.
10	NegativeArraySizeException Array created with a negative size.
11	NullPointerException Invalid use of a null reference.
12	NumberFormatException Invalid conversion of a string to a numeric format.
13	SecurityException Attempt to violate security.
14	StringIndexOutOfBoundsException Attempt to index outside the bounds of a string.
15	UnsupportedOperationException An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

Sr.No.	Exception & Description
1	ClassNotFoundException Class not found.
2	CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException Access to a class is denied.
4	InstantiationException Attempt to create an object of an abstract class or interface.
5	InterruptedException One thread has been interrupted by another thread.

6	NoSuchFieldException A requested field does not exist.
7	NoSuchMethodException A requested method does not exist.

5.5) EXCEPTION KEYWORDS IN JAVA

There are 5 keywords used in java exception handling.

- | | | |
|----------|------------|----------|
| 1. try | 3. finally | 5. throw |
| 2. catch | 4. throw | |

1)Java try block

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.

2)Java catch block

- Java catch block is used to handle the Exception. It must be used after the try block only.
- You can use multiple catch block with a single try.

JAVA TRY-CATCH BLOCK

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try {
    // Protected code
} catch (ExceptionName e1) {
    // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

EXAMPLE:-

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
    public static void main(String args[]){
        int data=50/0;//may throw exception
        System.out.println("rest of the code...");
    }
}
```



```

}
}

```

Output:

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

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

```

public class Testtrycatch2{
    public static void main(String args[]){
        try{
            int data=50/0;
        }catch(ArithmeticException e){System.out.println(e);}
        System.out.println("rest of the code...");
    }
}

```

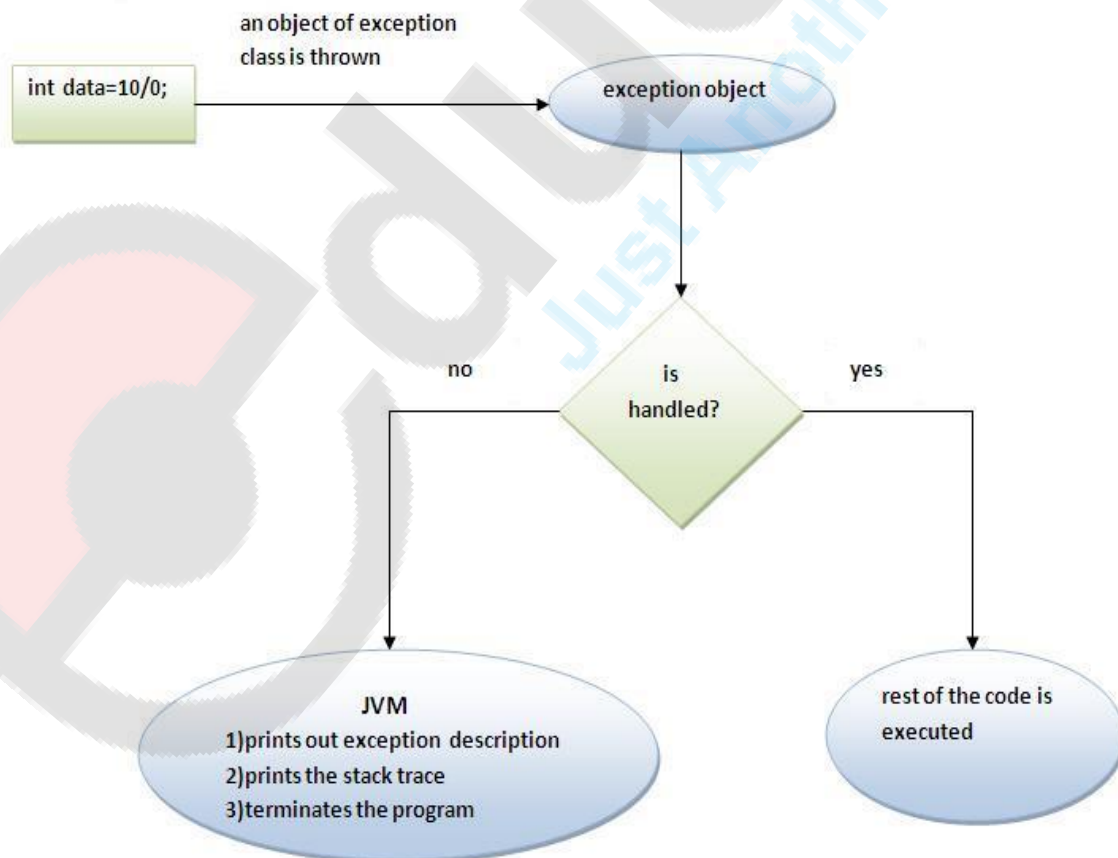
Output:

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

rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

JAVA MULTIPLE CATCH BLOCK

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following –

Syntax

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

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Let's see a simple example of java multi-catch block.

```
public class TestMultipleCatchBlock{  
    public static void main(String args[]){  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e){System.out.println("task1 is completed");}  
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}  
        catch(Exception e){System.out.println("common task completed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Test it Now

Output:task1 completed
rest of the code...

Rule: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
class TestMultipleCatchBlock1{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(Exception e){System.out.println("common task completed");}
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
        System.out.println("rest of the code...");
    }
}
```

Test it Now

Output:

Compile-time error

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it –

```
catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
}
```

JAVA NESTED TRY BLOCK

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b = 39/0;
            }catch(ArithmeticException e){System.out.println(e);}

            try{
                int a[] = new int[5];
                a[5] = 4;
            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

            System.out.println("other statement");
        }catch(Exception e){System.out.println("handed");}

        System.out.println("normal flow..");
    }
}
```

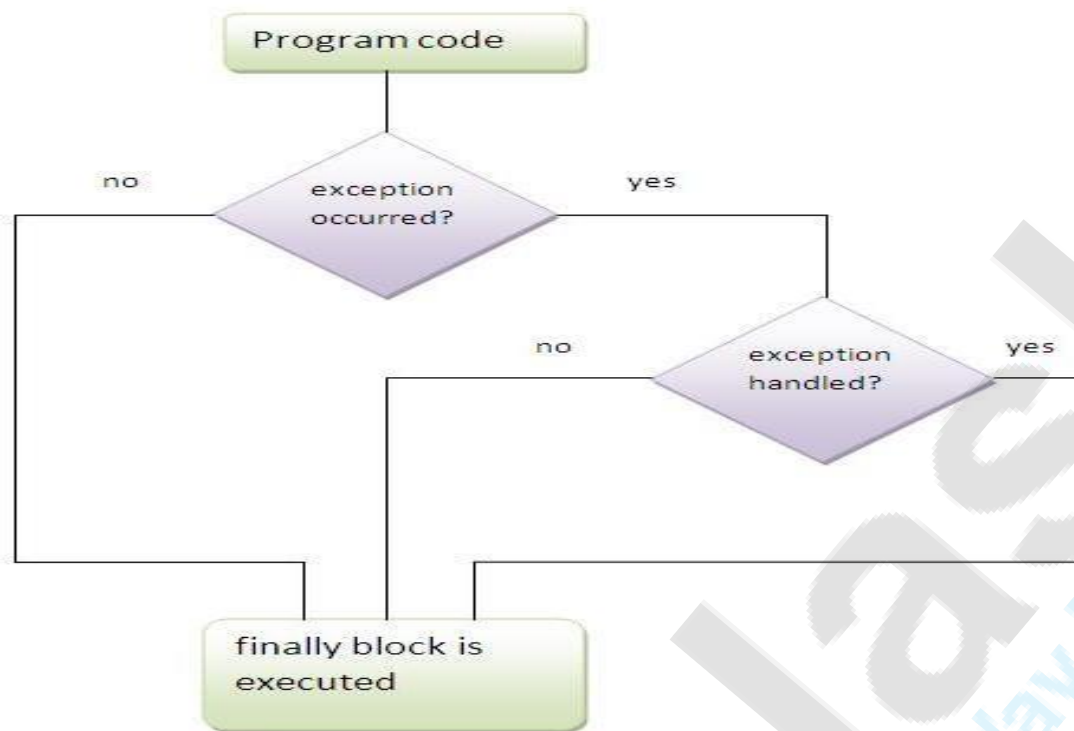
3) Java Finally block

- **Java finally block** is a block that is used to *execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.
- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
- A finally block appears at the end of the catch blocks and has the following syntax –

Syntax

```
try {
    // Protected code
}catch(ExceptionType1 e1) {
    // Catch block
}catch(ExceptionType2 e2) {
    // Catch block
}catch(ExceptionType3 e3) {
    // Catch block
}finally {
    // The finally block always executes.
}
```

Internal working of java finally block



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```

class TestFinallyBlock{
    public static void main(String args[]){
        try{
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
  
```

Output:5

finally block is always executed
rest of the code...

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
```

Test it Now

Output:finally block is always executed

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

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

4) Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.
- The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

1. **throw new** IOException("sorry device error);

Java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
    static void validate(int age){
        if(age<18)
            throw new ArithmeticException("not valid");
        else
            System.out.println("welcome to vote");
    }
    public static void main(String args[]){
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Test it Now

Output:

Exception in thread main java.lang.ArithmeticException:not valid

5)Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

1. return_type method_name() **throws** exception_class_name{
2. //method code
3. }

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
```



```

void m()throws IOException{
    throw new IOException("device error");//checked exception
}
void n()throws IOException{
    m();
}
void p(){
    try{
        n();
    }catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[]){
    Testthrows1 obj=new Testthrows1();
    obj.p();
    System.out.println("normal flow...");
}
}

```

Test it Now

Output:

exception handled

normal flow...

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.
2. **Case2:** You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```

import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
public class Testthrows2{
    public static void main(String args[]){
        try{
            M m=new M();
            m.method();
        }catch(Exception e){System.out.println("exception handled");}

        System.out.println("normal flow...");
    }
}

```

Test it Now

Output:exception handled

normal flow...

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();
        System.out.println("normal flow...");
    }
}
```

Test it Now

Output:device operation performed
normal flow...

B)Program if exception occurs

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();
        System.out.println("normal flow...");
    }
}
```

Test it Now

Output:Runtime Exception

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.

2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
void m(){
    throw new ArithmeticException("sorry");
}
```

Java throws example

```
void m()throws ArithmeticException{
    //method code
}
```

Java throw and throws example

```
void m()throws ArithmeticException{
    throw new ArithmeticException("sorry");
}
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
class FinalExample{
    public static void main(String[] args){
        final int x=100;
        x=200;//Compile Time Error
    }
}
```

Java finally example

```
class FinallyExample{
    public static void main(String[] args){
        try{
            int x=300;
        }catch(Exception e){System.out.println(e);}
        finally{System.out.println("finally block is executed");}
    }
}
```

```
}}
```

Java finalize example

```
class FinalizeExample{
public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}}
```

5.6) CREATING USER DEFINED EXCEPTIONS

- If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
class InvalidAgeException extends Exception{
InvalidAgeException(String s){
super(s);
}
}
class TestCustomException1{
static void validate(int age)throws InvalidAgeException{
if(age<18)
throw new InvalidAgeException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[]){
try{
validate(13);
}catch(Exception m){System.out.println("Exception occurred: "+m);}
System.out.println("rest of the code...");
}
}
```

Test it Now

Output:Exceptionoccured: InvalidAgeException:not valid
rest of the code...

Example of User defined exception in Java

```
/* This is my Exception class, I have named it MyException
* you can give any name, just remember that it should
* extend Exception class
*/
class MyException extends Exception{
String str1;
/* Constructor of custom exception class
```

* here I am copying the message that we are passing while
 * throwing the exception to a string and then displaying
 * that string along with the message.
 */

```
MyException(String str2){
    str1=str2;
}
public String toString(){
    return("MyException Occurred: "+str1);
}
}
class Example1 {
public static void main(String args[]){
    try{
        System.out.println("Starting of try block");
        // I'm throwing the custom exception using throw
        throw new MyException("This is My error Message");
    }
    catch(MyException exp){
        System.out.println("Catch Block");
        System.out.println(exp);
    }
}
}
```

Output:

Starting of try block

CatchBlock

MyExceptionOccurred:This is My error Message

Explanation:

You can see that while throwing custom exception I gave a string in parenthesis (`throw new MyException("This is My error Message");`). That's why we have a parameterized constructor (with a String parameter) in my custom exception class.

Notes:

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.

Another Example of Custom Exception

In this example we are throwing an exception from a method. In this case we should use throws clause in the method signature otherwise you will get compilation error saying that "unhandled exception in method". To understand how throws clause works, refer this guide: [throws keyword in java](#).

```
class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}
```

```
public class Example1
```

```
{
void productCheck(int weight)throws InvalidProductException{
    if(weight<100){
        throw new InvalidProductException("Product Invalid");
    }
}
}
public static void main(String args[])
{
    Example1 obj=new Example1();
    try
    {
        obj.productCheck(60);
    }
    catch(InvalidProductException ex)
    {
        System.out.println("Caught the exception");
        System.out.println(ex.getMessage());
    }
}
}
```

Output:

Caught the exception
ProductInvalid

5.7) ASSERTION IN JAVA

- Assertion is a statement in java. It can be used to test your assumptions about the program.
- While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.
- An assertion allows testing the correctness of any assumptions that have been made in the program.
- Assertion is achieved using the **assert** statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named **AssertionError**. It is mainly used for testing purposes during development.

Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

Syntax of using Assertion:

There are two ways to use assertion. First way is:

1. **assert** expression;
and second way is:
1. **assert** expression1 : expression2;

Simple Example of Assertion in java:

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



```
Scanner scanner = new Scanner( System.in );
System.out.print("Enter ur age ");
int value = scanner.nextInt();
assert value >= 18: " Not valid";
System.out.println("value is "+value);
}
```

If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used.

Compile it by: **javac AssertionExample.java**

Run it by: **java -ea AssertionExample**

Output: Enter ur age 11

Exception in thread "main" java.lang.AssertionError: Not valid

Example of Assertion:-

// Java program to demonstrate syntax of assertion
import java.util.Scanner;

```
class Test
{
    public static void main( String args[] )
    {
        int value = 15;
        assert value >= 20: " Underweight";
        System.out.println("value is "+value);
    }
}
```

Run on IDE

Output:

value is 15

After enabling assertions

Output:

Exception in thread "main" java.lang.AssertionError: Underweight

Enabling Assertions

By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

java -ea Test

Or

java -enableassertions Test

Here, Test is the file name.

Disabling Assertions

The syntax for disabling assertions in java are:

java -da Test

Or

java -disableassertions Test

Here, Test is the file name.

Why	to	Use	Assertion?
-----	----	-----	------------

Wherever a programmer wants to see if his/her assumptions are wrong or not.

- To make sure that an unreachable looking code is actually unreachable.
- To make sure that assumptions written in comments are right.

```
if ((x & 1) == 1)
```

```
{ }
```

```
else // x must be even
```

```
{ assert (x % 2 == 0); }
```

- To make sure default switch case is not reached.
- To check object's state.
- In the beginning of the method
- After method invocation

Where to use Assertions

- Arguments to private methods. Private arguments are provided by developer's code only and developer may want to check his/her assumptions about arguments.
- Conditional cases.
- Conditions at the beginning of any method.

Where not to use Assertion:

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. `IllegalArgumentException`, `NullPointerException` etc.
2. Do not use assertion, if you don't want any error in any situation.
3. Assertions should not be used to replace error messages
4. Assertions should not be used to check arguments in the public methods as they may be provided by user. Error handling should be used to handle errors provided by user.
5. Assertions should not be used on command line arguments.

5.8) ANNOTATIONS IN JAVA

Java Custom annotations or Java User-defined annotations are easy to create and use. The `@interface` element is used to declare an annotation. For example:

1. `@interface MyAnnotation{ }`

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

1. **@interface** MyAnnotation{ }

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value();
3. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value() **default** 0;
3. }

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

1. **@MyAnnotation(value=10)**
The value can be anything.
-

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value1();
3. String value2();
4. String value3();
5. }
6. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value1() **default** 1;
3. String value2() **default** "";
4. String value3() **default** "xyz";
5. }

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

1. **@MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")**
-

Built-in Annotations used in custom annotations in java

- @Target
- @Retention
- @Inherited
- @Documented

@Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example to specify annotation for a class

```
@Target(ElementType.TYPE)
```

```
@interface MyAnnotation{
```

```
int value1();
```

```
String value2();
```

```
}
```

Example to specify annotation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
```

```
@interface MyAnnotation{
```

```
int value1();
```

```
String value2();
```

```
}
```

Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.

RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .
-------------------------	---

Example to specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
    int value1();
    String value2();
}
```

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

File: Test.java

//Creating annotation

```
import java.lang.annotation.*;
```

```
import java.lang.reflect.*;
```

```
    @Retention(RetentionPolicy.RUNTIME)
```

```
    @Target(ElementType.METHOD)
```

```
    @interface MyAnnotation{
```

```
        int value();
```

```
    }
```

//Applying annotation

```
class Hello{
```

```
    @MyAnnotation(value=10)
```

```
    public void sayHello(){System.out.println("hello annotation");}
```

```
}
```

//Accessing annotation

```
class TestCustomAnnotation1{
```

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

```
        Hello h=new Hello();
```

```
        Method m=h.getClass().getMethod("sayHello");
```

```
        MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
```

```
        System.out.println("value is: "+manno.value());
```

```
    }}
```

Test it Now

Output: value is: 10

[download this example](#)

How built-in annotations are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

1. @Inherited
2. **@interface** ForEveryone { }//Now it will be available to subclass also
- 3.
4. **@interface** ForEveryone { }
5. **class** Superclass{ }
- 6.
7. **class** Subclass **extends** Superclass{ }

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

UNIT 6

MULTI THREADING

Explain Java Thread Model

The [Java](#) language and its run-time system was designed keeping in mind about multithreading. The run-time system depend upon multithreading. Java provides asynchronous thread environment, this helps to increase the utilization of [CPU](#).

Multithreading is best in all cases in contrast with single-thread model. Single-thread system uses an approach of event loop with polling. According to this approach a single thread in the system runs in an infinite loop. Polling the mechanism, that selects a single event from the event queue to choose what to do next. As the event is selected, then event loop forwards the control to the corresponding required event handler. Nothing else can be happened, until the event handler returns. Because of this CPU time is wasted. Here, only one part of the complete program is dominating the whole system, and preventing the system to execute or start any other process. In single-thread model one thread blocks all other threads until its execution completes. On other waiting or idle thread can start and acquire the resource which is not in use by the current thread. This causes the wastage of resources.

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- 2) You **can perform many operations together so it saves time**.
- 3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.

- Thread is lightweight.
- Cost of communication between the thread is low.

Java's multithreading provides benefit in this area by eliminating the loop and polling mechanism, one thread can be paused without stopping the other parts of the program. If any thread is paused or blocked, still other threads continue to run.

As the process has several states, similarly a thread exists in several states. A thread can be in the following states:

Ready to run (New): First time as soon as it gets CPU time.

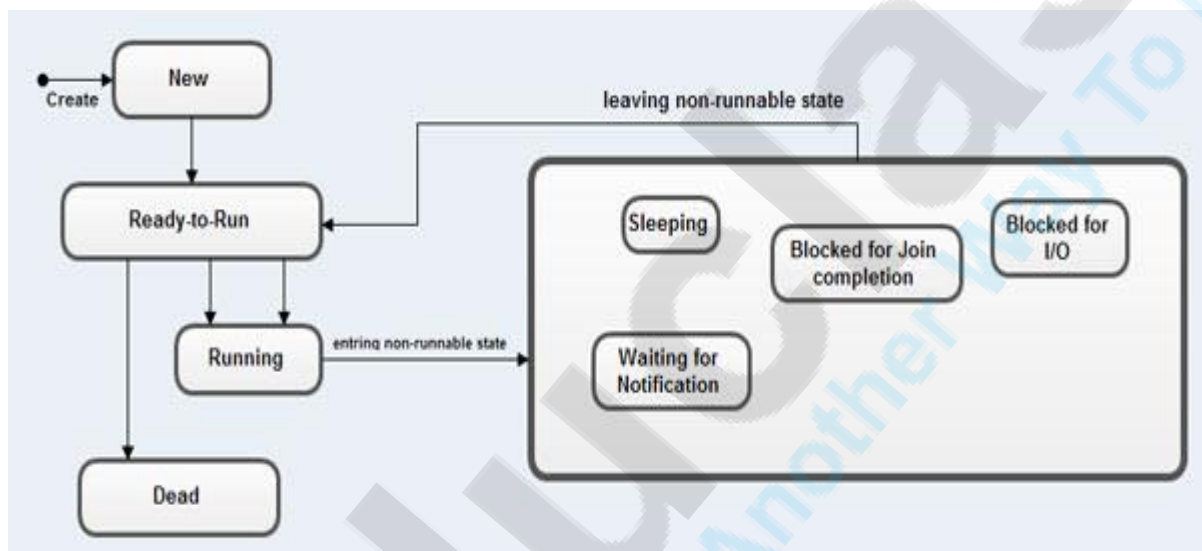
Running: Under execution.

Suspended: Temporarily not active or under execution.

Blocked: Waiting for resources.

Resumed: Suspended thread resumed, and start from where it left off.

Terminated: Halts the execution immediately and never resumes.



Java thread model can be defined in the following three sections:

Thread Priorities

Each thread has its own priority in Java. Thread priority is an absolute integer value. Thread priority decides only when a thread switches from one running thread to next, called *context switching*. Priority does increase the running time of the thread or gives faster execution.

Synchronization

Java supports an asynchronous multithreading, any number of thread can run simultaneously without disturbing other to access individual resources at different instant of time or shareable resources. But some time it may be possible that shareable resources are used by at least two threads or more than two threads, one has to write at the same time, or one has to write and other thread is in the middle of reading it. For such type of situations and circumstances Java implements synchronization model called *monitor*. The monitor was first defined by C.A.R. Hoare. You can consider the monitor as a box, in which only one thread can reside. As a thread enter in monitor, all other threads have to wait until that thread exits from the monitor. In such a way, a monitor protects the shareable resources used by it being manipulated by other waiting threads at the same instant of time. Java provides a simple methodology to implementsynchronization.

Messaging

A program is a collection of more than one thread. Threads can communicate with each other. Java supports messaging between the threads with lost-cost. It provides methods to all objects for inter-thread communication. As a thread exits from synchronization state, it notifies all the waiting threads.

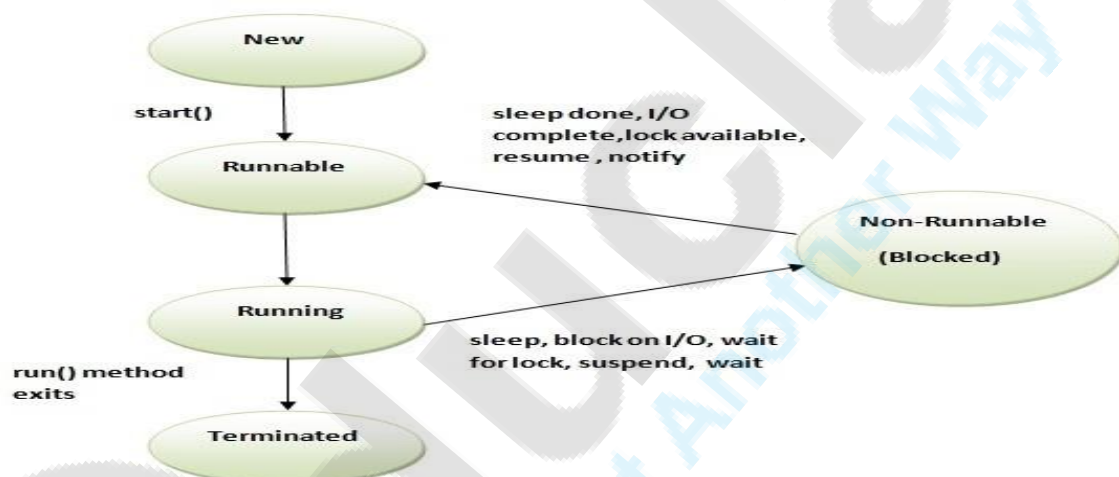
Q. Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Example to illustrate the life-cycle of the thread :

```
public class ThreadDemo extends Thread {
    public void run(){
        System.out.println("Thread is running !!");
    }
    public static void main(String[] args){
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();

        System.out.println("T1 ==> " + t1.getState());
        System.out.println("T2 ==> " + t2.getState());

        t1.start();
        System.out.println("T1 ==> " + t1.getState());
        System.out.println("T2 ==> " + t2.getState());

        t2.start();
        System.out.println("T1 ==> " + t1.getState());
        System.out.println("T2 ==> " + t2.getState());
    }
}
```

Output:

```
T1 ==> NEW
T2 ==> NEW
T1 ==> RUNNABLE
T2 ==> NEW
T1 ==> RUNNABLE
T2 ==> RUNNABLE
Thread is running !!
Thread is running !!
```

Q. Working with Thread class and the Runnable interface

Creating and Starting Threads

Creating a thread in Java is done like this:

```
Thread thread = new Thread();
```

To start the Java thread you will call its start() method, like this:

```
thread.start();
```

This example doesn't specify any code for the thread to execute. The thread will stop again right away after it is started.

There are two ways to specify what code the thread should execute. The first is to create a subclass of Thread and override the run() method. The second method is to pass an object that implements Runnable(java.lang.Runnable) to the Thread constructor. Both methods are covered below.

Thread Subclass

The first way to specify what code a thread is to run, is to create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyThread extends Thread {  
    public void run(){  
        System.out.println("MyThread running");  
    }  
}
```

To create and start the above thread you can do like this:

```
MyThread myThread = new MyThread();  
myThread.start();
```

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

You can also create an anonymous subclass of Thread like this:

```
Thread thread = new Thread(){  
    public void run(){  
        System.out.println("Thread Running");  
    }  
}
```

```
thread.start();
```

This example will print out the text "Thread running" once the run() method is executed by the new thread.

Runnable Interface Implementation

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread. Here is a Java Runnable example:

```
public class MyRunnable implements Runnable {  
  
    public void run(){  
        System.out.println("MyRunnable running");  
    }  
}
```

To have the run() method executed by a thread, pass an instance of MyRunnable to a Thread in its constructor. Here is how that is done:

```
Thread thread = new Thread(new MyRunnable());  
thread.start();
```

When the thread is started it will call the run() method of the MyRunnable instance instead of executing its own run() method. The above example would print out the text "MyRunnable running".

You can also create an anonymous implementation of Runnable, like this:

```
Runnable myRunnable = new Runnable(){
```



```
public void run(){
    System.out.println("Runnable running");
}
```

```
Thread thread = new Thread(myRunnable);
thread.start();
```

Subclass or Runnable?

There are no rules about which of the two methods that is the best. Both methods works. Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Threadinstance. When having the Runnable's executed by a [thread pool](#) it is easy to queue up the Runnableinstances until a thread from the pool is idle. This is a little harder to do with Thread subclasses.

Sometimes you may have to implement Runnable as well as subclass Thread. For instance, if creating a subclass of Thread that can execute more than one Runnable. This is typically the case when implementing a thread pool.

Common Pitfall: Calling run() Instead of start()

When creating and starting a thread a common mistake is to call the run() method of the Thread instead of start(), like this:

```
Thread newThread = new Thread(MyRunnable());
newThread.run(); //should be start();
```

At first you may not notice anything because the Runnable's run() method is executed like you expected. However, it is NOT executed by the new thread you just created. Instead the run() method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code. To have the run() method of the MyRunnable instance called by the new created thread, newThread, you MUST call the newThread.start() method.

Thread Names

When you create a Java thread you can give it a name. The name can help you distinguish different threads from each other. For instance, if multiple threads write to System.out it can be handy to see which thread wrote the text. Here is an example:

```
Thread thread = new Thread("New Thread") {
    public void run(){
        System.out.println("run by: " + getName());
    }
};
thread.start();
System.out.println(thread.getName());
```

Notice the string "New Thread" passed as parameter to the Thread constructor. This string is the name of the thread. The name can be obtained via the Thread's getName() method. You can also pass a name to a Thread when using a Runnable implementation. Here is how that looks:

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");
```

```
thread.start();
System.out.println(thread.getName());
```

Notice however, that since the MyRunnable class is not a subclass of Thread, it does not have access to the getName() method of the thread executing it.

Thread.currentThread()

The Thread.currentThread() method returns a reference to the Thread instance executing currentThread(). This way you can get access to the Java Thread object representing the thread executing a given block of code. Here is an example of how to use Thread.currentThread() :

```
Thread thread = Thread.currentThread();
```

Once you have a reference to the Thread object, you can call methods on it. For instance, you can get the name of the thread currently executing the code like this:

```
String threadName = Thread.currentThread().getName();
```

Java Thread Example

Here is a small example. First it prints out the name of the thread executing the main() method. This thread is assigned by the JVM. Then it starts up 10 threads and give them all a number as name ("" + i). Each thread then prints its name out, and then stops executing.

```
public class ThreadExample {  
  
    public static void main(String[] args){  
        System.out.println(Thread.currentThread().getName());  
        for(int i=0; i<10; i++){  
            new Thread("" + i){  
                public void run(){  
                    System.out.println("Thread: " + getName() + " running");  
                }  
            }.start();  
        }  
    }  
}
```

Note that even if the threads are started in sequence (1, 2, 3 etc.) they may not execute sequentially, meaning thread 1 may not be the first thread to write its name to System.out. This is because the threads are in principle executing in parallel and not sequentially. The JVM and/or operating system determines the order in which the threads are executed. This order does not have to be the same order in which they were started.

Q. Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Explanation of Thread Priority:

1. We can modify the thread priority using the **setPriority()** method.
2. Thread can have integer priority between 1 to 10
3. Java Thread class defines following constants –
4. At a time many thread can be ready for execution but the thread with highest priority is selected for execution
5. Thread have default priority equal to 5.

Thread : Priority and Constant

Thread Priority	Constant
MIN_PRIORITY	1
MAX_PRIORITY	10
NORM_PRIORITY	5

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
    public void run(){
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
    public static void main(String args[]){
        TestMultiPriority1 m1=new TestMultiPriority1();
        TestMultiPriority1 m2=new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

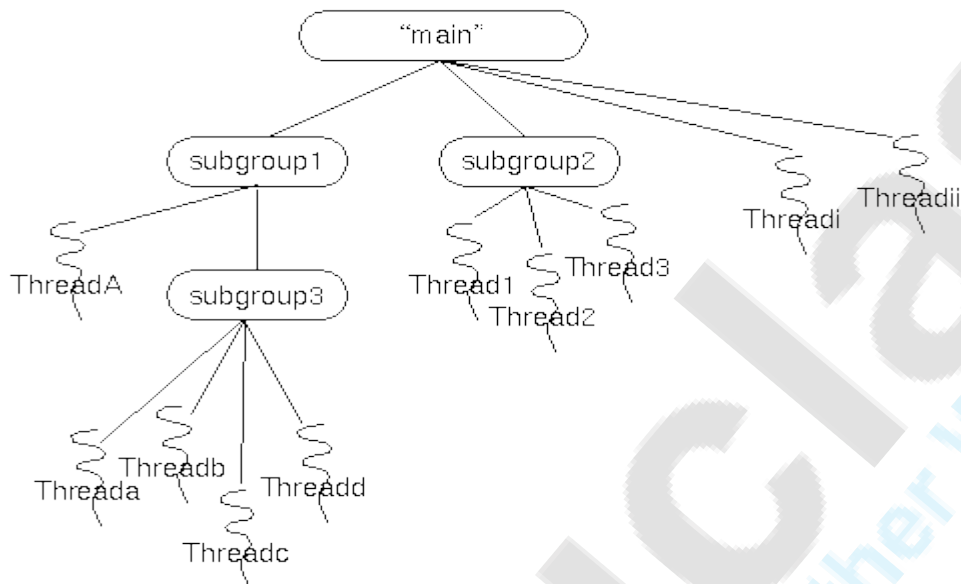
Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

Q. ThreadGroup class

The [ThreadGroup](#)^{api} class manages groups of threads for Java applications.

A ThreadGroup can contain any number of threads. The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.

ThreadGroups can contain not only threads but also other ThreadGroups. The top-most thread group in a Java application is the thread group named main. You can create threads and thread groups in the main group. You can also create threads and thread groups in subgroups of main. The result is a root-like hierarchy of threads and thread groups:



The ThreadGroup class has methods that can be categorized as follows:

- [Collection Management Methods](#)--Methods that manage the collection of threads and subgroups contained in the thread group.

No.	Method	Description
1)	intactiveCount()	returns no. of threads running in current group.
2)	intactiveGroupCount()	returns a no. of active group in this thread group.
3)	void destroy()	destroys this thread group and all its sub groups.
4)	String getName()	returns the name of this group.
5)	ThreadGroupgetParent()	returns the parent of this group.
6)	void interrupt()	interrupts all threads of this group.

7)	void list()	prints information of this group to standard console.
----	-------------	---

- [Methods That Operate on the Group](#)--These methods set or get attributes of the ThreadGroup object.
- [Methods That Operate on All Threads within a Group](#)--This is a set of methods that perform some operation, such as start or resume, on all the threads and subgroups within the ThreadGroup.
- [Access Restriction Methods](#)--ThreadGroup and Thread allow the security manager to restrict access to threads based on group membership.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below. Let's see a code to group multiple threads.

1. ThreadGroup tg1 = `new ThreadGroup("Group A");`
2. Thread t1 = `new Thread(tg1,new MyRunnable(),"one");`
3. Thread t2 = `new Thread(tg1,new MyRunnable(),"two");`
4. Thread t3 = `new Thread(tg1,new MyRunnable(),"three");`

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

File: ThreadGroupDemo.java

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable, "one");
        t1.start();
```

```
Thread t2 = new Thread(tg1, runnable, "two");
t2.start();
Thread t3 = new Thread(tg1, runnable, "three");
t3.start();

System.out.println("Thread Group Name: "+tg1.getName());
tg1.list();

    }
}
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
Thread[one,5,Parent ThreadGroup]
Thread[two,5,Parent ThreadGroup]
Thread[three,5,Parent ThreadGroup]
```

Q. Inter-thread communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

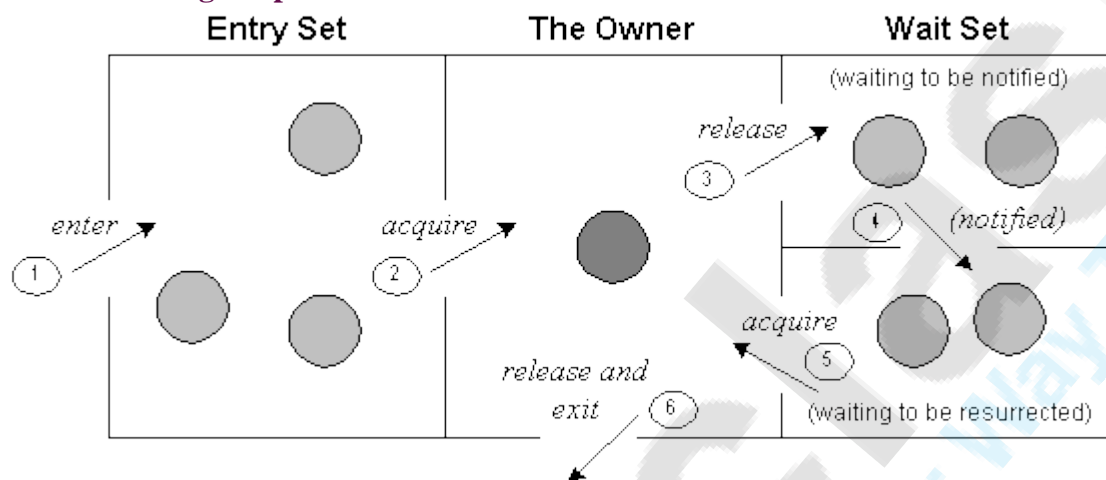
2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:
`public final void notify()`

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:
`public final void notifyAll()`

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
    int amount=10000;

    synchronized void withdraw(int amount){
        System.out.println("going to withdraw...");

        if(this.amount<amount){
            System.out.println("Less balance; waiting for deposit...");
        }
    }
}
```



```

try{ wait();}catch(Exception e){ }
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

```

```

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

```

```

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
}
}

```

Output: going to withdraw...

Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method

is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Q.Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement –

Syntax

```
synchronized(objectidentifier) {
    // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

//example of java synchronized method

```
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }
}
```

```
class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
        this.t=t;
    }
}
```

```
}  
public void run(){  
    t.printTable(5);  
}  
  
}  
class MyThread2 extends Thread{  
    Table t;  
    MyThread2(Table t){  
        this.t=t;  
    }  
    public void run(){  
        t.printTable(100);  
    }  
}  
  
public class TestSynchronization2{  
    public static void main(String args[]){  
        Table obj = new Table();//only one object  
        MyThread1 t1=new MyThread1(obj);  
        MyThread2 t2=new MyThread2(obj);  
        t1.start();  
        t2.start();  
    }  
}
```

Output: 5

10
15
20
25
100
200
300
400
500