## ✱ Throws keyword :-

- required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

## ✱ Access Modifiers :-

| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| Same class | ✓ | ✓ | ✓ | ✓ |
| Same package subclass | X | ✓ | ✓ | ✓ |
| Same package non-subclass | X | ✓ | ✓ | ✓ |
| Diff. package subclass | X | X | ✓ | ✓ |
| Diff. package non-subclass | X | X | X | ✓ |

✱
```
public abstract class AbstractClass {
    public static void main(String a[]) {
        System.out.println("Hello World");
    }
}
```

o/p :- Hello World

✱ It is allowed to define a public abstract class as long as we don't create objects of it.

* How to access variables of Inner class?

Ex- public class Test {

class A {
int i=9;
}
public static void main (String a[]) {

Test.A obj = new Test().new A();
System.out.println(obj.i);
}
}

o/p:- 9

* An inner class has access to all of the members of its enclosing class, but reverse is not true...
Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

Ex- public class Test {
private int y;
class Hello {
public int foo() {
y=1; *
return y;
}
}

```
public static void main(String a[]){
        Test.Hello h = new Test().new Hello();
        System.out.println(h.foo());
    }
}
```

o/p :- 1

* Here variable y of outer class can be directly accessible from inner class without creating object

Ex-

```
public class Test {
    class Hello {
        private int y;
    }
    public int foo() {
        y = 1;
        return y;
    }
    public static void main(String a[]){
        Test.Hello h = new Test().new Hello();
        System.out.println(h.
            Test t = new Test();
            System.out.println(t.foo());
    }
}
```

// error : can't find symbol

(Improved version)

```
Hello obj = new Hello();
obj.y = 10;
return obj.y;
```

o/p :- error : can't find symbol

* Here variable y of inner class can't be directly accessible from outer class, we have to create object of inner class first then only we can access the variable y.

variable y.

Ex- class A {
members;
class B {
} members;
}
class C {
members;
}

Here:- (i) C can access A's members by using objects
(ii) C can't access B in any way
(iii) B can access A's members directly
(iv) A can access B's members only by objects

* To create the object of static nested class:-
(i) StaticNestedClass obj = new StaticNestedClass();
(ii) OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();

* Static-nested class = static nested class
Non-static nested class = inner class

* To create object of inner class:-

OuterClass.InnerClass = new OuterClass().new InnerClass();

## Points about inner class & nested class :-

i) Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. or static or non-static.

ii) Static nested classes don't have access to the mebers of the enclosing class except static members.

iii) Because an inner class is associated with an instance, it cannot define any static members itself.

iv) Static nested class can only access static members of outer class directly while to access non-static members of outer class we have to create object of outer class.

Ex- public class A {

```
    int y;                          static int y;
    static class B {
        ~~y=10~~ public int meth() {
            y=10;                   A a = new A();
            return y;               a.y=10;
        }                           return a.y;
    }
    p s v m(String a[]) {           o/p:- 10
        B oop = new B();
        S.o.pln(oop.meth());
    }
}
```

o/p:- error: non-static variable can't be referenced from static context

v) Non-static nested class (inner class) can access both private or static (or non-static) members of outer class directly without creating object of outer class

Ex-

```
public class A {
    static int x;
    private int y; int z;
    class B {
        public void foo() {
            A a = new A();
            a.x = 10;
            a.y = 20; a.z = 80;
            S.o.pln(a.x + " " + a.y);
            S.o.p(a.z);
        }
    }
    p s v m(String a[]) {
        A.B obj = new A().new B();
        obj.foo();
    }
}
```

```
x = 10;
y = 20; z = 80;
S.o.pln(x + " " + y);
S.o.pln(z);
```

o/p :- 10 20 80

o/p :- 10 20 80

VI) Four kinds of nested class in Java:-

1. Static class - declared as static member of outer class
2. Inner class - declared as an instance member of outer class.
3. Local Inner class - declared inside an instance method of another class
4. Anonymous Inner class - Like a local inner class, but written as an expression which returns a one-off object

Example of local Inner class:-

```
class Outer {
    void outerMethod() {
        int x = 98;
        S.o.pln("Inside outermethod");
        class Inner {
            void innerMethod() {
                S.o.pln("x = " + x);
            }
        }
        Inner inner = new Inner();
        inner.innerMethod();
    }
    p s v m(String a[]) {
        Outer obj = new Outer();
        obj.outerMethod();
    }
}
```

O/P:- Inside outerMethod
x = 98

V.V.I

★ Method local Inner class can't use local variable of outer method until that local variable is not declared as final till jdk 1.7. Since jdk 1.8, it is possible to access the ~~non-local final~~ non-final local variable of outer class, in method local inner class.

vii) Inner class (non-static nested) can access both static and non-static members of the outer class. A static nested class can access only the static members of the outer class.

8. ans → ① By default, the data members of interface are public whereas the data members of abstract class can be private, protected.
② Abstract class can have final methods, whereas interface can't contain final methods.

9

Interface :-

(i) Interface can't be instantiated but we can make reference of it that refers to the object of its implementing classes.
(ii) All the methods are public & abstract by default. And all the fields are public, static and final by default.
(iii) A class that implements interface must implement all the methods in interface, or else declared abstract.
(iv) A class can ~~extend~~ implement more than one interfaces. And an interface can extend more than one interfaces.
(v) It is used to achieve multiple-inheritance.
(vi) Prior to JDK 8, interface couldn't define implementation. We can now add default & static implementation for interface methods.
(vii) Interfaces are used to achieve abstraction.
(viii) Why we use Interface when we have abstract classes?
— Abstract classes may contain non-final variables, whereas variables in interface are final, public and static by default.
(ix) If a class implements an interface and doesn't implement all methods specified in interface then the class must be declared as abstract.
(x) An interface can't contain a constructor (as it can't be used to create objects).

* Default method can't provide the implementation of Object class in an Interface. bcz every class extends Object class implicitly and it has own implementation so it doesn't require default implementation from interface.

(XI) If we define a default method in interface, and also in our own class that implements interface, then it overrides the method of interface.

(XII) Static interface methods are not inherited by either an implementing class or a subinterface.

(XIII) A nested interface can only be declared public inside an interface. By default it is static in nature.

(XIV) We can declare nested interface as public, default and protected inside a class.

(XV) Since Java 9, we have private methods in an interface.

(XVI) Can we define a class inside an interface?

→ Yes, if we define a class inside the interface, then Java compiler creates a static nested class.

(XVII) A variable in an interface must be initialized at the time of declaration.

(XVIII) Interface can't have any instance and static blocks.

(XIX) Functional Interface – An interface that has exactly one abstract method is called functional interface. It can have 3 kinds of methods : abstract, default, and static methods.
It can have default methods with implementation. A default method can't be abstract. It is used in lambda expressions, constructor and method references.

(XX) If we define an interface with @FunctionalInterface annotation, Java compiler will give an error in case if we define more than one abstract method within the interface.

(XXI) A default method is public by definition and can't be declared with private, protected, static, final or abstract modifiers.

(XXII) By the help of default keyword, we can add new methods to our interface without implementing the new methods of the interface in our implementing class.

Ex –

```
interface A {
    void meth();
    default void m() {                    // new method defined in interface need not to be implement in the implementing class
        S.o.pln("A");
    }
}
class B implements A {
    public void meth() {
        S.o.pln("B");
    }
    p s v m(String a[]) {
        A obj = new B();
        obj.meth();
    }
}
```

o/p – B

# Abstract Class :-

(i) An instance of an abstract class can't be created, but we can make abstract class as a reference variable which refers to subclass object.

(ii) An abstract class can have constructors and it is called when an instance of inherited class is created.

(iii) An abstract class can have constructors and static methods but can't have abstract constructors or abstract static methods.

(iv) Why abstract class have constructors if we don't create object of it?

→ to initialize the non-abstract methods and instance variables.

→ if we don't provide any constructor, then compiler will add default constructor in an abstract class.

→ The constructor inside an abstract class can be called during constructor chaining.

(v) If an abstract class doesn't have any abstract methods then it allows us to create class that cannot be instantiated but can only be inherited.

(vi) Abstract class can also have final methods & static methods

(vii) An abstract class can also be used to provide some implementation of the interface. In such case, we are not forced to override all the methods of interface. But the last inherited class of abstract class must implements all methods of interface plus abstract class.

(viii) Ex- public abstract class A {
public static void main(String a[]) {
S.o.pln("A");
}
}

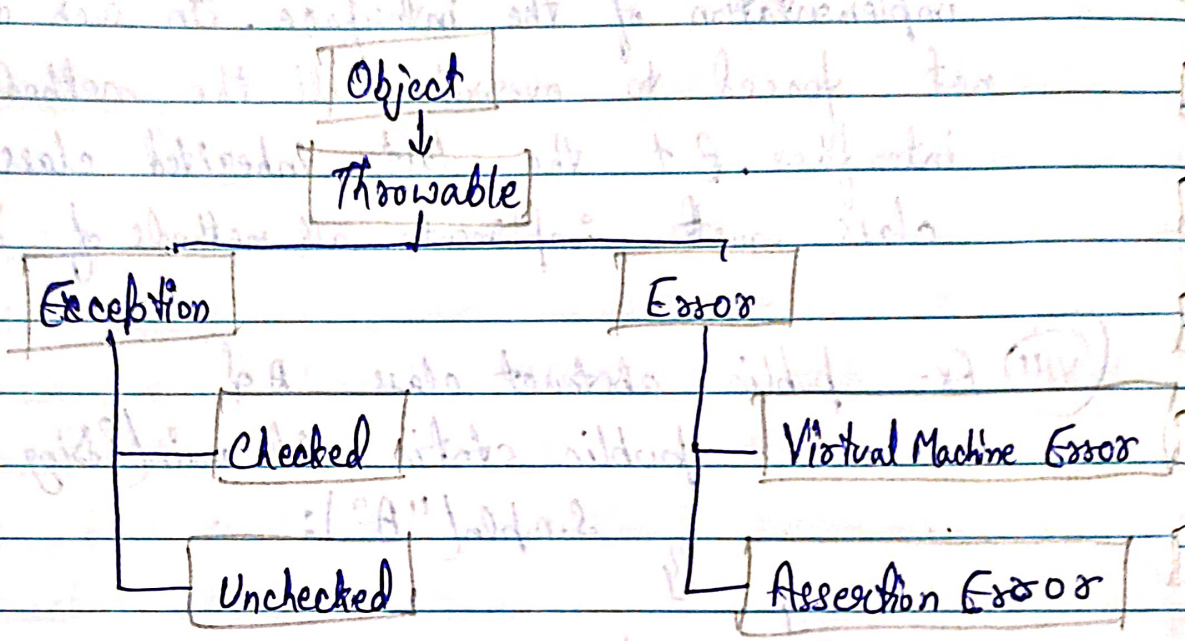
o/p:- A

* It is perfectly allowed to define a public abstract class as long as we don't create objects of it.

(ix) An abstract class can define inner abstract class, inner concrete class, static methods inside of it.

(x)

## Exception Handling :-




(i) Exception can be managed via 5 keywords - try, catch, throw, throws and finally.

(ii) When we use multiple catch statements, then exception subclasses must come before any of their superclasses.

(iii) If no catch statement matches, then the Java run-time system will handle the exception.

(iv) Throws keyword required only for checked (compile-time) exceptions and usage for unchecked (runtime) exceptions is meaningless.

(v) By the help of throws keyword we can provide information to the caller of the method about the exception, that can be thrown from that method.

(vi) Throw keyword is used to transfer control from try block to catch block.

* Catch block is used to handle the exception that occurs in associated try block.
* A try block contains a set of statements where an exception can occur. It must be used within methods.

(VII) A try block is always followed by atleast one catch block or finally block.

(VIII) Unchecked Exception need not be included in any method's throws list bcz these are unchecked exceptions and the compiler doesn't check to see if a method handles or throws these exceptions.

(IX) Checked Exceptions need to be included in a method's throws list if that method can generate one of these exceptions and doesn't handle it itself.

(XVI) If we throw any ~~compile-time~~ checked (checked) exception from try block then the same exception must be handled in the catch block. But if no exception ~~can~~ be thrown from the try block then we can't declare compile-time exception in catch block but we can declare runtime exception in this case.

(XVII) If we throw any ~~runtime~~ unchecked exception from try block then the same exception need not be included in catch block, we can declare any runtime (unchecked) exception but can't declare checked exceptions in the catch block.

(x) If an exception occurred in the code of outer try block, then it will ignore the entire inner try and move directly to its catch block.

(xi) If the superclass overridden method doesn't declare an exception, then subclass overriding method can't declare the checked exception but it can declare unchecked exceptions.

(xii) If the superclass overridden method declares an exception, subclass overriding method can declare same, subclass exception or no exception but can't declare parent exceptions.

(xv) Whatever checked exceptions can be thrown from the try block ~~it~~ must be defined in the catch block. This is not required for unchecked exceptions.

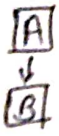
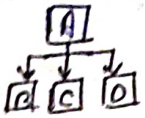
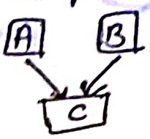
(XII) Child classes of unchecked exceptions are also unchecked exceptions and child classes of checked exceptions are also checked exceptions.

(XIII) No exception of type String, Object, int or char as well as Non-Throwable classes can be thrown. An exception type must be a subclass of Throwable.

There are two ways we can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator.

→ Ex-
```
class A {
    static void meth() {
        try {
            throw new String("Hello");
        }
        catch(Exception e) {
            S.o.p("Exception caught");
            throw e;
        }
    }
    p s v m(String a[]) {
        try {
            meth();
        }
        catch(Exception e) {
            S.o.p("Exception" + e);
        }
    }
}
```

o/p:- compilation error

| * Single:-  | * Multilevel:-  | * Hierarchical:-  | * Multiple:-  |
|---|---|---|---|



## Inheritance :—

(i) Every class has one and only one direct superclass except Object class bcz Object class is the superclass of all classes in java implicitly.

(ii) In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from the subclass. Therefore, it must complete its execution first.

(iii) In case of method overloading, it is important to understand that it is the type of the reference variable - not the type of the object that it refers to - that determines what members can be accessed.

(iv) In case of method overriding, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

(v) It is through superclass reference variables that overriden methods are resolved at runtime.

(vi) When methods are overridden, Java resolves call to an overridden method dynamically at runtime. This is called late-binding. Method overriding is also called runtime-polymorphism.

(VII) When methods are overloaded or the methods declared as final in overriding, a call to one can be resolved at compile time. This is called early binding. Method overloading is also called compile time polymorphism.

(VIII) A subclass doesn't inherit private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

(IX) When subclass object is created, a separate object of super class will not be created, so we can't blindly say that whenever a constructor of class is executed, object of that class is created or not.

(X) In case of static overriding methods, the variable type decides the method being invoked, not the assigned object type.

(XI) Advantages :-

(i) Code reusability

(ii) Method overriding (runtime polymorphism)

(XII) Call to super() must be the first statement in Derived class constructor.

(XIII) Either we define super() in subclass constructor or not, the object of subclass will automatically call the superclass constructor first due to constructor-chaining.

(XIV) If we define a parameterized constructor inside superclass and we create an object of subclass with default constructor, then it throws a compile-time error. We have to declare a no-argument/default constructor inside superclass.

Ex- class A {
A(int a) {
S.o.pln(a);
}
}
class B extends A {
~~B obj = new B();~~
p s v m(String a[]) {
B obj = new B();
}
}

o/p:- compilation error

(XV) A class cannot be extended when a constructor is declared as private.

(XVI) Java doesn't supports multiple Inheritance but with the help of single Inheritance it also supports multiple Inheritance bcz every class is explicitly the subclass of Object class bcz Object class is default.

(17) Super Class = Base Class = Parent class = Ancestor
Sub Class = Derived Class = Child Class = Descendant

★ Can we trigger garbage collection from code?

- The System.gc() command can issue a request to the JVM to prio give priority to garbage collection, but the non-deterministic nature of the garbage collection algorithms means there is no guarantee on when the JVM will respond to such a request.

Common wisdom is to avoid such command (System.gc()) in code and find other ways to configure the JVM's Java garbage collection algorithms.

★ The JVM splits allocated memory into 4 separate spaces:—

① Metaspace ④ Tenured
③ Survivor ② eden

Low-level JVM components, such as StringBuffer and compiled class, are allocated memory in the metaspace.

Garbage collection is done in eden, survivor or tenured spaces.

① When an object is first created, it is placed in the eden space
② If garbage collection occurs and the object is still referenced, it gets moved to the survivor space.
③ If enough garbage collections happen and an object in the survivor space never gets collected, it is then moved to the tenured space.

* Types of garbage collection —

i) A minor garbage collection does a mark-and-sweep routine on the eden space.
ii) A major garbage collection works on the survivor space.
iii) A full garbage collection works on the tenured space.

Since an event that triggers a full garbage collection will normally trigger a sweep of the eden, survivor and metaspace, a full garbage collection is often said to include these areas of the Java heap as well.

* What is the drawback to garbage collection?

- It freezes all active threads when the memory reclamation phase takes place.
- A full garbage collection cycle will run for several seconds -- or potentially even several minutes. Garbage collection routines can't be scheduled. Poorly timed garbage collection can make an enterprise look unpredictable and unreliable.

* There are generally 4 diff. ways to make an object eligible for garbage collection :-

1.) Nullifying the reference variable

```
Test obj = new Test();
obj = null;
```

2) Re-assigning the reference variable

```
A a = new A();
B b = new B();
a = b;
```

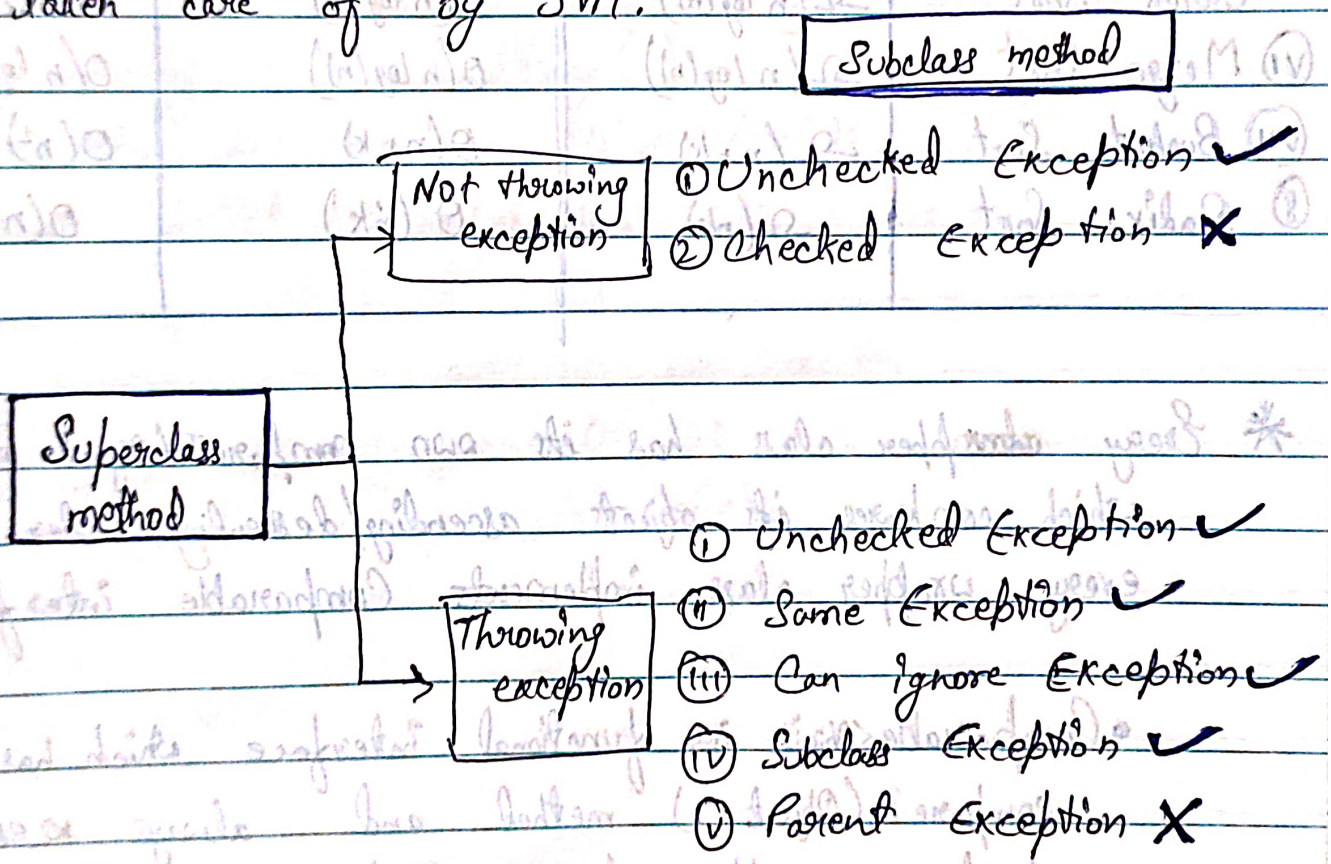
3) Object created inside method :- If some objects were created inside it then these objects becomes unreachable or anonymous after method execution and thus becomes eligible for garbage collection.

```
static void meth() {
    Test t1 = new Test();
}

static void show() {
    Test t2 = new Test();
    meth();
}

p s v m (String a[]) {
    show();
    System.gc();
}
```

NOTE - Here t1 will destroyed first then t2 will destroyed, bcz method calling works in stack frame. When a method is popped out all its members dies.

* Can we force the Garbage collector to run at any time?

No, we can not force Garbage collection in Java. Though we can request it by calling System.gc() or its cousin Runtime.getRuntime().gc(). It's not guaranteed that GC will run immediately as a result of calling these methods. or not guarantee that these requests will be taken care of by JVM.



* Algorithms

| Algorithms | Time Complexity | | |
|---|---|---|---|
| | Best | Average | Worst |
| (i) Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ |
| (ii) Bubble Sort | $\Omega(n)$ | $O(n^2)$ | $O(n^2)$ |
| (iii) Insertion Sort | $\Omega(n)$ | $O(n^2)$ | $O(n^2)$ |
| (iv) Heap Sort | $\Omega(n \log(n))$ | $\Theta(n \log(n))$ | $O(n \log(n))$ |
| (v) Quick Sort | $\Omega(n \log(n))$ | $\Theta(n \log(n))$ | $O(n^2)$ |
| (vi) Merge Sort | $\Omega(n \log(n))$ | $\Theta(n \log(n))$ | $O(n \log(n))$ |
| (vii) Bucket Sort | $\Omega(n+k)$ | $\Theta(n+k)$ | $O(n^2)$ |
| (viii) Radix Sort | $\Omega(nk)$ | $\Theta(nk)$ | $O(nk)$ |

* Every wrapper class has its own compareTo(Object o) method which compares its objects ascending/descending order. bcz every wrapper class implements Comparable interface

- Comparable<Object> is a functional interface which has compareTo(Object o) method and always exists inside wrapper class object. implicitly or we can implement this interface inside our user-defined comparing class. explicitly.

- Comparator<Object> is a functional interface which has compare(Object o1, Object o2) method that doesn't exists implicitly in our comparing class, we have to implement it explicitly or any wrapper class

outside that comparing class to compare two objects in ascending/descending order.

8/1/22

* To convert character array to String :-

```
char c[] = {'I', ' ', 'a', 'm', ' ', 'a', ' ', 's', 't', 'u',
            'd', 'e', 'n', 't', ' ', 'o', 'f', ' ', 'R', 'I', 'T'};
```

① `String s = new String(c);`

② `StringBuilder sb = new StringBuilder();`

```
for(int i=0; i<c.length; i++){
    sb.append(c[i]);
}
```

③ `String s = String.valueOf(c);`

④ `String s = String.copyValueOf(c);`

## (2) What is method overloading? Give a simple usecase.

(i) Overloaded methods are differentiated based on the no., order and type of the parameters passed as an argument to the method.

(ii) We can't define more than one method with the same name, order and type of the arguments. It would be a compiler error.

(iii) We can't declare two methods with the same signature and diff. return type as ~~overloading~~ compiler ~~methods~~ doesn't consider return type. It will throw a compile-time error.

(4) Why do we need Method Overloading?

- (i) If we need to do some kind of the operation with diff. ways i.e, for diff. inputs.
- (ii) It is hard to find many meaningful names for a single action.

(5) Diff. ways of doing overloading methods -

(i) by changing no. of parameters in two methods
(ii) " " order " " " " " "
(iii) " " type " " " " " "

(6) It is the type of the reference variable that determines which method has been called not the type of the object, and this resolves at compile-time. Therefore it is also called early binding.

Ex- class Overload {
void meth() { meth(int a, String str) {
S.o.pln("String :" + str + " int :" + int);
}
void meth(String str, int a) {
S.o.pln("String :" + str + ", int :" + int);
}
}

class Test {
p s v m(String a[]) {
Overload obj = new Overload();
obj.meth("String type", 2);
obj.meth(1, "Alphabets");
}
}

o/p :- String : String type , int : 2
String : Alphabets , int : 1

(3) What is method overriding? What are the diff. ways of implementing it?

(i) When a method in a subclass has the same name, same parameters or signature, and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to be override the method in the superclass.

(ii) Method overriding is one way by which we achieve Run time Polymorphism.

(iii) It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overriden method will be executed.

(iv) It is through superclass reference variables that overridden methods are resolved at runtime. This is called late-binding.

(v) Method in superclass = overriden method
" " subclass = overriding method

(vi) The access modifier for an overriding method can allow more, but not less, access than the overriden method.
Ex- a protected method inside superclass can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

(vii) If we don't want a method to be overridden, we declare it as final.

(VIII) When we define a static method with same signature as a static method in base class, it is known as method hiding.

(9) In case of static methods overridding, it doesn't override it hides the method in their own class. And the reference variable decides which method has been called.

(10) Private methods cannot be overridden as they are bonded during compile time.

(11) The overriding ~~ms~~ method must have same return type (or subtype).

(12) We can call overridden method by the help of super keyword at any time. But to call super constructor ~~we have~~ the super() must be the first statement in subclass constructor.

(13) If the superclass ~~method~~ overridden method does not throws an exception, subclass overridding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

(14) If the superclass overridden method does throws an exception, subclass overriding method can only throws same, subclass or no exception but can't throws parent exception will lead to compile time error.

(15) Overriding can be implemented in 2 ways :-
1) By using Interface
ii) By using Abstract class

Example :- Method overriding using Interface

```
Interface interface A {
        void meth();
    }
class B implements A{
        void meth(){
            S.o.pln("Inside B class");
        }
}
class C implements A{
        void meth(){
            S.o.pln("Inside C class");
        }
}
class Test {
        p s v m(String a[]){
            A a;
            a = new B();
            a.meth();
            a = new C();
            a.meth();
        }
}
```

o/p :- Inside B class
Inside C class

Here, which method has been called is decided at runtime, therefore it supports runtim method overriding.

Example :- Method overriding using Abstract class

```
abstract class A {
    abstract void meth();
}

class B extends A {
    void meth() {
        S.o.pln("Inside class B");
    }
}

class C extends A {
    void meth() {
        S.o.pln("Inside class C");
    }
}

class Test {
    p s v m(String a[]) {
        A a;
        a = new B();
        a.meth();
        a = new C();
        a.meth();
    }
}
```

o/p :- Inside Class B
Inside Class C

Here, which method has been called is resolved at runtime, therefore it also supports method overriding.

## (4) What are the different types of constructors in Java?

(i) Constructors can't be static, abstract, final.
(ii) Constructors shouldn't have any return type, even void also not allowed.
(iii) Constructors can't be override but can be overloaded.
(iv) Constructor's job is to initialize instance variables of an object.
(v) "this" and "super keyword" must be the first name line in the constructor.
(vi) Does constructor return any value?
- Constructor can't return any value, but it returns the current class instance. We can write "return;" inside a constructor.
(7) Constructors having private access modifier prevent users from - creating an object of class
- extend that class
(8) We can create object of class having private constructors within the same class.
(9) Private constructors can be used in singletone classes where the object of the class can't be created outside the class.
(10) Initialization blocks are executed whenever the class is initialized and before constructors are invoked.

(11) Order of execution of Initialization blocks and static blocks and constructors in Java :-

1. Static initialization blocks will run whenever the class is loaded first time in JVM.
2. Instance Initialization Blocks are executed whenever the class is initialized and before constructors are invoked.
3. Each time an object is created using a new() keyword, the constructor is invoked.

(12) There are two types of constructor :-

(1) No-argument constructor - A constructor that has no parameter is known as the default constructor. Default constructor provides the default values to 0, null and false for numeric types, reference types/String, and boolean, respectively.

Once we define our own constructor, the default constructor will no longer used.

Ex- class A {
    A() {
        S.o.println("Constructor called");
    }
    p s v m(String a[]) {
        A a = new A();
    }
}

o/p :- Constructor called

② ~~No Parameterized Constructor~~ - If we want to initialize a constructor with our own

② Parameterized Constructor :- If we want to initialize fields of the class with our own values, then use a parameterized constructor.

```
Ex- class A {
        String name;
        int id;
        A(String name, int id) {
            this.name = name;
            this.id = id;
        }
        p s v m(String a[]) {
            A a = new A("Shruti", 22);
            S.o.pln("Name = "+a.name+", id = "+a.id);
        }
    }
```

o/p :- Name = Shruti, id = 22

(6) Static variables - i) These variables are also known as Class variables.

ii) When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level.

iii) Static variables are, essentially, global variables. All instances of the class share the same static variable.

iv) Static local variables are not allowed in java. Instance

v) Static, ~~local~~ variable is a class variable (for whole class). So if we have static local variable (a variable with scope limited to function), it violates the purpose of static. Hence compiler doesn't allow static local variable.

vi) Static variables can't be overriden. Here, the reference variable decides which variable has been called.

vii) If we access the static variable with the help of object and without the class name, the compiler will automatically append the class name.

viii) In case of static, changes made in variable using one object will be reflected in other objects as static variables are common to all objects of a class.

9) Static variables are can be initialized from constructor or static method or static blocks. But we can't initialize final static variable from the constructor, it is mendatory to initialize static final variables at the time of declaration, otherwise a compile time error is generated.

10) Static variables can be accessed from non-static methods but non static variables can't be accessed from static methods.

11) Static variable can be initialized from constructor, final variable can be initialized from constructor by but final static variable can't be initialized from constructor but it can be initialized only from static block.

12) ~~We can change the value of static variable by using a constructor and static block but not inside a static method.~~

13) Advantage is, It saves memory.

12) We can change the value of static variable which is already initialized by using a constructor and static block but not inside an instance static method.

Example of static variable:-

```
class A {
    static int a=0;
    public void increment() {
        a++;
    }
    p s v m(String a[]) {
        A a = new A();
        A a1 = new A();
        a.increment();
        a1.increment();
        S.o.pln(a.count);
        S.o.pln(a1.count);
    }
}
```

o/p:- 2
2 same

Hence both the objects are sharing a copy of static variable that's why they displayed the same value of 'a'.

## (6) Explain static keyword in java.

(i) Instance variables declared as static are global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of a class share the same static variable.

(ii) Static variables can be accessed from anywhere of the class without creating object of that class.

(iii) Static methods can be accessed within the class directly but outside the class we have to write
Classname.staticMeth();

(iv) Static block executes when class is loaded. It can only be used for initializing static variables or calling any static methods.

(v) Static local variables are not allowed in any way.

(vi) Static variables can't be overridden but if it can be done then reference variables decide which variable will be called.

(vii) If a static variable is already initialized inside a class, then we can at only change the value of this variable inside static blocks or constructor but not inside static method.

(viii) Static final variables can't be initialized from constructor but it can only be initialized from static block.

(ix) We cannot use this or super in static method or static block.

(k) Non-static nested / Inner class can access both static or non-static members of Outer class but static nested class can't access both, it can only access static members of Outer class.

(10) What is the diff. b/w Interface and Abstract class?

| Abstract Class | Interface |
|---|---|
| (1) It can have final & static methods. | (i) It can't have final but have static, private and default methods. |
| (2) A class can extend only one abstract class. So, it doesn't support multiple-inheritance. | (ii) A class can implement multiple Interface. So it supports multiple-inheritance. |
| (3) It can provide the implementation of Interface. or else declared | (iii) It can't provide the implementation of abstract class. |
| (4) It may contain non-final, final, ~~variables~~. static, non-static variables. | (iv) Variables in Interface are by-default final, public and static. |
| (5) Members of abstract class can be private, protected. | (v) Members of Interface are public by default. |
| (6) It can have abstract, non-abstract methods. | (vi) It can only have abstract methods but from java 8 it can have default & static methods. |
| (7) It is a class so it can have state and it can be modified by non-abstract methods | (vii) An Interface can't have the state bcz they can't have instance variables. |
| (8) It can have constructor to initialize instance variables/methods. | (viii) It can't have constructor as it don't have any instance variables. |
| (9) We can't use default keyword inside any abstract/concrete class. | (ix) We use default keyword only inside Interface. |

(12) Explain System.out.println(); in detail.

i) System - It is a final class defined in the java.lang package.

ii) out - This is an instance of PrintStream type, which is a public and static member field of the System class.

iii) println() - As all instances of PrintStream class have a public method println(), hence we can invoke the same on out as well.

Performance Analysis of System.out.println()

println() might be dependent on various factors that drives the performance of this method.

Steps:-

(i) The message passed using println() is passed to the servers' console where kernel time is required to execute the task.

(ii) Kernel time refers to the CPU time.

(iii) Since println() is a synchronized method, so when multiple threads are passed could lead to the low-performance issue.

(iv) System.out.println() is a slow operation as compared to most IO operations.

Alternate way to perform output operations by invoking PrintWriter class

(14) Is String mutable or immutable? How to create mutable String?

String objects are immutable.

* Why?

- When the compiler sees a String literal, it looks for the String in the pool. If a match is found, the reference to the new literal is directed to the existing String and no new String object is created. This makes the existing String simply one more reference. That's why we make the String objects immutable:

In the String Constant Pool, a String object may have one or more references. If a String more than one ~~reference~~ references point to same String without even knowing it, it would be bad if one of the references modified that String value. That's why String objects are immutable in java. A lot of heap space is saved by JRE.

* Now can we override the functionality of String class?
No we can't override any method of String class bcz the class is marked as final.

Ex- String str = "java";
str.concat("rules");
S.o.pln(str);

o/p:- java

What's happening:-
1. The first line creates a new String "java" in String Constant Pool and refer str to it.
2. Next, the VM creates another new String "java rules", but nothing refers to it. So, the second String is instantly lost. We can't access it.
The reference variable str still refers to the original String "java".

It is because almost every method, applied to a String object in order to modify it, creates new String object. It shows that the strings are immutable.

Ex- String s = "abc";
s = s.concat("def");
S.o.pln(s);

o/p- abcdef

Note- String objects are immutable while reference variable is not. So, that's why, the reference was made to refer to a newly formed String object.

(15)

* What is the difference b/w StringBuilder & StringBuffer?

| StringBuffer | StringBuilder |
|---|---|
| (i) StringBuffer is synchronized i.e, thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | (i) StringBuilder is non-synchronized i.e, not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| (ii) StringBuffer is less efficient than StringBuilder | (ii) StringBuilder is more efficient than StringBuffer. |
| (iii) It was introduced in Java 1.0. | (iii) It was introduced in Java 1.5. |
| (iv) StringBuffer objects are stored in Heap area. | (iv) StringBuilder objects are also stored in Heap area. |

(16)

## Difference b/w ArrayList & LinkedList in Java.

| ArrayList | LinkedList |
|---|---|
| (i) This class uses a dynamic array to store the elements in it. It supports the storage of all types of objects. | (i) This class uses doubly-linked list to store the elements in it. It supports the storage of all type of objects. |
| (ii) get(int index) works faster in ArrayList bcz it uses index based system for its elements as it internally uses array data structure. Time Complexity - O(1). | (ii) get(int index) works slower in LinkedList bcz it doesn't provide index based access for its elements as it iterates either from the beginning or end to retrieve the node at specified element index. Time Complexity - O(n). |
| (iii) insert() or add(Object) :- If array is full i.e, worst case, there is extra cost of resizing array and copying elements to the new array, which makes runtime of add operation O(n) otherwise it is O(1). | (3) insert() or add(Object) ie, adding or insertion is faster in LinkedList i.e, O(1). |
| (4) remove(int) method involves copying elements from old array to new updated array, hence its runtime is O(n). | (4) remove() removes the head of the list and runs in constant time O(1). But remove(int) or remove(Object) traverse the LinkedList until it found the object and unlink it |

| | |
|---|---|
| | found the object and return it from the original list. Time Complexity - O(n). |
| ⑤ Iterating the ArrayList in reverse direction is not costly, so we need to write our own code to iterate over the ArrayList in reverse direction. | ⑤ LinkedList can be iterated in reverse direction using descendingIterator(). This method returns an Iterator over the elements. |
| ⑥ If the constructor is not overloaded, then ArrayList creates an empty list of initial capacity 10. | ⑥ LinkedList only constructs the empty list without any initial capacity. |
| ⑦ Memory overhead is less bcz each index only holds the actual object (data). | ⑦ Memory overhead is more bcz node in LinkedList needs to maintain the addresses of next and previous node. |
| ⑧ This class implements List Interface so it acts like a list. | ⑧ This class implements both List and Deque Interface so it acts like List and queue, deque. |
| ⑨ Bef Better storing and accessing data. | ⑨ Better for manipulating data. |
| ⑩ This class works better when the application demands storing the data and accessing it. | ⑩ This class works better when the application demands manipulation of the stored data. |

## (17) Difference b/w HashSet & HashMap.

| HashSet | HashMap |
|---|---|
| ① It implements Set Interface. | ① It implements Map Interface. |
| ② Doesn't allow duplicate values. | ② Duplicate values are allowed but duplicate keys are not. If key is duplicate then the old key is replaced with new value. |
| ③ Requires only one object to store objects. i.e, add (Object o). | ③ Requires two objects i.e, put (K key, V value) to add an element to HashMap Object. |
| ④ In HashSet, the argument passed in add(Object) method serves as key K. Java internally associates dummy value for each value passed in add(object) method. | ④ It doesn't have any concept of dummy values. |
| ⑤ It internally uses HashMap object to store or add the objects. | ⑤ It internally uses hashing to store or add objects. |
| ⑥ It is slower than HashMap. | ⑥ It is faster than HashSet bcz values are associated with a unique key. |

| | |
|---|---|
| ⑦ HashSet uses the add() method for add or storing data. | ⑦ It uses the put() for storing data. |
| ⑧ It can contain a single null value. | ⑧ It can contain a single null key and multiple null values. |
| ⑨ It is used when we need to maintain the uniqueness of data. | ⑨ Always prefer when we don't want to maintain the uniqueness. |
| ⑩ Example :- {1, 2, 3, 4, 5, 6}. | ⑩ Example - {a→1, b→2, c→2, d→1}. as HashMap is a key→value pair (key to value) map. |

of HashSet class

* The add method calls put(key, value) method internally on internal created HashMap object with element we have specified as it's key and constant Object called "PRESENT" as it's value. So, we can say that a Set achieves uniqueness internally through HashMap.

1. If map.put(key, value) returns null, then the statement "map.put(e, PRESENT) == NULL" will return true and element is added to the HashSet (internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement "map.put(e, PRESENT) == null" will return false and element is not added to the HashSet (internally HashMap).

## (19) Diff. b/w LinkedHashMap & TreeMap.

| LinkedHashMap | TreeMap |
|---|---|
| ① It maintains insertion order. | ① It is sorted by natural order of keys. |
| ② Only one null key and multiple null values are allowed. | ② Null keys are not allowed as natural order or Comparator doesn't support comparision on null keys. |
| ③ It implements Map Interface. | ③ It implements Map, Sorted Map Interfaces. |
| ④ Not synchronized but we can use Collections.synchronizedMap(). | ④ Not synchronized but we can use Collections.synchronizedMap(). |
| ⑤ It uses doubly linked lists to store objects. | ⑤ It uses Red black tree (a kind of self-balancing binary search tree) |
| ⑥ It can be used for least Recently Used (LRU) cache, or other places where insertion or access order matters. | ⑥ It can be used where Sorted or navigable features required. |

Ex- Class Test {

```
p.s.vm (String a[]) {
    Map<String, String> mp = new LinkedHashMap<>();
    mp.put("01", "abc");
    mp.put("03", "def");
    mp.put("04", "cat");
    mp.put("02", "rat");
    S.o.pln("LinkedHashMap: " + mp);
    mp = new TreeMap<>();
    S.o.pln("TreeMap: " + mp);
    mp = new HashMap<>();
    S.o.pln("HashMap: " + mp);
}
}
```

O/p:- LinkedHashMap: {01=abc, 03=def, 04=cat, 02=rat}
TreeMap: {01=abc, 02=rat, 03=def, 04=cat}
HashMap: {01=abc, 02=rat, 03=def, 04=cat}

## (20) Diff. b/w HashSet & TreeSet.

| HashSet | TreeSet |
|---|---|
| (1) It does not provide a guarantee to sort the data. | (1) It provides a guarantee to sort the data. |
| (2) Only one ~~ob~~ element can be null. | (2) No null elements are allowed. |
| (3) It uses hashcode() or equals() method for comparison. | (3) It uses compare() and compareTo() method for comparision. |
| (4) It is faster than TreeSet. The add(), remove() and contains methods have constant time complexity O(1) for HashSet. | (4) It is slow than HashSet as TreeSet has to sort the element after each insertion and removal operation i.e, ~~O(~~ log(n). |
| (5) Internally it uses HashMap to store its elements. | (5) Internally it uses TreeMap to store its elements. |
| (6) HashSet is backed up by a hash table. | (6) TreeSet is backed up by a Red-Black Tree. |

(21) What is Singleton class in java? Write code.

① In OOP, a Singleton class is a class that can have only one object (an instance of the class) at a time.

② After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created.

③ Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

④ Advantage - Saves memory bcz object is not created at each request. Only single instance is reused again & again.

⑤ Usage - Singleton pattern is mostly used in multi-threaded and database applications.

⑥ By making constructor private, no class can access the ~~const~~ constructor and hence cannot initialize it. Also Outer class or subclasses should be prevented to create the instance.

⑦ Instance should be globally accessible by making the access-specifier of instance public, so that each class can use it.

Code:-

```
class Singleton {
    private static Singleton object = null;
    private Singleton() { } // private constructor
    public static Singleton getInstance() { // public method to create instance of singleton class
        if(object == null)
            object = new Singleton();
        return object;
    }
}
class Main {
    p. s.v.m (String a[]) {
        Singleton x = Singleton.getInstance();
        Singleton y = Singleton.getInstance();
        Singleton z = Singleton.getInstance();
        if (x == y && y == z)
            S.o.pln("Three objects points to the same memory location
                on the heap area");
        else
            S.o.pln("Three objects points to the diff memory location on
                the heap area");
    }
}
```

o/p:- Three objects points to the same memory location on the heap area.

| Anonymous Inner Class | Lambda Expression |
|---|---|
| ① It is a class without a name. | It is a method without name. |
| ② It can extend abstract and concrete class. | It can't extend abstract and concrete class. |
| ③ It can implement an interface that contains any no. of abstract methods. | It can implement an interface which contains a single abstract method. |
| ④ It can be instantiated. | It can't be instantiated. |
| ⑤ An anonymous inner class object generates a separate class file after compilation that increases the size of a jar file. | A lamba expression is converted into a private method. It saves time and memory. |
| ⑥ We need to write the redundant class definition. | ⑥ We need to provide only the function body. |

| | ArrayList | LinkedList | HashSet | LinkedHashSet | TreeSet | HashMap | LinkedHashMap | TreeMap |
|---|---|---|---|---|---|---|---|---|
| ① Uniqueness | ✗ | ✗ | ✓ | ✓ | ✓ | unique keys duplicate values | same as HashMap | same as HashMap |
| ② Insertion Order | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| ③ Sorted Order | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| ④ Null values | many null values | many null values | one null value | many null values | ✗ | null key - 1 values - many | same as HashMap | ✗ |
| ⑤ Internal working | uses Object[] | uses doubly linked list | uses HashMap | uses LinkedHashMap | uses TreeMap | uses hashCode of key which provides index to store in buckets. | uses doubly-linked list of buckets | uses Red-Black self-balancing binary search tree |
| ⑥ Best Performance | best for search operations. | best for insert and delete operations | best for add(), remove(), get() methods. | best for add(), remove(), get() methods. | best for sorting in natural ascending order. | best for searching items based on key. | best for maintain insertion order or to maintain access order. | best for maintain sorted order i.e, natural ascending order. |

## Functional Interfaces :-

1. Predicate ⇒ test (InputDataType i) , returns boolean
2. Function ⇒ apply (InputDataType i) , returns any type
3. Consumer ⇒ accept (InputDataType i) , void
4. Supplier ⇒ get() , return anytype

1. Predicate<InputDataType> ⇒ take some i/p and perform some conditional checks and return boolean value always

2. Function<InputDataType, ReturnDataType> ⇒ take some i/p and perform some operations and return the result which can be of any type.

3. Consumer<InputDataType> ⇒ accept some i/p and perform required operations and not required to return anything.

4. Supplier<ReturnDataType> ⇒ supply the required o/p and it won't take any i/p.

① Example of Predicate :—

```
class A {
    p s v m (String a[]) {
        Predicate<Integer> p = i -> i%2==0;
        S.o.pln(p.test(3));      // false
        S.o.pln(p.test(4));      // true
        Predicate<Integer> p1 = a -> a>10;
        S.o.pln(p1.test(6));     // false
        S.o.pln(p1.test(20));    // true
        S.o.pln(p.and(p1).test(4));   // false *
        S.o.pln(p.or(p1).test(6));    // true *
        S.o.pln(p1.negate().test(50)); // false
    }
}
```

* and() first checks p condition the p1 i.e, (p && p1)
* or() checks p condition or p1 i.e, (p || p1)
* negate() negotiate p1 i.e, (^p1) it returns always opposite

② Example of Function :—

```
class A {
    p s v m (String a[]) {
        Function<String, Integer> f = s -> s.length();
        S.o.pln(f.apply("Shruti"));   // 6
    }
}
```

* Function Chaining :-

```
class A {
    p s v m(String a[]) {
        Function<Integer, Integer> f1 = i -> i*2;
        Function<Integer, Integer> f2 = i -> i*i*i;
        S.o.pln(f1.andThen(f2).apply(2));   // 64
        S.o.pln(f1.compose(f2).apply(2));   // 16
    }
}
```

* andThen() executes first on f1 and then f2
* compose() executes first on f2 and then f1

(3) Consumer Example :-

```
class A {
    p s v m(String a[]) {
        Consumer<Integer> c = i -> S.o.pln(i*2);
        c.accept(2);        // 4
    }
}
```

(4) Supplier Example :-

```
class A {
    p s v m(String a[]) {
        Supplier<String> s = () -> "Shruti";
        S.o.pln(s.get());   // Shruti
    }
}
```

* Bi Functional Interfaces (that takes two arguments) are :-
  i) BiPredicate
  ii) BiFunction
  iii) BiConsumer

* Primitive Predicate Types:-
  - IntPredicate
  - DoublePredicate
  - LongPredicate

* Primitive Function Types:-
  - DoubleFunction: can take i/p type as Double
    return type can be any type
  - IntFunction: can take i/p type as Int
  - LongFunction: can take i/p type as Long
  - DoubleToIntFunction: i/p type: double
    o/p type: int
    applyAsInt(double value)
  - DoubleToLongFunction: i/p type: double
    return type: long
    applyAsLong(double value)
  - IntToDoubleFunction: i/p type: int
    return type: double
    applyAsDouble(int value)
  - IntToLongFunction: i/p type: int
    return type: Long
    applyAsLong(int value)

(5) UnaryOperator<T> ⇒ if i/p and o/p are of same type then we should go for UnaryOperator
It is child interface of Function<T,T>.

UnaryOperator Example:-

```
class A{
    p s v m(String a[]){
        UnaryOperator<Integer> u = i -> i*i;
        S.o.pln(u.apply(5));   //25
    }
}
```

* Primitive types of UnaryOperator:-
- IntUnaryOperator: public int applyAsInt(int value).
- LongUnaryOperator: public long applyAsLong(long value)
- DoubleUnaryOperator: public double applyAsDouble(double value)

(6) BinaryOperator<T> ⇒ takes two i/ps and of same type and return type is also same.
It is child interface of BiFunction<T,T,T>

BinaryOperator Example:-

```
class A{
    p s v m(String a[]){
        BinaryOperator<String> b = (s1,s2) -> s1+s2;
        S.o.pln(b.apply("shouti","barnwal"));
    }
}
```

* Primitive Types of BinaryOperator :-
  - IntBinaryOperator: public int applyAsInt(int, int)
  - DoubleBinaryOperator: return type: int, method: applyAsDouble(int, int)
  - LongBinaryOperator: return type: long, method: applyAsLong(int, int)

Stream ⇒ The Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result. Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipeline.

There are two categories of methods provided by Stream API i.e, :-

(i) Intermediate methods
(ii) Terminal methods

1) Intermediate methods :- Intermediate operations do not execute until a terminal operation is invoked.

(1) filter(predicate) method
(2) sorted() / sorted(Comparator) method
(3) distinct() method
(4) map() method

* Intermediate operations return a new stream. They are always lazy; executing an intermediate operation such as filter() doesn't actually perform any filtering but creates a new stream that, when traversed, contains the elements of the initial stream that matches the given condition.



① filter() ⇒ this method is used to select elements as per the Predicate passed as argument.

② sorted() ⇒ this method is used to sort the stream.

③ distinct() ⇒ this method returns a stream consisting of the distinct elements.

④ map() ⇒ used to return a stream consisting of the results of applying the given function to the elements of this stream.

* Example of filter() :-

```
class Aq
{
    p s v m(String a[])
    {
        List<Integer> list = new ArrayList<>();
        list.add(10); list.add(51); list.add(31);
        list.add(9); list.add(66); list.add(42);

        List<Integer> evenNumbers = list.stream().filter(
            i -> (i%2==0)).collect(Collectors.toList());
        for(Integer a : evenNumbers)
            S.o.pln(a);
    }
}
```

o/p:- 10
66
42

## * Example of sorted() :-

```
class A{
    p s v m(String a[]){
        List<Integer> list = new ArrayList<>();
        list.add(42); list.add(66); list.add(91);
        list.add(9); list.add(70); list.add(11);

        List<Integer> asc = list.stream().sorted().collect(
                                Collectors.toList());
        for(Integer path : asc)
                S.o.p(path);

        List<Integer> desc = list.stream().sorted((i1,i2)->
                        -i1.compareTo(i2)).collect(
                            Collectors.toList());
        for(Integer d : desc)
                S.o.p(d);
    }
}
```

o/p :- 9 11 42 66 70 91
91 70 66 42 11 9

## * Example of distinct() :-

```
class A{
    p s v m(String a[]){
        List<Integer> list = new ArrayList<>();
        list.add(91); list.add(19); list.add(6);
        list.add(7); list.add(91); list.add(6);

        List<Integer> dist = list.stream().distinct().
                collect(Collectors.toList());
        for(Integer d: list)
            S.o.p(d);
    }
}
```

o/p :- 91, 19, 6, 7

## * Example of map() :-

o/p :- 81 9 49 1 16 64

```
class A{
    p.s.v m(String a[]){
        List<Integer> list = new ArrayList<>();
        list.add(9); list.add(3); list.add(7);
        list.add(1); list.add(4); list.add(8);

        List<Integer> incList = list.stream().map(
                i -> i*i).collect(
                Collectors.toList());
        for(Integer a: incList)
            s.o.p(a);
    }
}
```

* After the terminal operation is performed, the stream pipeline is considered consumed, and no longer be used; Almost all terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning.

2) Terminal methods :- Terminal operations produce a non-stream result such as primitive value, a collection or no value at all. These operations can be performed on Java Stream Directly.

65

List of Terminal methods :-

1. toArray()
2. collect()
3. count()
4. forEach()
5. min()
6. max()

① toArray() ⇒ converts a stream into Object[] array

② collect() ⇒ collect the o/p of the Stream into List or Set

③ count() ⇒ count the elements of a Stream and returns in long type.

④ forEach() ⇒ useful while debugging to print the values of the stream. This is used to perform one action on each value of stream and finally returns nothing.

⑤ min() ⇒ returns the minimum element of the stream based on the provided Comparator.

⑥ max() ⇒ returns the maximum element of the stream based on the provided Comparator.

* Example of toArray() :-

```
class A{
    p s v m(String a[]){
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10); list.add(0); list.add(15);
        list.add(31); list.add(42); list.add(25);

        Integer[] arr = list.stream().toArray(Integer[]::new);
        for(Integer i : arr)
            So.pln(i);
    }
}
```

o/p:- 10
0
5
31
42
25

* Example of collect() :-

```
class A{
    p s v m(String a[]){
        List<Integer> list = new ArrayList<>();
        list.add(42); list.add(89); list.add(21);
        list.add(5); list.add(6); list.add(94);

        List<Integer> evenNumber = list.stream().filter(
                i->i%2==0).collect(
                Collectors.toList());
        for(Integer i: evenNumber)
            So.pln(i);
    }
}
```

o/p:- 42
6
94

* Example of count():-

```
class A{
    p s v m(String a[]){
        List<Integer> marks = new ArrayList<>();
        marks.add(25); marks.add(85); marks.add(81);
        marks.add(94); marks.add(11); marks.add(75);

        long failedStudents = marks.stream().filter(
                                    i -> i<35).count();
        S.o.pln("Failed students are "+failedStudents);
    }
}
```

o/p :- Failed Students are 2

* Example of forEach():-

```
class A{
    p s v m(String a[]){
        List<Integer> list = new ArrayList<>();
        list.add(7); list.add(9); list.add(6);
        list.add(5); list.add(4); list.add(8);

        list.stream().forEach(i -> S.o.pln(i*i));
    }
}
```

o/p :- 49
81
36
25
16
64

* Example of min() & max() :-

```
Class Test9
{
    p s v m(String a[]){
    List<Integer> list = new ArrayList<>();
    list.add(36); list.add(74); list.add(91);
    list.add(24); list.add(11); list.add(100);
ascending order =>  [list.stream().max((i,j) -> i.compareTo(j)).get();]
    Integer maxInAsc = list.stream().max((i,j) -> (i>j)?1:(i<j)?-1:0).get();
    Integer minInAsc = list.stream().min((i,j) -> (i>j)?1:(i<j)?-1:0).get();
 ascending order => [list.stream().min((i,j) -> i.compareTo(j)).get();]
    S.o.pln("Max in Ascending: "+maxInAsc+", Min in Ascending: "+minInAsc);
Descending order => [list.stream().max((i,j) -> -i.compareTo(j)).get();]
    Integer maxInDesc = list.stream().max((i,j) -> (i>j)?-1:(i<j)?1:0).get();
    Integer minInDesc = list.stream().min((i,j) -> (i>j)?-1:(i<j)?1:0).get();
Descending order => [list.stream().min((i,j) -> -i.compareTo(j)).get();]
    S.o.pln("Max in Descending: "+maxInDesc+", Min in Descending: "+minInDesc);
    }
}
```

o/p :- Max in Ascending: 100, Min in Ascending: 11
Max in Descending: 11, Min in Descending: 100

based indexing.

## ★ Diff. b/w Streams & Collections.

| Streams | Collections |
|---|---|
| ① It doesn't store data, it operates on the source data structure i.e, collection | It stores/holds all the data/values that the data structure currently has. |
| ② They use functional interfaces like lambda which makes it a good fit for prog. languages. | They don't use functional interfaces. |
| ③ Streams are consumable i.e, to traverse the stream, it needs to be created every time. | They are non-consumable. i.e, can be traversable multiple times without creating it again. |
| ④ Streams are not modifiable i.e, one we can't add or remove elements from streams. | These are modifiable i.e, we can easily add or remove elements from collections. |
| ⑤ Streams are iterated internally by just mentioning the operations. | Collections are iterated externally using loops. |
| ⑥ No direct way of accessing, elements specific | Elements are easy to access. |

| Checked Exception | Unchecked Exception |
| --- | --- |
| ① Checked Exceptions are checked by compiler during compile-time | Unchecked exceptions are not checked by the compiler instead checked at the runtime. |
| ② All exceptions under Exception class are referred checked exceptions, except RuntimeException and its subclasses and Error. | RuntimeException and Errors classes are together referred unchecked exception and not a part of Exception class. |
| ③ Checked Exceptions must be handled using the try-catch block or throws keyword. | Unchecked Exceptions do not cause any harm if not handled as they do not stop compilation of code. |

## ConcurrentHashMap :—

1) This class is thread-safe i.e, multiple threads can operate on a single object without any complications.

2) At a time any no. of threads are applicable for read operation without locking the ConcurrentHashMap which is not there in HashMap.

3) In this class, the object is divided into a no. of segments according to the concurrency level.

4) The default concurrency level of ConcurrentHashMap is 16.

5) In ConcurrentHashMap, at a time any no. of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is called Segment locking or bucket locking. Hence at a time, 16 update operations can be performed by threads.

6) Inserting null value is not possible in ConcurrentHashMap as a key or value.

Key points :-

- Concurrency-Level :- It is the no. of threads concurrently updating the map. The implementation performs internal sizing to try to accommodate this many threads.
- Load-Factor :- It's a threshold (limit), used to control resizing.
- Initial Capacity :- Accommodation of a certain no. of elements initially provided by the implementation.

  Ex- if initial capacity is 10 it means that it can store 10 entries.

* default value of concurrency level = 16
  " " " initial capacity = 16
  " " " load-factor = 0.75

Key points :-

- Concurrency-Level :- It is the no. of threads concurrently updating the map. The implementation performs internal sizing to try to accommodate this many threads.
- Load-Factor :- It's a threshold (limit), used to control resizing.
- Initial Capacity :- Accommodation of a certain no. of elements initially provided by the implementation.
  Ex- if initial capacity is 10 it means that it can store 10 entries.

* default value of concurrency level = 16
" " " initial capacity = 16
" " " load-factor = 0.75

| Thread | Runnable |
|---|---|
| ① It is a class used to create a thread. | It is a functional interface which is used to create a thread. |
| ② It has multiple methods including start() and run(). | It has only abstract method run(). |
| ③ Each thread creates a unique object and gets associated with it. | Multiple threads share the same objects. |
| ④ More memory required. | Less memory required. |
| ⑤ Overhead of additional methods (as it comes with Thread class) | No overhead of additional methods. |

| run() | start() |
|---|---|
| ① If run() method calls itself, then no new thread will be created and it goes to the current call stack rather than at the beginning of a new call stack. | If start() calls run() method then it always create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM. |
| ② If run() method calls itself then the code inside run() will execute immediately. | If start() calls run(), then the code inside run() will execute after execution of main() method. |
| ③ Multiple invocation is possible. | Can't be invoked more than one times otherwise throws java.lang.IllegalStateException. |
| ④ Defined in java.lang.Runnable interface and must be overridden in the implementing class. | Defined in java.lang.Thread class. |