OOPs Concepts In Java

⇒ OOPS Overview

* OOPs means object oriented programming
* Here object means real world entity like Car, ATM, Bike etc.

Procedural Programming	OOPS
Program is divided into parts called functions.	Program is divided into objects
Doesn't provide a way to hide data, gives importance to function and data moves freely.	Objects provides data hiding, gives importance to data.
Overloading is not possible	Overloading is possible
Inheritence is not possible	Inheritence is possible
Code reusability does not present	Code reusability is present.
Eg:- Pascal, C, FORTRAN etc.	Eg:- Java, C#, Python, C++ etc.

⇒ Objects & Classes

* Object has 2 things :-
  - Properties or State
  - Behavior or Function

For Example:-

* Dog is an object because :-
  - Properties like : Age, colour, breed etc.
  - Behavior like : Bark, sleep, eat etc.

* Car is an object because it has :-
  - Properties like Colour, Type, Brand, Weight etc.
  - Behavior like Apply brake, Drive, Increase speed etc.

* Class is a blueprint / skeleton of an object.
- To create an object, a Class is required.
- So, class provides the template or blueprint from which an object can be created
- From class, we can create multiple objects.
- To create a class, use keyword class :.

Eg:-

```
Class Student
{
    int age;
    string name;
    string address;
```

] Data Variables

```
updateAddress ()
{

}
```

Data Method

```
getAge ()
{
    return age;
}
```

Data Method

Now let's create an object of type Student.

```
Student engStu = new Student ();
```

⇒ 1st Pillar of OOPs - Data Abstraction

* It hides the internal implementation and shows only essential functionality to the user.

* It can be achieved through Interface and abstract classes.

Example:-

- Car: we only show the BRAKE pedal, and if we press it, Car speed will reduce. But HOW? That is ABSTRACTED from us.

- Cellphone: how call is made that is ABSTRACTED to us.

* Advantages of Abstraction:-
- It increases security & confidentiality

DEMO:-

```
Interface Car {

    public applyBrake();
    public incSpeed();
    public handbrake();
    :
    :
}

Class carImp Implements Car
{
    public applyBrake()
    {
        // step-1
        // step-2
        // step-3
        :
    }
}
```

So when user calls applyBrake(), internally it's Invoking step-1, step-2 --- but all that is hidden from the user but ultimately car stops.

So this improves security as user is not aware of the Internal functionality and only Knows about the result.

⇒ 2nd Pillar of OOPS – DATA ENCAPSULATION

* Encapsulation bundles the data & code working on that data in a single unit.
* Also known as DATA-HIDING.
* Steps to achieve encapsulation
  - Declare variable of a class as private
  - Provide public getters & setters to modify & view the values of the variables

* Advantages of encapsulation :-
- Loosely coupled code
- Better access control & security.

DEMO:

```
class Dog
{
    private string Dog;

    string getColour ( )
    {
        return this.colour;
    }

    void setColour ( string colour)
    {
        this.colour = colour;
    }
```

Now let's create an object of Dog type

```
Dog lab = new Dog();
lab.setColour("black");
lab.getColour; // will return black
```

So here we haven't given the access of the variable colour of class dog. Instead we did it with the help of the getter & setter which in turn have the access of variable.

⇒ 3rd Pillar of OOPS - INHERITENCE

* Capability of a class to inherit properties from their parent class.
* It can inherit both functions and variables so that we don't have to write them again in the child class.
* Can be achieved using extends keyword or through interface.
* Types of inheritence:-
  - Single inheritence
  - Multilevel inheritence
  - Hierarchical inheritence
  - Multiple inheritence (Not actually supported by Java due to diamond problem but through interface, we can solve the diamond problem.

* Advantages Of Inheritence
  - Code reusability
  - We can achieve polymorphism using inheritence

DEMO:-

Vehicle (Parent)

↓

Car (child)

```
Class Vehicle
{
    boolean engine;
    boolean getEngine()
    {
      return this.engine;
    }
}

Class Car extends Vehicle
{
    String type;
    String getCarType()
    {
      return this.type;
    }
```

Now let's create an object of Car.

```
Car swift = new Car();
swift.getEngine();

Vehicle vehicle = new Vehicle();
vehicle.getCarType();    // Should not work
```

So, since swift is an object of Car which extends vehicle hence it can call getEngine whereas vice versa isn't possible

* Single

Class A

↓

Class B

* Multilevel

Class A

↓

Class B

↓

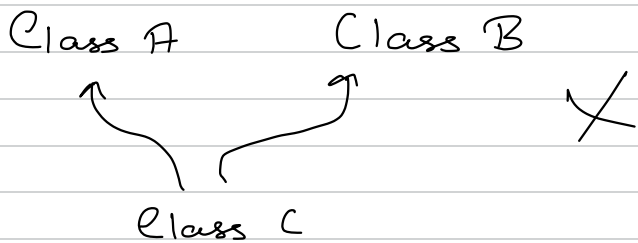
Class C

* Hierarchical

Class A

Class B Class C

* Multiple



This is not supported in Java due to diamond problem but there is a workaround for it using interfaces.

⇒ 4th Pillar of OOPS – POLYMORPHISM

* Poly means "Many" & morphism means "Form"
* A same method, behaves differently in different situation
* Example:
  - A person can be father, husband, employee etc.
  - Water can be liquid, solid or gas.

* Types of polymorphism:-
  - Compile Time / Static Polymorphism / Method Overloading
  - Run Time / Dynamic Polymorphism / Method Overriding

DEMO:

```
Class Sum
{
    int doSum (int a, int b)
    {
        return a+b;
    }

    int doSum (int a, int b, int c)
    {
        return a+b+c;
    }
```

So this practice of creating methods with same name but different parameters is kla overloading.
So the method will be called based on the parameters.

```
Class A
{
    int getEngine ()
    {
        return 1;
    }
```

```
Class B extends A
{
    int getEngine ()
    {
        return 2;
    }
```

Now let's create an object of class B

```
B obj = new B();
obj.getEngine();   // This 'll return 2
```

So which method to call is decided at runtime & this is kla method overriding.
So, in overriding, everything i.e. arguments, return type, method name is same.

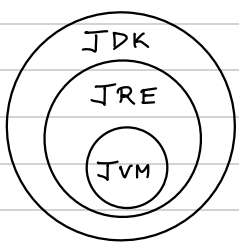
* Is - a relationship
  - Achieved through Inheritence
  - Example: DOG is - a animal.
  - Inheritence form an is - a relation between its parent child classes

* Has - a relation ship
  - Whenever, an object is used in Other class, it's called HAS - A relationship.
  - Relation ship could be one - to - one, one - to - many, many - to - many
  - Example
    - School has students
    - Bike has engine
    - School has classes
  - Association : relationship between 2 different objects
    - Aggregation - Both objects can survive individually, means ending of one object will not end another object.
    - Composition - Ending of one object will end another object.

Java Basics Overview

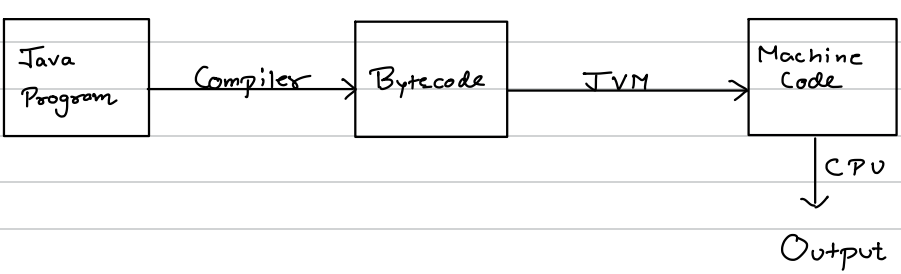
* What is JAVA?
  - Platform Independent Language
  - Supports OOPS
  - Portability [WORA]
    Write Once Run Anywhere

* 3 Main Components Of Java



So JVM, JRE & JDK are 3 main components of Java

⇒ JVM : Java Virtual Machine
- It's just an abstract machine that doesn't exist physically.

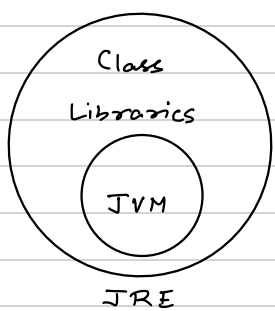


So JVM is platform dependent

So we need to install JVM based on the platform i.e MacOS, Linux or windows. Input for JVM is bytecode & output is machine code. Now since bytecode can be run by any JVM, it makes a Java program platform independent.
JVM has JIT (Just In Time) compiler which takes bytecode & convert it into machine code.

⇒ JRE : Java Runtime Environment

* JRE contains JVM & class libraries i.e the libraries which we've used in the code.



So if we have JRE, we can run any Java Program but we cannot code the program.

⇒ JDK : Java Development Kit
- It has programs language information
- It has compiler (javac)
- It has debugger

So JDK = JRE + (Program Language + compiler + debugger + other dev. components)

So JVM, JRE & JDK all three are platform dependent but the compiled bytecode is platform independent.

* JSE : Java Standard Edition
  JEE : Java Enterprise Edition | Jakarta EE)
  JME : Java Micro/Mobile Edition

JSE is the core Java
JEE = JSE + Servlets + JSP + Transaction API + Persistence API
JME = API for mobile applications

JAVA VARIABLES
(Primitive Data Types In Depth)

* What is a variable?
  - It is a container which holds a value
  - How to declare?
    - Datatype Variable Name = value;

      Eg: int x = 1;

      boolean myz = True;

      Datatype — variable name — value

* Java is Static typed language i.e we mandatorily have to define the datatype of a variable.

* Java is a Strongly Typed Language i.e there is a restriction on what value can be assigned to a variable.

⇒ Variable Naming Convention:-
  - Variable name is case sensitive.
  - Variable name can be any legal identifier means can contain Unicode letters & Digits
  - Variable name can start with $, _ (underscore) and letter.
  - Variable name cannot be Java reserved Keyword like "new", "class", "while", "for", "interface", "int", "float" etc.
  - Variable should be small if it contains only 1 word else camel case should be followed.
  - For constant, variable name should be defined in Capital Letters.

Types Of Variables

- **Primitive Type**
  - Char
  - byte
  - short
  - int
  - long
  - float
  - double
  - boolean

  Integral Type: char, byte, short, int, long

  Fractional Type: float, double

- **Non-Primitive / Reference Type**
  - Class
  - interface
  - array
  - string
  - enum

So, there are 8 types of primitive variables :-

1) Char
   - 2 bytes i.e 16 bits
   - Character representation of ASCII values
   - Range : 0 to 65535 i.e. '\u0000' to '\uffff'
   - Default value is '\u0000' i.e NUL

2) byte
   - 1 byte i.e 8 bits
   - Signed 2^s complement
   - Range : -128 to 127

- default value is 0

0 _ _ _ _ _ _ _

↙

positive

1 _ _ _ _ _ _ _

↙

negative

2nd complement = Complement + 1

Let's take an example of 3

So +3 is 0 0 1 1

Now -3 is second complement of +3

i.e +3 0 0 1 1

1 1 0 0 (Complement)

+ 1

1 1 0 1 (2nd Complement)

So if we add +3 & -3

+3	0	0	1	1
-3	1	1	0	1
	0	0	0	0

ie. 0

So in total 7 bits represent number &

last bit represent sign.

3) Short

→ 2 bytes i.e 16 bits

→ Signed 2nd complement

- Range : -32768 to 32767
- Default value is 0

4) int
- 4 bytes ie 32 bits
- Range : -2^{31} to $2^{31}-1$
- Default value is 0
- Signed 2nd Complement

5) Long
- 8 bytes ie 64 bits
- Signed 2nd complement
- Range : -2^{63} to $2^{63}-1$
- Default value is 0
- Eg:- long var = 100l;

  → This l signifies that it is long type.

* We'll discuss fractional type in detail later.

6) boolean
- 1 bit
- Value : True or False
- Default Value is True

* Types of conversion :-

1) Widening / Automatic Conversion

- Automatic conversion when we go from lower data type to higher datatype

byte (1 byte)
short (2 byte)
int (4 bytes)
long (8 bytes)

Eg:- int var = 10;
long varLong = var; // Automatically converted int to long

2) Narrowing / Downcasting / Explicit Conversion

- It is opposite of widening ie going from higher data type to lower datatype.
- In this case downcasting doesn't happen automatically. So, we have to manually do it.

Eg:- int integerVariable = 10;
byte byteVariable = (byte) integerVariable;

- If we're downcasting beyond range then it'll again reset to -128 & it goes on.

So if integerVariable's value is 128 then byteVariable's value will be -128 (Next after 127 is -128 for byte)
& if it's 148 then byteVariable's value will be -108

3) Promotion during expression

- This happens internally during expression
- As soon as value of expression crosses the range of the datatype then promotion happens internally to higher datatype.
- byte & short promotes to int.
- Eg:- byte a = 1

  byte b = 127

  byte sum = a+b; // won't work since rang is crossing

So we'll have to declare it as int as the result will be int.

Although we can explicitly downcast it but value will change as per explicit downcasting terms.

↳ This is kla explicit casting during expression.

- In an expression, if one datatype is of higher datatype, then all other will also be automatically converted to higher data type.

Eg:- int a = 34;

double doubleVar = 20d;

int sum = a + doubleVar; // give error

double sum = a + doubleVar;

⇒ Kind Of Variables :-

* Member / Instance Variable

- It is a variable of the class i.e is created when an object of the class containing it is created. So, each object of the class has its individual copy of member variable.

* Local Variable

- These variables are the variables that are defined inside a method.
- If the method finishes, it gets destroyed.

* Static / Class Variable
- Only one copy of static / class variable exists. All objects can access it using class name.

* Method Parameters
- These are the variables that are passed to a method.

* Constructor Parameters
- These are the variables that are passed to a constructor.

⇒ Fractional Types

* How float & double are stored in memory?
  - Float

1 Bit	8 Bits	23 Bits
Stores sign ie 0 → positive, 1 → negative	Stores exponent	Stores Mantissa / Significant

Eg :- 4.125f

Step - 1 : Convert to binary

4 → 100 & .125 = 0.125 × 2 = 0.25 → 0

.25 × 2 = 0.5 → 0

.5 × 2 = 1.00 → 1

0.125 = 0.001

So binary equivalent of 4.125 = 100.001

Step-2 : Make it in the form of $(1.xxx) \times 2^{\text{exponent}}$

$100.001 \Rightarrow 1.00001 \times 2^{2}$

(Since base is binary, moving 2 decimal left adds 2 as power of 2)

Step-3 : Add bias to the exponent

$100.001 \Rightarrow \underbrace{1.00001}_{\text{Mantissa}} \times 2^{2 \rightarrow \text{exponent}}$

Since bias for float is 127 we'll add it to exponent
ie $127 + 2 = 129$

Step-4 Fill the numbers

$0 \; \underbrace{10000001}_{} \; \underbrace{00001}_{} 000 \cdots 0$

Sign bit — Binary of 129 ie exponent — Since Mantissa is 00001 rest bits are empty.

Now let's convert this binary form back to float

$(-1)^{\text{sign}} \times (1 + \text{mantissa}) \times 2^{\text{exponent} - 127}$

This is reverse of the above process

$(-1)^{0} \times (1 + 00001) \times 2^{129-127}$

$1 \times (1 + 2^{-5}) \times 2^{2}$

$1 \times (1 + 0.03125) \times 4$ ie $\underline{4.125}$

Now for 0.7f

Step-1

$0.7 \times 2 = 1.4 \quad 1$

$.4 \times 2 = 0.8 \quad 0$

$.8 \times 2 = 0.6 \quad 1$

$.6 \times 2 = 0.2 \quad 1$

$.2 \times 2 = 0.4 \quad 0$

$.4 \times 2 = 0.8 \quad 0$

$.8 \times 2 = 0.6 \quad 1$

⋮

Keeps repeating

So binary is 0.10110 0110 0110 0110

Step-2 : It becomes $(1.0110\ 0110\ 0110) \times 2^{-1}$

-1 since decimal shifted 1 pos^n to right

Step-3 : $-1 + 127 = 126$ (Add 127 bias)

Step-4: Now assemble / store

$$\underset{\text{Sign}}{0}\ \underbrace{01111110}_{\text{Exponent}}\ \overset{2^{-1}\ -2\ -3\ -4\ -5\ \cdots}{\underbrace{0110\ 0110\ 0110\ 0110\ 0110\ 011}_{\text{Mantissa}}}$$

& if we revert back to float

$$(-1)^0 \times (1 + \text{mantissa}) \times 2^{126-127}$$

$$\text{Mantissa} = \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^{10}} + \frac{1}{2^{11}} + \frac{1}{2^{14}} + \frac{1}{2^{15}} + \frac{1}{2^{18}} + \frac{1}{2^{19}} \cdots$$

So mantissa will be 0.399414062

so $1 \times (1 + 0.399414062) \times 1^{-1}$

$= \underbrace{0.699707031}$

So here value comes out to be less than 0.7 when we stored, so we generally use Big Decimal instead of float

* This is same for double as well, just that double is 64 Bits

JAVA VARIABLES
(Non Primitive Data Types In Depth)

⇒ Reference Data Types / Non-Primitive Data Types

* There are mainly 4 type of reference data types:
  - Class
  - String
  - Interface
  - Array

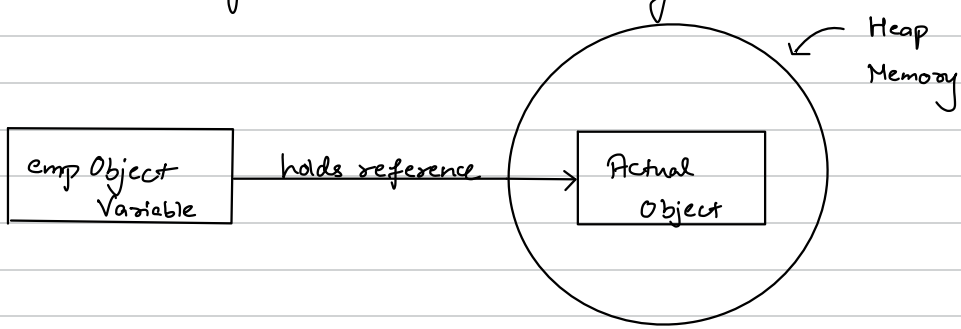
* What is reference?
Let's understand with an example of class
So let's create a class

```
public class Employee {

    int empId;

    public int getEmpId() {
        return empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }
}
```

Now to create an object of class Employee

```
public class Student {

    public static void main(String args[]){

        Employee empObject = new Employee();
    }

}
```

new keyword allocates a memory block object & the variable's name holds a reference to actual memory.

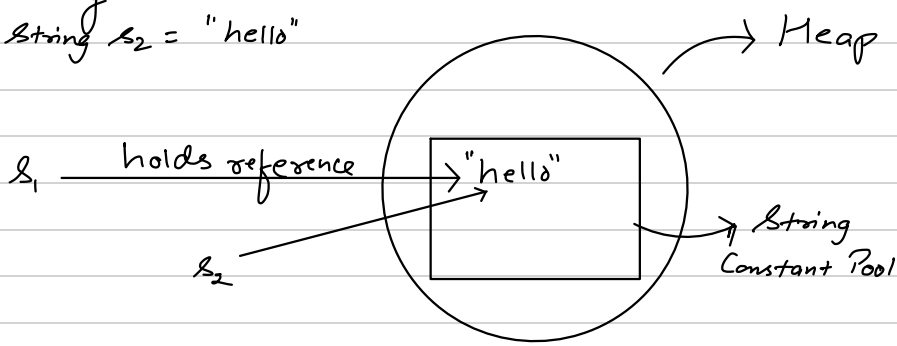


* In Java, everything is pass by value. So with the help of reference variables we're achieving the functionality of pointers in CPP.

* String
  - Strings are immutable in Java
  - It contains String literal.

Inside heap, there is a fixed memory space kla String Constant Pool so the String variable holds a reference of corresponding String literal in String Constant Pool.

Eg:- String s1 = "hello".
String s2 = "hello"



Since strings are immutable, 2 same strings holds reference to same string literal in String Constant Pool
So both s_1 & s_2 holds reference to same string literal "hello".
This applies for normal strings. If we create using new keyword, it'll be stored as normal object in heap.
Eg:- String s_3 = new String ("hello")

* Interface
To understand better, let's create an interface & implement it.

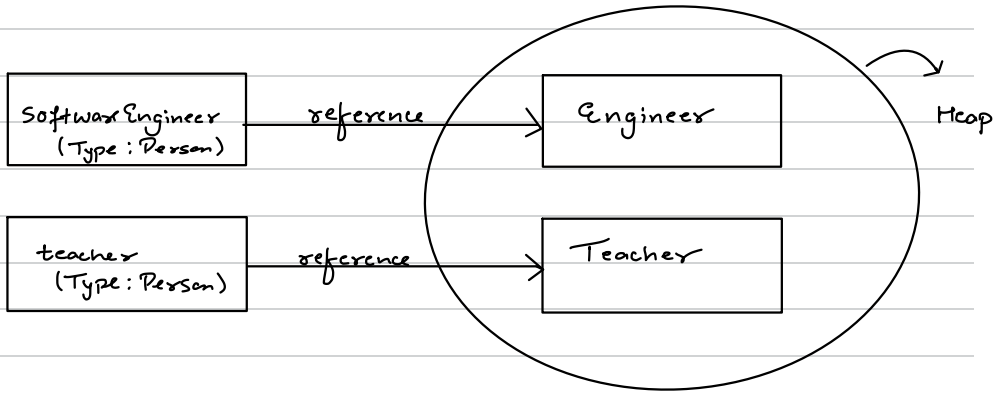
```
public interface Person {

    public String profession();
}
```

```
public class Teacher implements Person{

    @Override
    public String profession() {
        return "teaching";
    }
}
```

```
public class Engineer implements Person{

    @Override
    public String profession() {
        return "software engineer";
    }
}
```

Now let's create a few objects in a class:-

```
public class Student {
    public static void main ( String args [ ] ) {
        Person softwareEngineer = new Engineer ( );
        Person teacher = new Teacher ( );
        Teacher teacher1 = new Teacher ( );
        Engineer softwareEngineer1 = new Engineer ( );
```



So here the variables softwareEngineer & teacher holds a reference of Engineer & Teacher type objects in the heap memory.
So we can store the objects of a child in a parent one or we can store the objects in same class itself but we cannot create an object of interface.

ie $\underbrace{\text{Person person = new Person();}}$

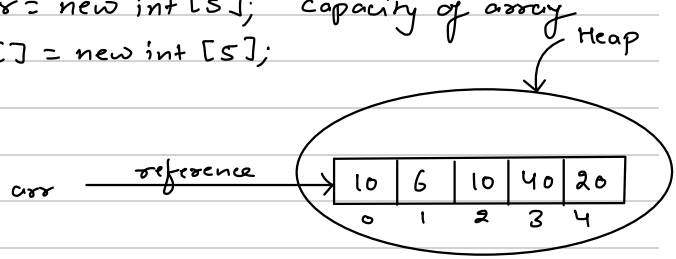
This is wrong.

⇒ Array

- Sequence of memory storing same datatype.

For eg:- int [] arr = new int [5]; capacity of array

or int arr [] = new int [5];



arr[0] = 10, arr[1] = 6, arr[2] = 10, arr[3] = 40, arr[4] = 20

* Arrays can be assigned in multiple ways & can be of multiple types like 1D, 2D etc.

⇒ Wrapper Class

Autoboxing

Unboxing

For each of the primitive data types, we have corresponding reference types that are known as Wrapper Classes

int	→	Integer
char	→	Character
short	→	Short
byte	→	Byte
long	→	Long
float	→	Float
double	→	Double
boolean	→	Boolean

* Why the need for wrapper classes?
  → We get the advantages of passing by reference. For example if we've declared a wrapper type of int i.e Integer type, we can change it later on & it'll change in memory as well because in wrapper we're storing reference.
  In primitive this won't be possible as they're stored in stack & not heap.

- The collections works on objects only i.e on reference data types, so we need wrapper class to use collections.

* Autoboxing
  - To convert a primitive data type to its wrapper.

    Eg:- int a = 10;

    Integer a1 = a;

    primitive to its wrapper

* Unboxing :-
  - To convert a wrapper class to primitive

    Eg: Integer x = 20;

    int x1 = x;

    wrapper to primitive

⇒ Constant Variable
  - We cannot change the value of a constant variable. This is usually created using final keyword.

    Eg:- static final VAR = 10;

    It means only one copy exist

    It means the value of VAR can't be changed.

METHODS IN JAVA

* What is method?
  - Method is used to perform a certain task
  - Collection of instructions that performs a specific task
  - It can be used to bring the code readability & re-usability

    For eg:-

```java
public class Calculation {

    public int sum(int val1, int val2){
        int  total = val1 + val2;          → Instruction 1
        //doing some logging stuff         → Instruction 2
        System.out.println("addition of val1 and val2 is:" + total);
        return total;   → Instruction 3
    }

    public int getPriceOfPen(){
        int capPrice = 2;
        int penBodyPrice = 5;
        int totalPenPrice = sum(capPrice, penBodyPrice);
        return totalPenPrice;
    }
```

Here sum is a method which will take 2 arguments & return the addition of both the numbers. Since we reused the code of method to perform add" so it provides reusability.

* How to declare method?

Access Specifier — Return Type — Method Name — Method Arguments

```
public int sum (int a, int b) throws Exception {
    // method body
}
```

* Access Specifiers
  - Defines accessibility of a method i.e who can use the method
  - There are 4 types of access specifiers in Java methods:-

1) Public : can be accessed through any class in any package.
2) Private : can be accessed by methods only in the same class
3) Protected : can be accessed by other classes in same package or other sub classes in different package.
4) Default : It can only be accessed by classes in same package. If we do not mention anything, then Default access specifier is used by Java.

* Return Type
  - It tells what type the method will return after computation. If the method don't return anything, void return type is used.
  - Use class name or primitive data types as return type of the method.

* Method Name :
  - It should be verb (or some kind of action)
  - Should start with small letter and follow camel case in case of multiple words

* Method Parameters
  - It's a list of variables that will be used in the method.
  - Parameter list can be blank too.

* Method Body :
  - Method body get finished when you call 'return' in mid.
  - Gets finished when reached to the end
  - We can also stop method by 'return' even for void return type

⇒ Types Of Methods :-

* System Defined Methods :-
  - Methods which are already defined and ready to use in Java like Math.sqrt()

* User Defined Methods :
  - Methods which the programmer create based upon the program necessity

* Overloaded Method :
  - More than one method with same name is created in same class
  - Overloaded Method only gets differentiated based on arguments so name should be same, arguments should be differend and return type is not even considered.

```
public class Invoice {

    void getInvoice() {
        System.out.println("inside invoice method");
        return;
    }

    void getInvoice(String z){

    }

    int getInvoice(int a){

        //doing some other stuff
    }

    void getInvoice(int a, int b){

        //something else
    }

    public void printInvoice(){
        getInvoice();
    }
}
```

* Overridden Method

- Subclass / Child Class has the same method as the parent class.

* Static Methods:

- These methods are associated with the class
- Can be called just with class name.
- Static methods can not access Non Static Instance variables and methods
- Static methods cannot be overridden.

So when to declare method static:

- Methods which do not modify the state of the object can be declared static.
- Utility method which do not use any instance variable and compute only on arguments.
  Example: Factory design pattern

* Final Methods:

- Final Method cannot be overridden in Java. It is so because final method means its implementation cannot be changed. If child class cannot change its implementation then no use of overridden.

* Abstract Method

- It is defined only in abstract class.
- Only method declaration is done.
- Its implementation is done in child classes

* Variable Arguments (Varargs):

- Variable number of inputs in the parameter.
- Only one variable argument can be present in the method.
- It should be the last argument in the list.
- Used when we don't know the number of arguments

For eg:-

```java
public class Calculation {

    static int carPrice = 40;

    public int sum(int a, int ...variable){

        int output = 0;
        for(int var : variable){
            output = output + var;
        }
        return output;
    }

}
```

Variable argument

So while calling, we can give multiple no. of arguments.

```java
public class Main {

    public static void main(String args[]) {

    Calculation calculationObj = new Calculation();
    calculationObj.sum( a: 3);
    calculationObj.sum( a: 3, ...variable: 8,9,10);
    calculationObj.sum( a: 3, ...variable: 8,9,10,3,3,2,2,2,2,2,2);

    }

}
```

CONSTRUCTORS IN JAVA

* What is constructor ?
  - It is used to create an instance / initialise the instance variable
  - It's similar to method except:
    - Name: Constructor name is same as class name
    - Return Type: Constructor do not have any return type.
    - Constructor cannot be static or final or abstract, synchronized.

* new keyword tells java to call constructor

* Why constructor name is same as of class name ?
  - Constructor name is always same as class name because it is easy to identify & there is no return type because implicitly java adds class as return type.

* Why constructor do not have return type ?
  - There can be methods with same name and even class as return type but they cannot be called constructors as they do not obey the rules of constructor ie same name without return type.

* Why constructor cannot be final ?
  - Constructors are different from usual methods & cannot be inherited. So it doesn't make sense to make them final because final is used to prevent overriding & If constructors cannot be inherited then

there is no requirement for final

* Why constructor cannot be abstract?
- Since for abstract method, the responsibility of implementation is of child class. But constructors can't even be inherited so no point of making them abstract.

* Why constructor cannot be static?
- Since static methods can only access static variables & other static methods, so it won't be able to initialise the instance variable. We also won't be able to use constructor choining & call super().

* Can we define constructor in interface?
- No because we cannot create object so no point of constructor.

⇒ Type Of Constructors :-

1) Default Constructor
- When we do not define a constructor, java internally provides a constructor which is known as default constructor. Default constructor also set default values for all the instance variables. It is added only when we do not define a constructor

2) No Argument Constructor
- A constructor that does not take any argument. It is very similar to default constructor but we are defining it instead of java.

3) Parameterised Constructors

- It takes arguments and assign the instance variables with those parameters. We can initialise one or multiple instance variables using a parameterised constructor. For the variables where we don't provide any argument, they'll be instantiated with default values.

4) Constructor overload

- We can create multiple constructor with different parameters.

5) Private Constructor

- We can create a private constructor & no one outside the class will be able to call the constructor. This is used usually in Singleton design pattern. To create an object of a class having private constructor, we can create another static method to create the object & then call that method using class name.

⇒ Constructor Chaining

* It means that we can call one constructor in other constructor. This is done using this() & super(). To chain a constructor within the same class, this() is used. For eg:

Calculation.java | Calculation.class | Main.java

```java
public class Calculation {

    String name;
    int empID;

    Calculation(){
        this( empID: 10);
    }


    Calculation(int empID){

        this( name: "sj", empID);
    }

    Calculation(String name, int empID){

        this.name = name;
        this.empID = empID;
    }

```

Here we called other constructor within a constructor using this().

* Using super()
  - The constructor of a child class always invokes the constructor of parent class first & then invokes its own constructor. This is done using super(), so if we explicitly don't add super() in child constructor then Java adds it internally

So this() & super() are used for constructor chaining.

* If the parent class has a parameterised constructor, then we'll have to mandatorily pass an argument to super() to call the parent class's parameterised constructor.

JAVA MEMORY MANAGEMENT

* There are 2 types of memory which Java creates. JVM manages these both:-
  1) Stack
  2) Heap

⇒ Stack Memory
- Stack stores temporary variables & separate memory block for methods.
- Store primitive data types.
- Store reference of heap objects
  - Strong reference
  - Weak reference
    - Soft reference
- Each thread has its own stack memory
- Variables within a SCOPE is only visible and as soon as any variable goes out of the SCOPE, it gets deleted from the stack (in LIFO order)
- When stack memory goes full, it throws "java.lang.StackOverflowError"

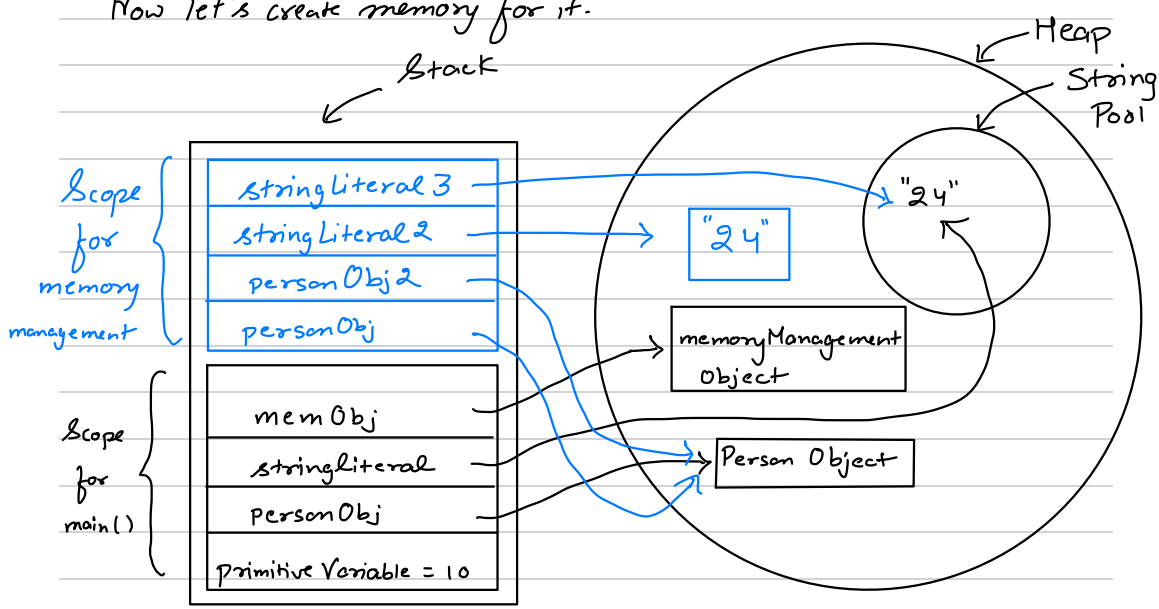
⇒ Heap Memory
- Store objects & There is no order of allocating the memory
- Garbage Collector is used to delete unreferenced objects from the heap.
  - Mark & Sweep Algorithm
  - Types of Garbage Collector i.e Single GC, Parallel GC, G1 and CMS (concurrent mark & sweep)
- Heap memory is shared with all the threads.

* Now let's understand both with the example

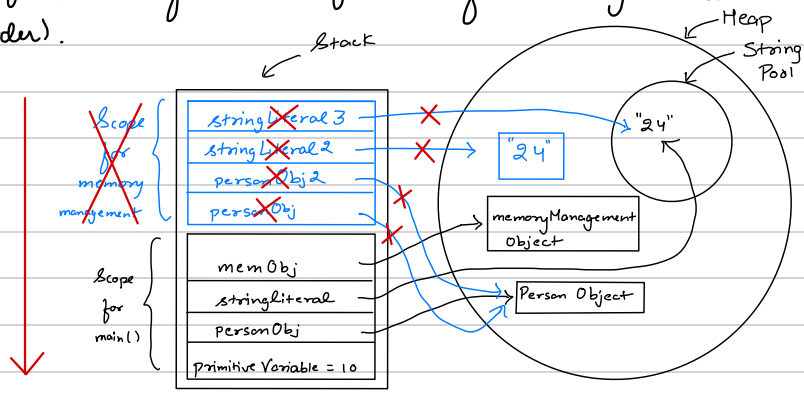
```java
public class MemoryManagement {

    public static void main(String args[]) {

        int primitiveVariable1 = 10; // primitive data type
        Person personObj = new Person(); // object
        String stringLiteral = "24"; // string literal
        MemoryManagement memObj = new MemoryManagement();
        memObj.memoryManagementTest(personObj);

    }

    private void memoryManagementTest(Person personObj){

        Person personObj2 = personObj; // reference to object
        String stringLiteral2 = "24"; // string literal
        String stringLiteral3 = new String( original: "24");
    }
}
```

↳ String Object

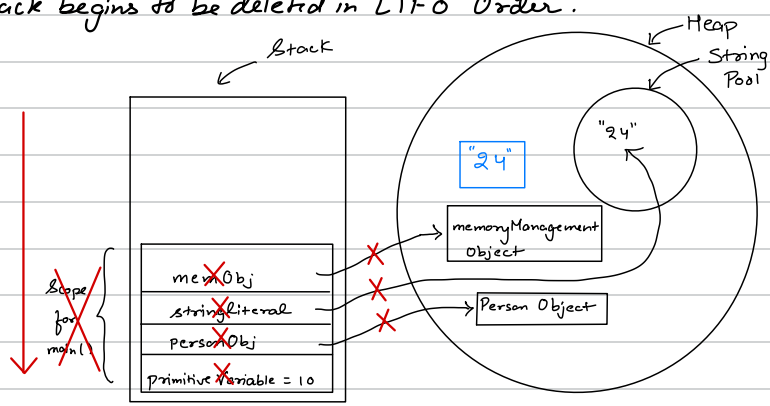
So, we have a simple class with various types of data is stored.
Now let's create memory for it.



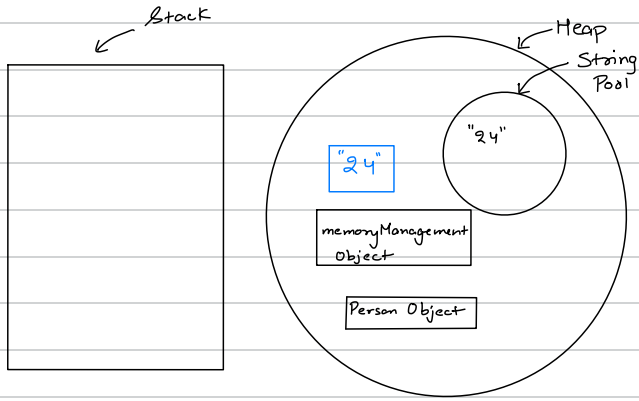
Now as soon as we encounter the closing bracket of memoryManagementTest method, its scope ends, it will delete its scope so all of the blue portion of stack gets deleted (in LIFO Order).



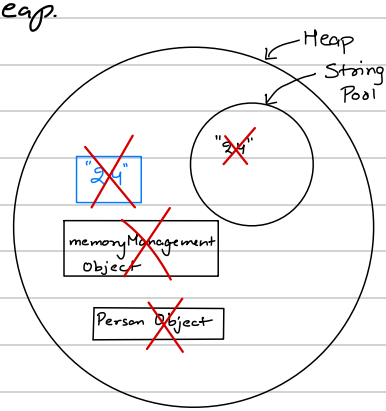
Now control comes back to main() method. Since nothing is there after calling memoryManagementTest api, we encounter the closing bracket which means the scope of main ends & it's portion in stack begins to be deleted in LIFO Order.



So now the stack is cleared & all the references are deleted from the stack as well. Now the memory looks like this



Now the stack is cleared & all the references are deleted but the objects are in the heap. So that's where garbage collector's work comes. Garbage Collector will delete all the unreferenced objects from the heap.



Garbage Collector runs periodically & JVM controls when to run the garbage collector. We can also tell the JVM to run the garbage collector using System.gc() but this doesn't guarantee that GC will run & that is why all of this is kla automatic memory management. The frequency of GC running is directly proportional to how much of the heap memory is currently full.

⇒ Type Of References :-

* Strong Reference :-

- It is when a variable in stack is referencing an object in Heap memory.
- Till the time the reference exists, GC won't be able to delete the object from the Heap Memory.

For eg:-

```
Person pobj = new Person();
```

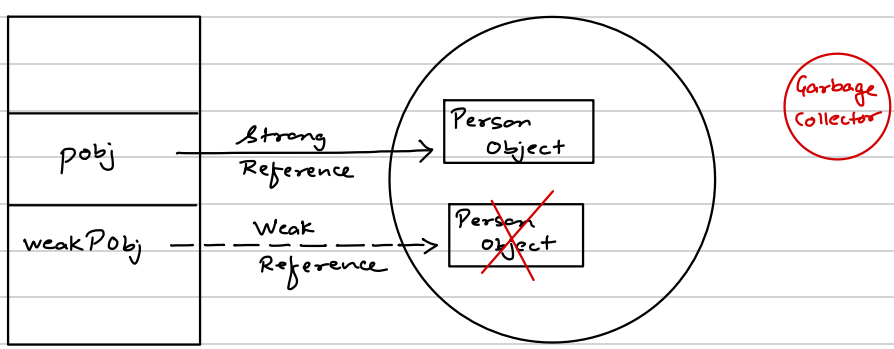
So here pobj has strong reference to a Person Object in the heap memory & till the time this reference exists, GC won't be able to delete Person object from Heap.

* Weak Reference :-

- In weak reference also the reference exists to an object in the heap but as soon as GC runs the object is deleted from heap memory even if some variable is referencing this object from the stack. The variable in the stack will get null if it tries to access the object post GC run.

For eg:-

```
WeakReference <Person> weakPobj = new WeakReference<Person>(new Person);
```



* Soft Reference :-

- It is a type of weak reference but the difference is that in this case the object will be deleted only when there is shortage of space in Heap. So GC is allowed to delete a soft reference but it'll keep the object if sufficient space is there in heap.

* Reference can be changed by referencing a current object to a new variable.

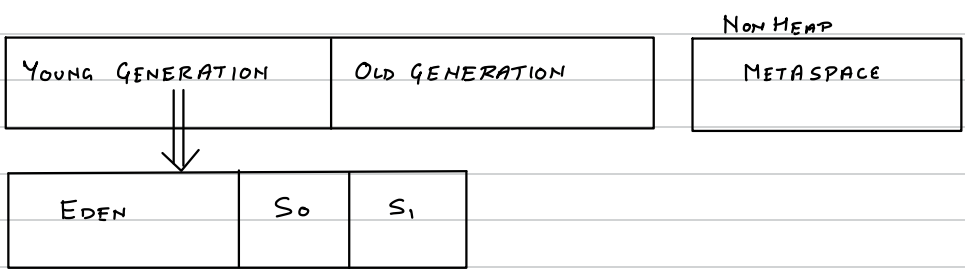
Eg:- Person obj1 = new Person (); // created new object
Person obj2 = new Person (); // created another object
obj1 = obj2;

Now obj1 will have a reference to the object of obj2 in heap & when GC runs, the earlier object which obj1 was refering to will be deleted.

⇒ Heap Memory

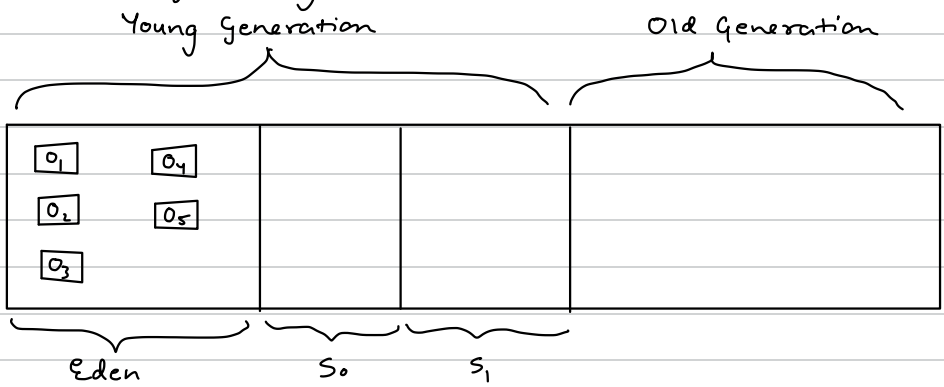
Heap memory is divided into two parts i.e Young Generation and Old Generation. There is also one more part which is generally known as Non Heap (Metaspace). Before Java 7 it is called permgen but now it is not used & it is used as metaspace.

Now young generation is further divided into 3 parts named Eden, S0 (Szero) and S_1. Szero & S_1 are known as survivor space



Now let's see when we create an object, what happens to it.

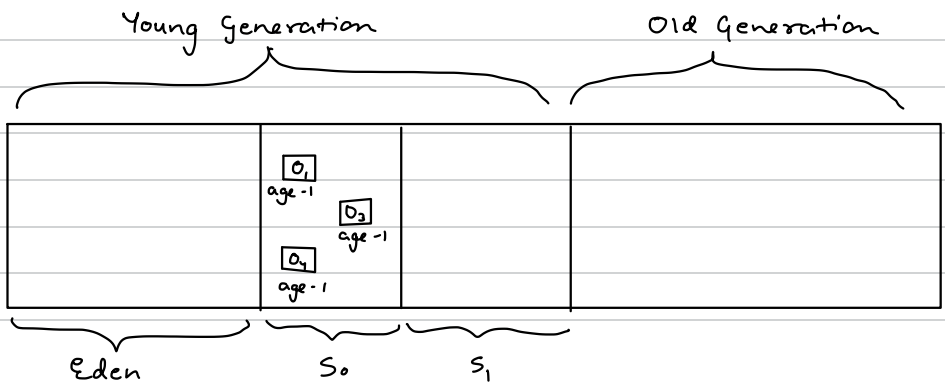
* Whenever a new object is created, it goes to Eden first. Let's say we've created 5 objects (O_1, O_2, O_3, O_4 & O_5). They'll be created in Eden first. So now heap memory looks like.



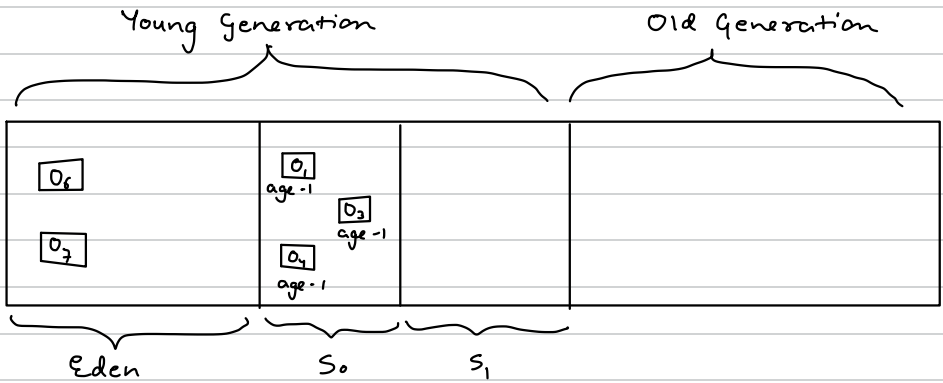
So 5 objects are created inside Eden.

Now let's say Garbage Collector run & there is no reference to O_2 & O_5 in the Heap Space. So now GC will use Mark & Sweep Algorithm i.e in Mark, GC will mark the objects which no more have reference & then sweep in which it'll do 2 things:

- First remove de referenced objects (O_2 & O_5) from the memory
- Move the rest of survivor objects into one of the survivor space ie S_0 or S_1 and adds age to the objects. So after GC runs, heap now looks like:



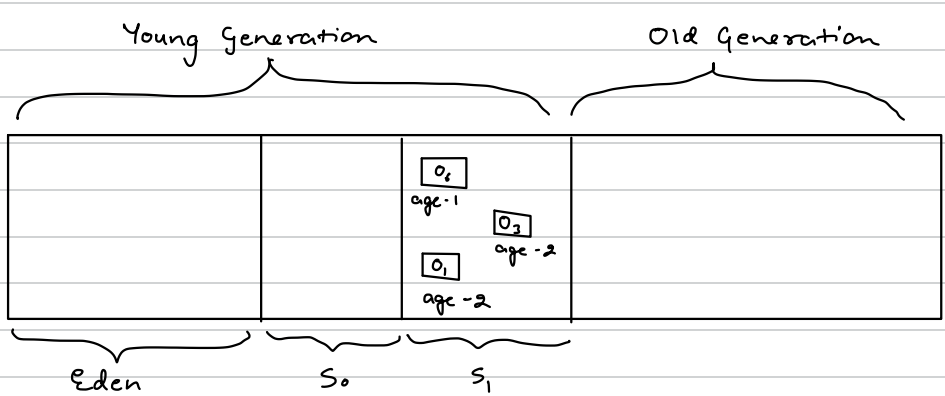
Now GC has ran once. This whole process is called minor GC as it happens very periodically & very fast.
Let's now create 2 more objects O_6 & O_7. So heap now looks like



O_6 & O_7 are now created in Eden. Now let's say the GC ran again & this time no reference is there for O_4 & O_7. So now GC will now:-

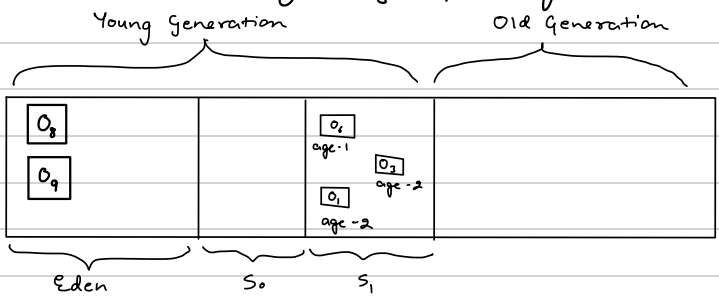
- Mark O_4 & O_7
- Deletes O_4 & O_7
- Moves O_1, O_6 & O_7 (survivors) to S_1 with corresponding ages.

Therefor post this minor GC, the heap looks like:



So at one time Eden would be completely free after the GC & one of the survivor space (S_0 or S_1) would be free and we put data alternatively in S_0 & S_1 along with the respective age.

Now let's create two more objects O_8 & O_9 & heap looks like:-



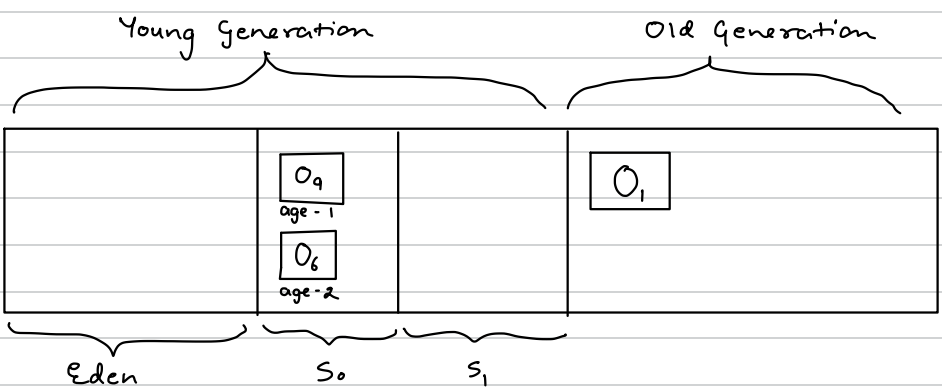
Now let's say I've set the threshold age to 3

Therefore now objects with age 3 needs to be promoted. Promotion means that now the objects with age 3 will be moved to Old Generation

Let's now run GC assuming that there are no more references to O_3 & O_8 hence will be deleted. So GC will now

- Mark O_3 & O_8
- Delete O_3 & O_8
- Move the survivor objects (O_6 & O_9) to S_0 & since O_1 still have reference with age 3, it'll be promoted i.e moved to old generation

Now the heap looks like:



In the old generation only difference is that here it is called major GC because the GC in Old Generation won't run too periodically. So in Old Generation the GC runs less periodically as compared to young gen and the objects in old gen are kind of big objects that are used too frequently & these might have a lot of references pointing to them.

Now let's come to metaspace. What will it store?
The metaspace contains :-

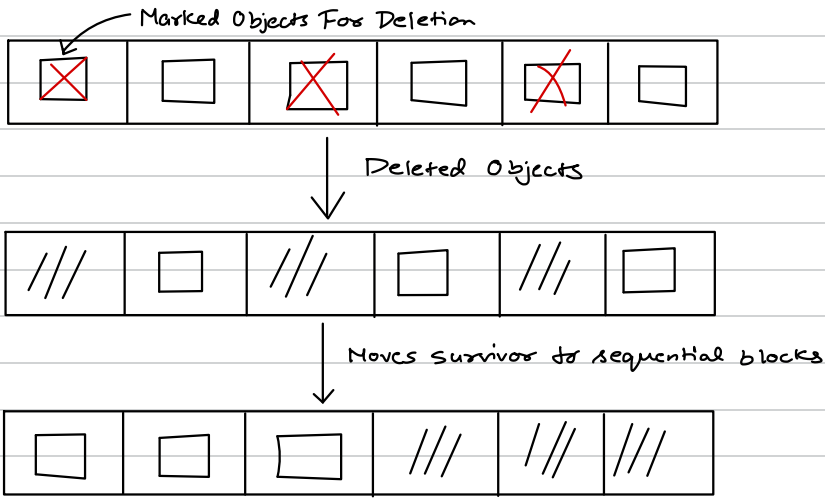
- Class variables
- Class metadata (basically stores info about class from which objects can be created)
- Constants

* Prior to Java 8, there is a fixed space kla perma gen which used to store the data that is stored in meta space & since it was non expandable, once it filled, we'll get out of memory error.
  But now from Java 8, metaspace is there which is different from heap & is expandable.

⇒ Garbage Collector Algorithms

* Mark & Sweep Algorithm (Already Discussed)
* Mark & Sweep with compact memory
  In this once the GC runs, the remaining objects are put in a sequential memory block leaving another sequential block free to put objects

Let's take this example



So this is kla mark & sweep with compaction

⇒ Versions Of GC

1) Serial GC
   - In this only one GC thread is working for both minor and major GC.
   - It's disadvantage is that due to the single thread, GC will be slower but since GC is expensive (because when GC work starts, all application threads pauses). So this is one of the disadvantages of serial.

2) Parallel GC
   - In this there are parallel threads running based on the CPU configuration. So GC works faster. So the application threads will be in paused state for lesser time.

3) Concurrent Mark & Sweep

- In this the GC will try its best to run concurrently with application threads but there is no guarantee. In this there is no memory compaction happens.

4) G_1 Garbage Collection

- This is a better version of concurrent mark & sweep in which GC will try not to stop / pause the application thread and supports memory compaction as well.

* Currently Java 8 is using parallel GC
* In latest Java Versions using CMS & G_1, there is a high chance that pause time is minimal thus increasing throughput & decreasing latency.

Classes In Java

* Type Of Classes

- Concrete Class
- Abstract Class
- Super Class and Sub Class
- Object Class
- Nested Class
  - Inner Class (Non Static Nested Class)
  - Anonymous Inner Class
  - Member Inner Class
  - Local Inner Class
  - Static Nested Class / Static Class
- Generic Class
- POJO Class
- Enum Class
- Final Class
- Singleton Class
- Immutable Class
- Wrapper Class

⇒ Concrete Class:

- These are those class that we can create an instance using NEW keyword.
- All the methods in this class have implementation.
- It can also be your child class from interface or extend abstract class.
- A class access modifier can be "public" or "package private" (no explicit modifier defined)

For example :-

```java
public class Person {

    int empID;

    Person(int empID){
        this.empID = empID;
    }

    public int getEmpID(){
        return empID;
    }
}
```

```java
public interface Shape {
    public void computeArea();
}
```

```java
public class Rectangle implements Shape{

    @Override
    public void computeArea() {
        System.out.println("Compute Rectangle Are
    }
}
```

⇒

Abstract Class (0 to 100% Abstraction):

Show only important features to users and hide its internal implementation.
2 ways to achieve abstraction:

- Class is declared as abstracted through keyword "abstract".
- It can have both abstract(method without body) and non abstract methods.
- We can not create an instance of this class.
- We parent has some features which all child classes have in common, then this can be used.
- Constructors can be created inside them. And with super keyword from child classes we can access them.

For Example :

```java
public abstract class Car {

    int mileage;

    Car(int mileage){
        this.mileage = mileage;
    }

    public abstract void pressBreak();

    public abstract void pressClutch();

    public int getNumberOfWheels(){
        return 4;
    }
}
```

→ Abstract Method

→ Abstract Method

→ Non-Abstract Method

```java
public abstract class LuxuryCar extends Car{

    LuxuryCar(int mileage){
        super(mileage);
    }

    public abstract void pressDualBreakSystem();

    @Override
    public void pressBreak(){
        //implementation of it goes here
    }
```

* Another abstract class inheriting above abstract class

→ Additional abstract method

```java
public class Audi extends LuxuryCar{

    Audi(int mileage){
        super(mileage);
    }

    @Override
    public void pressClutch() {
        //its implementation goes here
    }

    @Override
    public void pressDualBreakSystem() {
        //its implementation goes here
    }
}
```

* Concrete class inheriting abstract class (Luxury Car)

Both abstract methods are implemented

⇒ Super and Sub Class:

- A class that is derived from another class is called a Subclass.
- And from class through which Subclass is derived its called Superclass.
- In Java, in the absence of any other explicit superclass, every class is implicitly a subclass of **Object** class.
- Object is the topmost class in Java.

It has some common methods like clone(), toString(), equals(), notify(), wait() etc…

For Example:

```java
public class ObjectTest {

    public static void main(String[] args){

        ObjectTest obj = new ObjectTest();

        Object obj1 = new Person( empID: 1);
        Object obj2 = new Audi( mileage: 10);

        System.out.println(obj1.getClass());
        System.out.println(obj2.getClass());
    }
}
```

```
class Person
class Audi

Process finished with exit code 0
```

* Since Object is the parent class of every class in Java & we know that reference of child class can be kept in parent class, hence Audi & Person objects can be kept in reference of Object.

⇒ Nested Class:

Class within another class is called Nested Class.

When to use?
If you know that, a class(A) will be used by only one another class(B), then instead of created new file(A.java) for it, we can create nested class inside B class itself.
And
Also help us to group logically related classes in one file.

Scope:
Its Scope is same as of its Outer class.

It is of 2 types:
- Static Nested Class
- Non Static Nested Class
  - Member Inner Class
  - Local Inner Class
  - Anonymous Inner Class

Now let's go through types of nested classes one by one:

⇒ Static Nested Class:

- It do not have access to the non static instance variable and method of outer class.
- Its object can be initiated without initiating the object of outer class.
- It can be private, public, protected or package-private(default, no explicit declaration)

For Example:

```
class OuterClass {

    int instanceVariable = 10;
    static int classVariable = 20;

    static class NestedClass{

        public void print(){
            System.out.println(classVariable + instanceVariable);
        }
    }
}
```

Error because Static inner class can only access static variable / methods

```java
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass.NestedClass nestedObj = new OuterClass.NestedClass();
        nestedObj.print();
    }
}
```

Since static classes can be accessed directly with the name of the class. So we'll need the name of outerclass. Hence object of outer class was not needed.

* Nested Classes can be created with any type of access modifiers i.e. public, private, protected, default, package-protected

Let's take the example of a nested class with private access modifier

* Nested class object can be created within the same class itself.

```java
class OuterClass {

    int instanceVariable = 10;
    static int classVariable = 20;

    private static class NestedClass{

        public void print(){
            System.out.println(classVariable);
        }
    }

    public void display(){
        NestedClass nestedObj = new NestedClass();
        nestedObj.print();
    }
```

```java
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass outerClassObj = new OuterClass();
        outerClassObj.display();
    }
}
```

So, to access the private nested class, we had to create an object within the class itself & then expose

⇒

Inner Class or Non Static Nested Class:

- It have access to all the instance variable and method of outer class.
- Its object can be initiated on after initiating the object of outer class.

1. Member Inner Class:
   - its can be private, public, protected, default

For Example :

```java
class OuterClass {

    int instanceVariable = 10;
    static int classVariable = 20;

     class InnerClass{

        public void print(){
            System.out.println(classVariable + instanceVariable);
        }
    }
}
```

* To invoke it, we'll need an object of outer class

```java
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass outerClassObj = new OuterClass();

        OuterClass.InnerClass innerClassObj = outerClassObj.new InnerClass();
        innerClassObj.print();

    }
```

⇒

2. Local Inner Class:

- These are those classes which are defined in any block like for loop, while loop block, If condition block, method etc.
- It can not be declared as private, protected, public. Only default(not defined explicit) access modifier is used.
- It can not be initiated outside of this block.

For Example:

```java
class OuterClass {

    int instanceVariable = 1;
    static int classVariable = 2;


    public void display(){

        int methodLocalVariable = 3;

        class LocalInnerClass{
            int localInnerVariable = 4;

            public void print(){
                System.out.println(instanceVariable + classVariable + methodLocalVariable + localInnerVariable);
            }
        }

        LocalInnerClass localObj = new LocalInnerClass();
        localObj.print();

    }
}
```

So, it can be invoked inside the block only. As soon as the scope of block ends, it's scope also ends.

* Inheritence in Nested Classes

- Example - 1: One inner class can inherit another inner class in same outer class.

```java
class OuterClass {

    int instanceVariable = 1;
    static int classVariable = 2;


    class InnerClass1{

        int innerClass1 = 3;
    }

    class InnerClass2 extends InnerClass1{

        int innerClass2 = 4;

        void display(){
            System.out.println(innerClass1 + innerClass2 + instanceVariable + classVariable);
        }
    }

}
```

```java
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass outerClassObj = new OuterClass();
        OuterClass.InnerClass2 innerClass2Obj = outerClassObj.new InnerClass2();
        innerClass2Obj.display();

    }
}
```

Example - 2: Static inner class inherited by different class

```java
class OuterClass {
                    Nested Class
    static class InnerClass {
        public void display(){        Static Nested Class
            System.out.println("inside InnerClass");
        }
    }
}
```

```java
                                                Nested Class
public class SomeOtherClass extends OuterClass.InnerClass {

    public void display1(){
        display();
    }
}
```

Example - 3: Non Static Inner Class inherited by different class

```java
class OuterClass {

    class InnerClass {
        public void display(){
            System.out.println("inside InnerClass");
        }
    }
}
```

```java
public class SomeOtherClass extends OuterClass.InnerClass {

    SomeOtherClass(){
        new OuterClass().super();
        //as you know, when child class constructor invoked, it first invoked the constructor of parent.
        //but here the parent is Inner class, so it can only be accessed by the object of outerclass only.
    }

    public void display1(){
        display();
    }
}
```

⇒ Anonymous Inner Class

Anonymous Inner Class:

An inner class without a name called Anonymous class.

Why its used:
- When we want to override the behaviour of the method without even creating any subclass.

For Example:

```java
public abstract class Car {

    public abstract void pressBreak();
}
```

```java
public class Test {

    public static void main(String args[]) {

        Car audiCarObj = new Car() {

            @Override
            public void pressBreak() {
               //my audi specific implementation here
                System.out.println("Audi specific break changes");
            }
        };

        audiCarObj.pressBreak();
    }
}
```

Now we know that we cannot create an object of abstract class but here we had done. Let's understand what happened:

So 2 things happened behind the scene:

- Sub class is created, name decided by the compiler
- Creates an object of sub class and assigns its reference to object.

Similarly for interface also, it works in the same way:

va × Test$1.class × Test.class ×

iled .class file, bytecode version: 52.0 (Java 8)

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

final class Test$1 extends Car {
    Test$1() {
    }

    public void pressBreak() {
        System.out.println("Audi specific break changes");
    }
}
```

Generic Classes

* Generic class helps us to write a class in a generic manner that helps to avoid the typecasting that we'll have to use with Object class

Let's see via an example:

```
public class Print {

    Object value;

    public Object getPrintValue(){
        return value;
    }

    public void setPrintValue(Object value){
        this.value = value;
    }
}
```

Since Object is parent of every class, we used it here. Now here value can be of any type i.e String, integer etc. The only issue is that we'll have to typecast it as per our use.

```
public class Main {

    public static void main(String args[]) {

        Print printObj1 = new Print();
        printObj1.setPrintValue(1);
        Object printValue = printObj1.getPrintValue();
        //we can not use printValue directly, we have to typecast it, else it will be compile time er
        if((int)printValue == 1){
            //do something
        }
    }
}
```

So we had to typecast printValue to int to compare it with 1.

* How to define Generic Classes ?

→ We can do it using <T>, this T can be any alphabet like A, B, C etc.

So let's make our previous Print Class generic

```
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value = value;
    }
}
```

Generic Type (In above example <T>) can be any non-primitive Object

```
public class Main {

    public static void main(String args[]) {

        Print<Integer> printObj1 = new Print<Integer>();
        printObj1.setPrintValue(1);
        Integer printValue = printObj1.getPrintValue();
        if(printValue == 1){
            //do something
        }
    }
}
```

Therefore, while creating printObject1 we replaced T by Integer making it of an Integer type.

⇒ Inheritence In Generic Classes

1) Non - Generic Subclass

```java
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value = value;
    }
}
```

↓

```java
public class ColorPrint extends Print<String>{

}
```

So, here above we have extended / inherited a Generic Class to a non-generic subclass.

While extending / inheriting a generic class to a non generic subclass, we have to define the type of T at the time of extending itself.

```java
public class Main {

    public static void main(String args[]) {

        ColorPrint colorPrintObj = new ColorPrint();
        colorPrintObj.setPrintValue("2");
    }
}
```

2.) Generic Subclass

```java
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value = value;
    }
}
```

```java
public class ColorPrint<T> extends Print<T>{

}
```

```java
public class Main {

    public static void main(String args[]) {

        ColorPrint<String> colorPrintObj = new ColorPrint<>();
        colorPrintObj.setPrintValue("2");
    }
}
```

So, for a non-generic subclass, we can specify the type of T at the time of object creation

* More than one Generics Type Example:

- We can create as many number of generic parameters as we want i.e 1, 2, 3, 4 -- N (Classname $< T_1, T_2, T_3, T_4 ---- T_N >$

For Example:

```java
public class Pair<K,V> {

    private K key;
    private V value;

    public void put(K key, V value){
        this.key = key;
        this.value = value;
    }
}
```

```java
public class Main {

    public static void main(String args[]) {

        Pair<String, Integer> pairObj = new Pair<>();
        pairObj.put("hello", 1243);

    }
}
```

So, here we've used two generic parameters.

* Also both the syntaxes are right:

Pair < String, Integer > object = new Pair < String, Integer > ();

&

Pair < String, Integer > object = new Pair < > ();

* **Generic Method:**

What if we only wants to make Method Generic, not the complete Class, we can write Generic Methods too.

- Type Parameter should be before the return type of the method declaration.
- Type Parameter scope is limited to method only.

To make a method generic, put the generic type that we want it to accept before the return Type. For Example:-

```
public class GenericMethod {

    public <K,V> void printValue(Pair<K,V> pair1, Pair<K,V> pair2 ){

        if(pair2.getKey().equals(pair2.getKey())){
            //do something
        }
    }
}
```

* Generic Method with single generic parameter

```
public class Print{


    public <T> void setValue(T busObject){

        //do something
    }

}
```

```
public class Main {

    public static void main(String args[]) {

        Print printObj = new Print();
        printObj.setValue(new Bus());

    }
}
```

* Raw Type Object

It's a name of the generic class or interface without any type argument.

For Example :

```java
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value=value;
    }
}
```

```java
public class Main {

    public static void main(String args[]) {

        Print<String> parametrizedTypePrintObject = new Print<>();

        //internally it passes Object as parametrized type.
        Print rawTypePrintObject = new Print();
         rawTypePrintObject.setPrintValue(1);
         rawTypePrintObject.setPrintValue("hello");
    }
}
```

So while creating the object, we didn't pass any type as parameterized type. Therefore, internally it passes Object as parameterized type.

So rawTypePrintObject is a raw type object.

* Bounded Generics

Bounded Generics:
It can be used at generic class and method

- Upper Bound (<T extends Number>) means T can be of type Number or its Subclass only. Here superclass (in this example Number) we can have interface too.
- Multi Bound .

It is used to restrict what all objects we can pass at this type parameter.

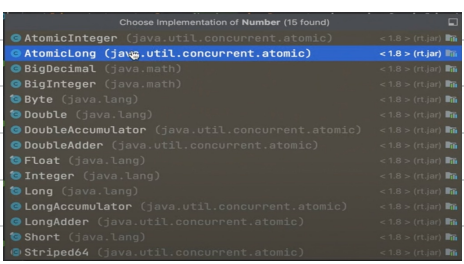
For Example:

- Upper Bound

Can also be interface

```
public class Print<T extends Number> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value=value;
    }
}
```

So now only Number or its child/subclasses can be used as objects which in our case are:




```
public class Main {

    public static void main(String args[]) {

        Print<Integer> parametrizedTypePrintObject = new Print<Integer>();
    }
}
```

⇒ Correct since Integer is a child class of number

```
public class Main {

    public static void main(String args[]) {

        Print<String> parametrizedTypePrintObject = new Print<>();
    }
}
```

⇒ Incorrect as String is not a child class of Number.

*

Multi Bound:
<T extends Superclass & interface1 & interface N>

- **The first restrictive type should be concrete class**
- **2,3 and so on... can be interfaces.**

For Example:

```
public class A extends ParentClass implements Interface1, Interface2{
}
```

So let's say I want the exact above as my type parameter. It'll be:

```
public class Print<T extends ParentClass & Interface1 & Interface2> {

    T value;

    public T getPrintValue() { return value; }

    public void setPrintValue(T value) { this.value = value; }
}
```

So, the below code will work fine

```
public class Main {

    public static void main(String args[]) {

        A obj = new A();
        Print<A> printObj = new Print<>();

    }
}
```

*

Wildcards:

- **Upper bounded wildcard : <? extends UpperBoundClassName> i.e. class Name and below**
- **Lower bounded wildcard : <? Super LowerBoundClassName> i.e. class Name and above**
- **Unbounded wildcard <?> only you can read**

For Example:

Let's say I have a Vehicle class & Bus, Car are extending it

```
public class Main {

    public static void main(String args[]) {

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new Bus());
        vehicleList.add(new Car());

        List<Bus> busList = new ArrayList<>();

        vehicleList = busList; //NO

          busList = vehicleList; //no

        Vehicle vehicleOb = new Vehicle();
        Bus busObj = new Bus();

        vehicleOb = busObj; // ok

    }
}
```

Here both above assignments are invalid because lists are different from objects

- busList = vehicleList is invalid because in busList we can store only the objects of type Bus.

- vehicleList = busList is invalid because in vehicleList we can store both bus & car objects while busList can contain only bus objects. Hence, it is invalid.

So, here wildcards can help us

```
public class Print{

    public void setPrintValues(List<? extends Vehicle> vehicleList){

    }
}
```

So we've used upperbound wild card. Therefore we can pass Vehicle & its child class

```java
public class Main {

    public static void main(String args[]) {

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new Bus());
        vehicleList.add(new Car());

        List<Bus> busList = new ArrayList<>();


        Print printObj = new Print();
        printObj.setPrintValues(busList);

    }
}
```

```java
public class Main {

    public static void main(String args[]) {

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new Bus());
        vehicleList.add(new Car());

        List<Bus> busList = new ArrayList<>();


        Print printObj = new Print();
        printObj.setPrintValues(vehicleList);

    }
}
```

So now we can pass vehicleList & busList since vehicleList & its child classes are allowed.

* Now let's see lower bound wildcards

```java
public class Print{
    public void setPrintValues(List<? super Vehicle> vehicleList){
    }
}
```

Now we can pass vehicle & its superclass type. Since vehicle is inheriting anything so Object can be used along with vehicle.

```java
List<Object> objList = new ArrayList<>();
printObj.setPrintValues(objList);
```

Now a question arises that can't we use generic type parameter in place of wildcards to achieve the functionality.

Therefore let's see the difference b/w wildcard & generic type

Let's create one wildcard method & one generic type method

```java
public class Print {


    //wild card method
    public void computeList(List<? extends Number> source, List<? extends Number> destination){



    }


    //generic type method
    public <T extends Number> void computeList1(List<T> source, List<T> destination){

    }

}
```

The main difference is that wildcards are a bit less restrictive as compared to generic type. For example, in the above, we can give 2 different type of list as source & destination for the wildcard but for generic method we have to give same type of list to source & destination.

In wildcard we have boundation as well i.e we can use super but in generic method we can't do so.

```java
public class Main {

    public static void main(String args[]) {


        List<Integer> wildCardIntegerSourceList = new ArrayList<>();
        List<Float> wildCardIntegerDesitanationList = new ArrayList<>();

        Print printObj = new Print();
        printObj.computeList(wildCardIntegerSourceList, wildCardIntegerDesitanationList);

        printObj.computeList1(wildCardIntegerSourceList, wildCardIntegerDesitanationList);


    }
}
```

So since compute1 is a generic method & it'll need same types as parameters Therefore showing compilation error in case of different types.

* Unbounded Wildcard
  - Mostly used when my method can work on the methods provided by the object class

* Type Erasure:
  Whatever the code we write, when the bytecode is generated, it's all replaced by actual values.

- Generic Class Erasure:

```java
public class Print <T> {

    T value;

    public  void setValue(T val) {
        this.value = val;
    }
}
```

→

```java
public class Print {

    Object value;

    public  void setValue(Object val) {
        this.value = val;
    }
}
```

- Generic Class Bound Type Erasure:

```java
public class Print<T extends Number> {

    T value;

    public  void setValue(T val) {
        this.value = val;
    }
}
```

→

```java
public class Print{

    Number value;

    public  void setValue(Number val) {
        this.value = val;
    }
}
```

- Generic Method Erasure:

```java
public class Print{

    public <T> void setValue(T val) {
        System.out.println("do something");
    }
}
```

```java
public class Print{

    public void setValue(Object val) {
        System.out.println("do something");
    }
}
```

- Generic Bound Type Method Erasure:

```java
public class Print{

    public <T extends Bus> void setValue(T val) {
        System.out.println("do something");
    }
}
```

```java
public class Print{

    public void setValue(Bus val) {
        System.out.println("do something");
    }
}
```

POJO Class

POJO Class:

- Stands for "Plain Old Java Object".
- Contains variables and its getter and setter methods.
- Class should be public.
- Public default constructor.
- No annotations should be used like @Table, @Entity, @Id etc..
- It should not extend any class or implement any interface.

For Example:

```java
public class Student {
    2 usages
    int name;
    2 usages
    private int rollNumber;
    2 usages
    protected String address;

    no usages
    public int getName() {
        return name;
    }

    no usages
    public void setName(int name) {
        this.name = name;
    }

    no usages
    public int getRollNumber() {
        return rollNumber;
    }

    no usages
    public void setRollNumber(int rollNumber) {
        this.rollNumber = rollNumber;
    }

    no usages
    public String getAddress() {
        return address;
    }

    no usages
    public void setAddress(String address) {
        this.address = address;
    }
}
```

Public class with default public constructor

Variables

Getter & Setter Methods

* Whenever any request comes to our system i.e component recieve the data, it is advisable that we map the request object to a POJO which all other classes understand and use. If anything changes in future, we only have to make change to this POJO.

Enum Class

ENUM Class:

- It has a collection of CONSTANTS (variables which values can not be changed)
- Its CONSTANTS are static and final implicitly (we do not have to write it).
- It can not extend any class, as it internally extends java.lang.Enum class
- It can implement interfaces.
- It can have variables, constructor, methods.
- It can not be initiated (as its constructor will be private only, even you give default, in bytecode it make it private)
- No other class can extend Enum class
- It can have abstract method, and all the constant should implement that abstract method.

⇒ Normal Enum Class

```java
public enum EnumSample{

    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;
}
```

* Internally for any constant we define in enum, a value is assigned starting from 0. So MONDAY will have 0, TUESDAY 2 and so on. It will happen only if we don't define custom values.

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

        /*Common functions which is used
        - values()
        - Ordinal()
        - valueOf()
        - name()
         */

        //1. usage of Values() and ordinal()
        for(EnumSample sample : EnumSample.values()){
            System.out.println(sample.ordinal());
        }


        //2. usage of ValueOf() and name()
        EnumSample enumVariable = EnumSample.valueOf( name: "FRIDAY");
        System.out.println(enumVariable.name());
    }
}
```

```
0
1
2
3
4
5
6
FRIDAY
```

* EnumSample.values() will return an array of all the constants present in the Enum. Each value that is associated/assigned to the variable is known as ordinal

* EnumValue.valueOf("FRIDAY") will iterate over all constants in the Enum & will return the constant which will match exactly with the given String (which in our case was "FRIDAY").

* .name() will return the name of the constant

⇒ Enum With Custom Values

```java
public enum EnumWithCustomValues {
    MONDAY( value: 101,  comment: "1st Day Of the Week"),
    TUESDAY( value: 102, comment: "2nd Day Of the Week"),
    WEDNESDAY( value: 103,  comment: "3rd Day Of the Week"),
    THURSDAY( value: 104,  comment: "4th Day Of the Week"),
    FRIDAY( value: 105,  comment: "5th Day Of the Week"),
    SATURDAY( value: 106,  comment: "6th Day Of the Week"),
    SUNDAY( value: 107,  comment: "7th Day Of the Week");

    private int value;
    private String comment;

    EnumWithCustomValues(int value, String comment) {
        this.value = value;
        this.comment = comment;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public String getComment() {
        return comment;
    }

    public void setComment(String comment) {
        this.comment = comment;
    }

    public static EnumWithCustomValues getEnumFromValue(int value){
        for(EnumWithCustomValues enumWithCustomValue : EnumWithCustomValues.values()){
            if(enumWithCustomValue.value == value){
                return enumWithCustomValue;
            }
        }
        return null;
    }
}
```

* So, each of the variable we define in enum class will be applicable for every constant in the class. In a way each constant is an object of enum class having defined variables

* We also need to define parameterised constructor which will be invoked for every constant

* To define a method for the whole class (enum) we need to make it static, otherwise it'll be applicable for all the constants.

```java
public class EnumDemo {
    public static void main(String[] args) {
        EnumWithCustomValues enumWithCustomValue = EnumWithCustomValues.getEnumFromValue(107);
        EnumWithCustomValues enumWithCustomValue1 = EnumWithCustomValues.TUESDAY;
        System.out.println(enumWithCustomValue.getComment());
        System.out.println(enumWithCustomValue1.getComment());
        System.out.println(EnumWithCustomValues.MONDAY.getValue());
        System.out.println(EnumWithCustomValues.MONDAY.getComment());
    }
}
```

```
7th Day Of the Week
2nd Day Of the Week
101
1st Day Of the Week
```

⇒ Method Override by Constant :

```java
public enum EnumMethodOverrideByConstant {
    MONDAY{
        @Override
        public void dummyMethod(){
            System.out.println("Monday Dummy Method");
        }
    },
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    public void dummyMethod(){
        System.out.println("Default Dummy Method");
    }
}
```

* Since dummyMethod is not static, it is applicable for all constants.

* We can override the method for a particular constant.

```java
public class EnumDemo {
    public static void main(String[] args) {
        EnumMethodOverrideByConstant enumMethodOverrideByConstant = EnumMethodOverrideByConstant.MONDAY;
        enumMethodOverrideByConstant.dummyMethod();
        EnumMethodOverrideByConstant enumMethodOverrideByConstant1 = EnumMethodOverrideByConstant.TUESDAY;
        enumMethodOverrideByConstant1.dummyMethod();
    }
}
```

```
Monday Dummy Method
Default Dummy Method
```

Since we've overriden dummy method for MONDAY, we got different output.

⇒ Enum With Abstract Method

```java
public enum EnumWithAbstractMethod {

    MONDAY{
        public void dummyMethod(){
            System.out.println("In Monday Dummy Method");
        }
    },
    TUESDAY{
        public void dummyMethod(){
            System.out.println("In Tuesday Dummy Method");
        }
    },
    WEDNESDAY{
        public void dummyMethod(){
            System.out.println("In Wednesday Dummy Method");
        }
    };
    public abstract void dummyMethod();
}
```

```java
public class EnumDemo {
    public static void main(String[] args) {
        EnumWithAbstractMethod enumWithAbstractMethod = EnumWithAbstractMethod.MONDAY;
        enumWithAbstractMethod.dummyMethod();
        EnumWithAbstractMethod enumWithAbstractMethod1 = EnumWithAbstractMethod.TUESDAY;
        enumWithAbstractMethod1.dummyMethod();
    }
}
```

```
In Monday Dummy Method
In Tuesday Dummy Method
```

So enum can have an abstract method which in turn will be implemented by all the constants.

⇒ Enum Implements Interface

- Enum can also implements interface

```java
public interface MyInterface {
    public String toLowerCase();
}
```

```java
public enum EnumImplementInterface implements MyInterface{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    @Override
    public String toLowerCase(){
        return this.name().toLowerCase();
    }
}
```

So here enum implements an interface & we can call toLowerCase() for every constant.

```java
public class EnumDemo {
    public static void main(String[] args) {
        EnumImplementInterface enumImplementInterface = EnumImplementInterface.MONDAY;
        System.out.println(enumImplementInterface.toLowerCase());
        EnumImplementInterface enumImplementInterface1 = EnumImplementInterface.TUESDAY;
        System.out.println(enumImplementInterface1.toLowerCase());
    }
}
```

```
monday
tuesday
```

* So what is the benefit of Enum Class when we can create constant through "static" and "final"?
- Let's understand this with an example

```java
public class WeekConstants {

    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;

}
```

```java
public enum EnumSample{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;
}
```

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

        isWeekend( day: 2); //wednesday, so it will return false
        isWeekend( day: 6); //sunday, so it will return true
        isWeekend( day: 100); //this value is not expected, but still we are able to send this in parameter


        //better readability and full control on what value we can pass in parameter
        isWeekend(EnumSample.WEDNESDAY); // return false
        isWeekend(EnumSample.SUNDAY); // return true


    }

    public static boolean isWeekend(int day){

        if(WeekConstants.SATURDAY == day || WeekConstants.SUNDAY == day){
            return true;
        }
        return false;
    }

    public static boolean isWeekend(EnumSample day){

        if(EnumSample.SATURDAY == day || EnumSample.SUNDAY == day){
            return true;
        }
        return false;
    }
}
```

For Constant Class

For Enum

So as seen in the above example, enum has following advantages:-
- It has better readability
- We have control on what value we can pass in parameter.

⇒ Final Class

- It is a class which cannot be inherited.

For Example:

```
public final class TestClass {
}
```

```
public class MyOtherClass extends TestClass{
}
```

Cannot inherit from final 'EnumDemo.TestClass'

Make 'TestClass' not final ⌥⇧⏎ More actions... ⌥⏎

Singleton Class

* The objective of this class is to create only single object.
  For example: If we want to create a DB Connection, we want it to be singleton.

* Different ways of creating Singleton Class:
  - Eager Initialization
  - Lazy Initialization
  - Synchronization Block
  - Double Check Lock (there is a memory issue, resolved through Volatile instance variable)
  - Bill Pugh Solution
  - Enum Singleton

⇒ Eager Initialization

```
public class DBConnection {

    private static DBConnection conObject = new DBConnection();

    private DBConnection(){
    }

    public static DBConnection getInstance(){
        return conObject;
    }
}
```

So we have:

1) Created & initialized the object itself as private static so that it is associated with class and cannot be accessible outside the class.
2) Created private constructor so that nobody is allowed to create the object of the class using "new" Keyword.

3) Created a public static method so that if other classes needed the object of this class, they can access it via this method which return this same object only.

```
public class Main {

    public static void main(String args[]) {

        DBConnection connObject = DBConnection.getInstance();
    }

}
```

So, this is eager initialization as we're creating & initialising the object as soon as the program starts even if it isn't being currently used by others.

⇒ Lazy Initialization

- It is created to solve the problem of eager initialization where we're creating & initializing the object as soon as program starts.

```
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

    public static DBConnection getInstance(){

        if(conObject == null){
            conObject = new DBConnection();
        }
        return conObject;
    }
}
```

So, in this the object will be created only when someone calls the provided method for the first time. After that the same object will be

returned if someone needs the object & calls getInstance method.

* It's disadvantage is that if two threads tries to get the object at the same time, the object is null & hence two objects will be created.

⇒ Synchronized Method:

- It is used to overcome the problem of Lazy Initialization where two objects might created if two threads tries to get the object for the first time.

```
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

    synchronized public static DBConnection getInstance(){

        if(conObject == null){
            conObject = new DBConnection();
        }
        return conObject;
    }
}
```

So, this synchronized keyword does two things:

- Put a lock on the method
- unlock the method.

So, only one thread is allowed to enter the method at a time. Hence the possibility of 2 objects being created will be zero.

* The only disadvantage of using synchronized is that it is very very slow and generally not used.

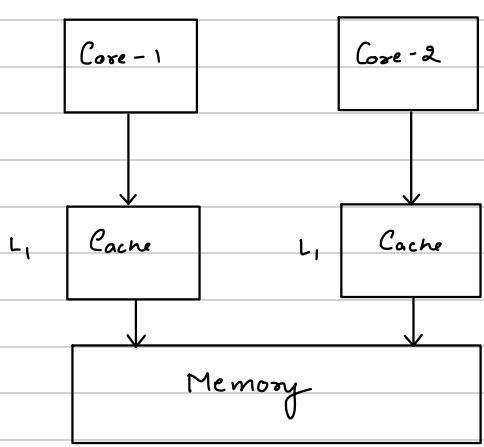
⇒ Double Checked Locking (This is used majorly)

- It overcomes the problem in synchronized method i.e slowness.

```java
public class DatabaseConnection {

    private static volatile DatabaseConnection conObject;

    private DatabaseConnection() {
    }

    public static DatabaseConnection getInstance() {
        if(conObject == null){
            synchronized (DatabaseConnection.class){
                if(conObject == null) {
                    conObject = new DatabaseConnection();
                }
            }
        }
        return conObject;
    }
}
```

So, here the synchronized is not on method, but on the block. Because of synchronized on the block, the lock unlock once only.

It's disadvantage is that there is a memory issue in this. Let's understand it.



Each core has its dedicated L_1 Cache which is used to cache the objects & time to time it syncs with memory.

Now let's say Thread T_1's computation is happening at core-1 and it enters the method to get object. Since it'll get null for the first time an object will be created & temporarily stored in cache. At this point Thread T_2 whose computation is happening at Core-2 tries to get the object. Now we do have the object created but not yet synced with the memory. Hence a second object will be created because the object is not there in memory. Therefore two objects are created despite double checked locking.

This is solved using the volatile keyword. Volatile keyword means that the object will be created in memory instead of cache.
So, if we've created any object volatile, any read/write operation happening to this always happens in memory.
Since we're using memory & synchronized, this is also a bit slow.

⇒ Bill Pugh Solution:

```java
public class DatabaseConnection {

    private DatabaseConnection() {
    }

    private static class DBConnectionHelper {
        private static final DatabaseConnection INSTANCE_OBJECT = new DatabaseConnection();
    }

    public static DatabaseConnection getInstance() {
        return DBConnectionHelper.INSTANCE_OBJECT;
    }
}
```

So, Bill Pugh Solution rectified the issue of eager initialization by putting the object inside private static nested class because this nested class

do not get loaded on program startup. Instead it is loaded when it is referred.

⇒ ENUM SINGLETON

- As we know, in Enum, all constructors are private & only one object of enum is created per JVM. So by default enum are singleton only.

```
enum DBConnection {

    INSTANCE;
}
```

⇒ IMMUTABLE CLASSES

IMMUTABLE CLASS:

- We can not change the value of an object once it is created.
- Declare class as 'final' so that it can not be extended.
- All class members should be private. So that direct access can be avoided.
- And class members are initialized only once using constructor.
- There should not be any setter methods, which is generally use to change the value.
- Just getter methods. And returns Copy of the member variable.
- Example: String, Wrapper Classes etc.

```
final class MyImmutableClass {

    private final String name;
    private final List<Object> petNameList;

    MyImmutableClass(String name, List<Object> petNameList){
        this.name = name;
        this.petNameList = petNameList;
    }

    public String getName(){
        return name;
    }

    public List<Object> getPetNameList(){
        //this is required, because making list final,
        // means you can not now point it to new list, but still can add, delete values in it
        //so thats why we send the copy of it.

        return new ArrayList<>(petNameList);
    }
}
```

```java
public class Main {

    public static void main(String args[]){
        List<Object> petNames = new ArrayList<>();
        petNames.add("sj");
        petNames.add("pj");

        MyImmutableClass obj = new MyImmutableClass( name: "myName", petNames);
        obj.getPetNameList().add("hello");
        System.out.println(obj.getPetNameList());
    }
}
```

Output:

```
[sj, pj]
```

Since we're passing the copy of the list, new items won't be added to it hence making it truly final.

⇒ Wrapper Class

- We've already discussed it in Java Variables part.

Java: Interface in Depth

"Concept && Coding" YT Video Notes

- What is Interface and How to define
- Why we need interface
- Methods in interface
- Fields in an interface
- Interface Implementation
- Nested Interface
- Difference between Interface and Abstract class.

Java8 Interface features:

- Default method
- Static method

Java9 interface feature:

- Private method and
- Private Static method

Report Abuse

What is Interface?

Interface is something which helps 2 system to interact with each other, without one system has to know the details of other.

Or in simple term i can say, it helps to achieve ABSTRACTION.

How to define the Interface?

Interface declaration consist of

- Access specifier
- "Interface" keyword
- Interface Name
- Comma separated list of parent interfaces
- Body

Only **Public** and **Default** Modifiers are allowed. **Protected** and **private** are not allowed.

```
public interface Bird {

    no usages
    public void fly();
}
```

```
interface Bird {

    no usages
    public void fly();
}
```

Comma separated list of parent Interfaces (It can extend from **Class**) Example:

```
public interface NonFlyingBirds extends Bird, LivingThings{

    no usages
    public void canRun();
}
```

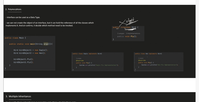
Why we need interface?

1. Abstraction:

Using interface, we can achieve full Abstraction means, we can define WHAT class must do, but not HOW it will do

```
public interface Bird {

    no usages
    public void fly();
}
```

```
public class Eagle implements Bird{
    no usages
    @Override
    public void fly() {
        //the abstract process of flying take place here
    }
}
```



```
public interface Bird {
    no usages
    int MAX_HEIGHT_IN_FEET = 2000;
}
```

```
public interface Bird {
    no usages
    public static final int MAX_HEIGHT_IN_FEET = 2000;
}
```

Interface implementation:

- Overriding method can not have more restrict access specifiers.
- Concrete class must override all the methods declared in the interface.
- Abstract classes are not forced to override all the methods.
- A class can implement from multiple interfaces.

```
public interface Bird {

    2 usages   2 implementations
    public void fly();
}
```

```
public class Eagle implements Bird{
    2 usages
    @Override
    public void fly() {
        System.out.println("Eagle Fly Implementation");
    }
}
```

```
public class Eagle implements Bird{
    1 usage
    @Override
    protected void fly() {

        //do something
    }
}
```

Example of Abstract class implementation of Interface:

```
public interface Bird {
    no usages   1 implementation
    public void canFly();
    no usages   1 implementation
    public void noOfLegs();
}
```

```
public abstract class Eagle implements Bird{
    no usages
    @Override
    public void canFly() {
        //Implementation goes here
    }

    no usages
    public abstract void beakLength();
}
```

```
public class WhiteEagle extends Eagle{

    no usages
    @Override
    public void noOfLegs() {
        //implement interface method
    }

    no usages
    @Override
    public void beakLength() {
        //implementing abstract class method
    }
}
```

Nested Interface:

- Nested Interface declared withing another Interface.
- Nested Interface declared within a Class.

Generally its is used to group, logical related interfaced. And Nested interface

Rules:

- A nested interface declared withing an interface must be public.
- Nested interface declared within a class can be public, protected, default.
- When you implement outer interface, inner interface implementation is not required and vice versa.

```
public interface Bird {
    no usages   1 implementation
    public void canFly();
    no usages
    public interface NonFlyingBird {
        no usages
        public void canRun();
    }
}
```

```
public class Bird {
    1 usage
    protected interface NonFlyingBird{
        no usages
        public void canRun();
    }
}

public class Eagle implements Bird.NonFlyingBird{

    no usages
    @Override
    public void canRun() {

    }
}
```

Interface Vs Abstract Class:

S.No.	Abstract Class	Interface
1.	Keyword used here is "abstract"	Keyword used here is "interface"
2.	Child Class should use keyword "extends" to inherit	Child class should use keyword "implements" to inherit
3.	It can have both abstract and non abstract method	It can have only Abstract method (from Java 8 onwards we can have default, static and private method also in interface, from Java 9 onwards)
4.	It can extend from Another class and multiple interfaces	It can only extends from other interfaces
5.	Variables can be final, non-static, final, static	Variable are by default CONSTANTS
6.	Variables and Methods can be private, protected, public, default	Variables and Methods are by default public (in Java8, private methods is allowed)
7.	Multiple inheritance is not supported	Multiple inheritance is supported
8.	It can provide the implementation of the interface	It can not provide implementation of any other interface or abstract class
9.	It can extend from Another class	It can extend from other interface
10.	To declare the method abstract, we have to use "abstract" keyword and it can be protected, public, default.	No need for any keyword to declare method abstract. And be default its public.

How to extend interface, that contains Default Method:

1st Way:

```java
public interface LivingThing {

    default boolean canBreathe(){
        return true;
    }
}
```

```java
public interface Bird extends LivingThing{

}
```

```java
public class Eagle implements Bird{

}
```

```java
public class Main {

    public static void main(String args[]){

        Eagle eagleObj = new Eagle();
        eagleObj.canBreathe();

    }
}
```

2nd Way:

```java
public interface LivingThing {

    default boolean canBreathe(){
        return true;
    }
}
```

```java
public interface Bird extends LivingThing{

    boolean canBreathe();
}
```

```java
public class Eagle implements Bird{

}
```

```java
public class Eagle implements Bird{

    @Override
    public boolean canBreathe() {
        return true;
    }
}
```

3rd Way:

```java
public interface LivingThing {

    default boolean canBreathe(){
        return true;
    }
}
```

```java
public interface Bird extends LivingThing {

    default boolean canBreathe() {
        boolean canBreatheOrNot = LivingThing.super.canBreathe();
        // do something else
        return canBreatheOrNot;
    }
}
```

```java
public class Eagle implements Bird{
}
```

2. Static Method (Java8):

- We can provide the implementation of the method in interface.
- But it can not be overridden by classes which implement the interface.
- We can access it using Interface name itself.
- Its by default public.

```java
public interface Bird {

    static boolean canBreathe() {
        return true;
    }
}
```

```java
public class Eagle implements Bird{

    public void digestiveSystemTestMethod() {

        if(Bird.canBreathe()){

            //do something
        }
    }
}
```

If you try to override it,
it will just be treated as new method in Eagle class

```java
public class Eagle implements Bird{

    public boolean canBreathe() {
        System.out.println("in interface");
        return true;
    }
}
```

If you try to add @override annotation,
it will throw compilation error

```java
public class Eagle implements Bird{

    @Override
    public boolean canBreathe() {
        System.out.println("in interface");
        return true;
    }
}
```

3. Private Method and Private Static method (Java9):

- We can provide the implementation of method but as a private access modifier in interface.
- It brings more readability of the code. For example if multiple default method share some code, that this can help.
- It can be defined as static and non-static.
- From Static method, we can call only private static interface method.
- Private static method, can be called from both static and non static method.
- Private interface method can not be abstract. Means we have to provide the definition.
- It can be used inside of the particular interface only.

```java
public interface Bird {

    void canFly();//this is equivalent to public abstract void canFly();
    void canFly();//this is equivalent to public abstract void canFly();

    public default void minimumFlyingHeight() {
        myStaticPublicMethod();  //calling static method
        myPrivateMethod(); //calling private method
        myPrivateStaticMethod();   //calling private static method
    }

    static void myStaticPublicMethod() {
        myPrivateStaticMethod(); //from static we can call other static method only
    }

    private void myPrivateMethod(){
        // private method implementation
    }

    private static void myPrivateStaticMethod(){
        // private static method implementation
    }
}
```

Java: Functional Interface and Lambda Expression

"Concept && Coding" YT Video Notes

- What is Functional Interface?
- What is Lambda Expression?
- How to use Functional Interface with Lambda expression
- Advantage of Functional Interface?
- Types of Functional Interface?
  - Consumer
  - Supplier
  - Function
  - Predicate
- How to handle use case when Functional Interface extends from other Interface(or Functional Interface)?

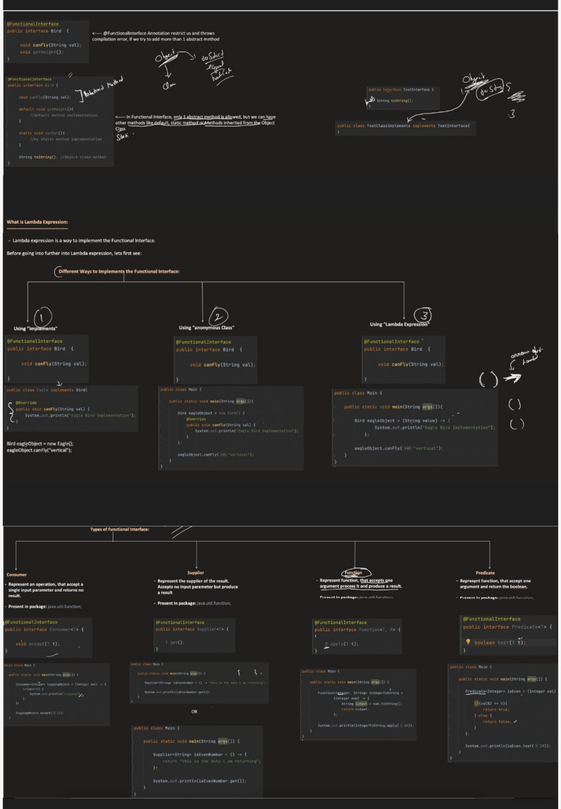
What is Functional Interface:

- If an interface contains only 1 abstract method, that is known as Functional Interface.
- Also know as SAM Interface (Single Abstract Method).
- @FunctionalInterface keyword can be used at top of the interface (But its optional).

```java
@FunctionalInterface
public interface Bird  {
      void canFly(String val);
}
```

OR

```java
public interface Bird  {
      void canFly(String val);
}
```



Handle use case when Functional Interface extends from other Interface:

Use Case 1: Functional Interface extending Non Functional Interface

```java
public interface LivingThing {

    public void canBreathe();
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

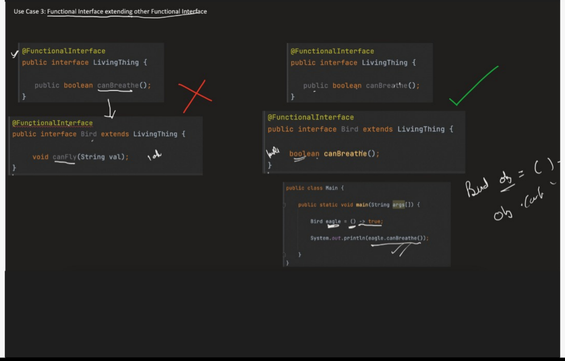
```java
public interface LivingThing {

    default public boolean canBreathe(){
        return true;
    }
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

Use Case 2: Interface extending Functional Interface

```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

```java
public interface Bird extends LivingThing {

    void canFly(String val);
}
```



Java: Reflection

"Concept && Coding" YT Video Notes

What is Reflection?

This is used to examine the Classes, Methods, Fields, Interfaces at runtime and also possible to change the behavior of the Class too.

For example:

- What all methods present in the class
- What all fields present in the class.
- What is the return type of the method.
- What is the Modifier of the Class
- What all interfaces class has implemented
- Change the value of the public and private fields of the Class etc.....

How to do Reflection of Classes?

To reflect the class, we first need to get an *Object* of **Class**
(So, first understand, Class then we will come back to how to reflect the class.)

What is this class Class?

- Instance of the class Class represents classes during runtime.
- JVM creates one Class object for each and every class which is loaded during run time.
- This Class object, has meta data information about the particular class like its method, field, constructor etc.

How to get the particular class Class object?

There are 3 ways:

1. Using **forName()** method

```
//assume that we have one class called Bird
class Bird { }

//get the object of Class for getting the metadata information of Bird class.
Class birdClass = Class.forName("Bird");
```

2. Using **class**

```
//assume that we have one class called Bird
class Bird { }

//get the object of Class for getting the metadata information of Bird class.
Class birdClass = Bird.class;
```

3. Using **getClass()** method

```
//assume that we have one class called Bird
class Bird { }

Bird birdObj = new Bird();

//get the object of Class for getting the metadata information of Bird class.
Class birdClass = birdObj.getClass();
```

Now lets get back to, **how to do Reflection of Classes:**

```
public class Eagle {
    public String breed;
    private boolean canSwim;

    public void fly() {
        System.out.println("fly");
    }

    private void eat() {
        System.out.println("eat");
    }
}
```

```
public class Main {
    public static void mainString String[] {
        Class eagleClass = Eagle.class;

        System.out.println(eagleClass.getName());
        System.out.println(Modifier.toString(eagleClass.getModifiers));
    }
}
```

OUTPUT:

```
Eagle
public
Process finished with exit code 0
```

Methods Available in Class object, all are get, not set methods

```
getClassLoader()                                        ClassLoader
getModifiers()Ljava.lang.Class;                         Class
getName()Ljava.lang.Class;                              Class
getSuperclass()                                         Class<? extends Object>
getBreed()                                              String
getDeclaredAnnotationsInternal()                        Map<Class,Annotation>
getDeclaredAnnotationMap()                              Map<Class,Annotation>
getDeclaredPublicMethods()                              Method[]
getDeclaredFields()                                     Field[]
getDeclaredPublicFields()                               Field[]
getDeclaredConstructors()                               Constructor<T>[]
getDeclaredConstructor(Class<?>... parameterTypes)      Constructor<T>
getDeclaredPublicConstructors()                         Constructor<T>[]
getDeclaredPublicConstructor(Class<?>... parameterTypes)  Constructor<T>
getDeclaredClasses()                                    Class<?>[]
getDeclaredPublicClasses()                              Class<?>[]
getDeclaredMethod(String name, Class<?>... parameterTypes)  Method
getDeclaredPublicMethod(String name, Class<?>... parameterTypes)  Method
getDeclaredField(String name)                           Field
getDeclaredPublicField(String name)                     Field
getDeclaringClass()                                     Class<?>
getEnclosingClass()                                     Class<?>
getEnclosingConstructor()                               Constructor<?>
getEnclosingMethod()                                    Method
getDeclaringClass0()                                    Class<?>
getPolicy()                                             ClassPolicy
getFactory()                                            CoreReflectionFactory
getGenericInfo()                                        ClassRepository
getEnclosingMethodInfo()                                EnclosingMethodInfo
getGenericScope()                                       ClassScope
getRawTypeParams()                                      TypeVariable<Class<T>>[]
getTypeParameters()                                     TypeVariable<Class<T>>[]
getSuperclass()                                         Class<? super T>
getGenericSuperclass()                                  Type
getPackageImpl()                                        Package
getProtectionDomain0()                                  ProtectionDomain
getPrimitiveClass(String name)                          Class
getResourceAsStream(String name)                        InputStream
getClassLoader()                                        ClassLoader
getModifiersRaw()                                       int
getInterfaces()                                         Class[]
getCachedConstructor()                                  Constructor
getCachedMethods()                                      Method[]
getCachedFields()                                       Field[]
getCachedPublicFields()                                 Field[]
getCachedPublicMethods()                                Method[]
getCachedConstructors()                                 Constructor<T>[]
getCachedPublicConstructors()                           Constructor<T>[]
getCachedDeclaredFields()                               Field[]
getCachedDeclaredPublicFields()                         Field[]
getCachedDeclaredMethods()                              Method[]
getCachedDeclaredPublicMethods()                        Method[]
getCachedDeclaredConstructors()                         Constructor<T>[]
getCachedDeclaredPublicConstructors()                   Constructor<T>[]
getCachedRecordComponents()                             RecordComponent[]
getSimpleBinaryName()                                   String
getSimpleBinaryName0()                                  String
getDeclaringClassBinaryName()                           String
getCanonicalName()                                      String
getCanonicalName0()                                     String
getSimpleName()                                         String
getTypeName()                                           String
getPackage()                                            Package
getProtectionDomain()                                   ProtectionDomain
getResource(String name)                                URL
getResourceAsStream(String name)                        InputStream
getClasses()                                            Class<?>[]
getDeclaredFields()                                     Field[]
getDeclaredMethods()                                    Method[]
getDeclaredConstructors()                               Constructor<?>[]
getDeclaredClasses()                                    Class<?>[]
getRecordComponents()                                   RecordComponent[]
isPrimitive()                                           boolean
isAnnotation()                                          boolean
isInterface()                                           boolean
isArray()                                               boolean
isRecord()                                              boolean
getModifiers()                                          int
isSealed()                                              boolean
isConstructor()                                         boolean
isEnum()                                                boolean
isMemberClass()                                         boolean
isLocalClass()                                          boolean
isAnonymousClass()                                      boolean
isCanonicalConstructor()                                boolean
getClassFileVersion()                                   int
getClassFileAccessFlags()                               int
hashCode()                                              int
toGenericString()                                       String
toString()                                              String
```

The package "java.lang.reflect" provides classes that can be used to access and manipulate the value like fields, methods, constructor etc.
And these classes are generally returned by above listed get Methods only.

Invoking Method using Reflection:

```
public class Eagle {
    Eagl() {

    }

    public void fly(int intParam, boolean boolParam, String strParm) {
        System.out.println("fly intParam: " + intParam + " boolParam: " + boolParam + " strParm: " + strParm);
    }
}
```

```
public class Main {
    public static void mainString String[] throws NoSuchMethodException, IllegalAccessException, InvocationTargetException {

        Class eagleClass = Class.forName("Eagle");
        Object eagleObject = eagleClass.newInstance();

        Method[] methods = eagleClass.getMethods();
        for (Method m : methods) {
            System.out.println("name: " + m.getName() + ", parameterTypes: " + Arrays.toStringm.getParameterTypes());
            if (m.getName().equals("fly")) {
                m.invoke(eagleObject, 1, true, "hello");
            }
        }
    }
}
```

Output:

```
fly intParam: 1 boolParam: true strParm: hello

Process finished with exit code 0
```

Reflection of Fields:

```
public class Eagle {
    public String breed;
    private boolean canSwim;

    public void fly() {
        System.out.println("fly");
    }

    private void eat() {
        System.out.println("eat");
    }
}
```

Output:

```
fieldName: breed
Type: class java.lang.String
Modifier: public
Value: null

fieldName: canSwim
Type: boolean
Modifier: private

Process finished with exit code 0
```

Get method supported

```
getClassLoader()                                        ClassLoader
getName()                                               String
getModifiers()                                          int
getSuperclass()                                         Class<?>
getInterfaces()[Class<?> annotationClass]               boolean
getAnnotationTypes()                                    AnnotationType
getAnnotationsByType(Class<?> annotationClass)          Annotation[]
getDeclaredAnnotationsByType(Class<?> annotationClass)  Annotation[]
getDeclaredAnnotationMap()                              Map<Class,Annotation>
getBoolean(Object obj)                                  boolean
getByte(Object obj)                                     byte
getChar(Object obj)                                     char
getDeclaredAnnotationsInternal()                        Annotation[]
getDeclaredAnnotation(Class<?> annotationClass)         Annotation
getDeclaringClass()                                     Class<?>
getDouble(Object obj)                                   double
getFloat(Object obj)                                    float
getInteger(Object obj)                                  int
getInt(Object obj)                                      int
getLong(Object obj)                                     long
getShort(Object obj)                                    short
getDeclaredAnnotationsByType(Class<?> annotationClass)  T[]
getClass()                                              Class<? extends Field>
```

Setting the value of private field:

```
public class Main {
    public static void mainString String[] throws NoSuchMethodException, IllegalAccessException {

        Class eagleClass = Eagle.class;

        Eagle eaglObj = new Eagle();
        // Get all public and private fields with this
        Field[] fields = eaglObj.getDeclaredFields();
        Field canSwimField = null;
        for (Field f : fields) {
            System.out.println("fieldName: " + f.getName());
            if (f.getName().equals("canSwim")) {
                canSwimField = f;
                canSwimField.setAccessible(true);
                boolean canSwimVal = (boolean) canSwimField.get(eaglObj);
                System.out.println("value is set to true");
                canSwimField.setBoolean(eaglObj, true);
                break;
            }
        }
    }
}
```

Output:

```
value is set to true

Process finished with exit code 0
```

Reflection of Constructor:

```
public class Eagle {
    private Eagle() {
        //System.out.println("fly");
    }

    public void fly() {
        System.out.println("fly");
    }
}
```

```
public class Main {
    public static void mainString String[] throws InvocationTargetException, InstantiationException, IllegalAccessException {

        Class eagleClass = Eagle.class;

        Constructor[] constructors = eagleClass.getDeclaredConstructors();
        for (Constructor c : constructors) {
            System.out.println("constructor: " + c.getName() + ", parameterTypes: " + Arrays.toStringc.getParameterTypes());
            if (c.getParameterTypes().length == 0) {
                c.setAccessible(true);
                Object eagleObject = (Eagle) eagleClass.newInstance();
                eagleObject.fly();
            }
        }
    }
}
```

```
getDeclaredConstructors()                               Constructor<?>[]
getDeclaredFields()                                     Field[]
getDeclaredMethods()                                    Method[]
getPermittedSubclasses()                                Class<?>[]
getNestMembers()                                        Class<?>[]
getClassValueMap()                                      ClassValueMap
getConstructor0(Class<?> parameterTypes, int which)     Constructor<T>
getProtectionDomain()                                   ProtectionDomain
getPrimitiveClass()                                     Class
getCachedClassInfo()                                    ClassInfo
getEnclosingMethod0()                                   Object[]
getDeclaringClass()                                     Class<?>
getSimpleName()                                         String
getTypeName()                                           String
getCanonicalName()                                      String
isLocalOrAnonymousClass()                               boolean
getResource(String name)                                URL
getClassLoader()                                        ClassLoader
getClasses()                                            Class<?>[]
getDeclaredFields()                                     Field[]
getDeclaredMethods()                                    Method[]
getDeclaredConstructors()                               Constructor<?>[]
getDeclaredClasses()                                    Class<?>[]
getPackage()                                            Package
getProtectionDomain()                                   ProtectionDomain
getResourceAsStream(String name)                        InputStream
getClassLoader()                                        ClassLoader
getModifiersRaw()                                       int
getInterfaces()                                         Class[]
getComponentType()                                      Class<?>
isPrimitive()                                           boolean
isAnnotation()                                          boolean
isInterface()                                           boolean
isArray()                                               boolean
isRecord()                                              boolean
getModifiers()                                          int
isSealed()                                              boolean
isConstructor()                                         boolean
isEnum()                                                boolean
isMemberClass()                                         boolean
isLocalClass()                                          boolean
isAnonymousClass()                                      boolean
isCanonicalConstructor()                                boolean
getClassFileVersion()                                   int
getClassFileAccessFlags()                               int
hashCode()                                              int
toGenericString()                                       String
toString()                                              String
```

Java: ANNOTATIONS

"Concept && Coding" YT Video Notes

What is Annotation?

- It is kind of adding **META-DATA** to the Java code.
- Always start having **@** symbol.
- We can use this meta data information at **runtime** and can add certain logic in our code if warrnot.
- How to read Meta data information? Using **Reflection** as discussed in previous video.
- Annotations can be applied at anywhere like Classes, Methods, Interface, fields, parameters etc.

Example:

```
public interface Bird {
    public boolean fly();
}
```

Annotation phrased simply:

```
public class Jay implements Bird {
    @Override
    public boolean fly() {
        return true;
    }
}
```

TYPES OF ANNOTATIONS:

- Pre-Defined Annotations
  - Used on Annotations (Called Meta Annotations)
    - @Target
    - @Interface
    - @Documented
    - @Inherited
    - @Deprecated (Deprecated)
  - Used on Java code (In Classes, Methods etc.)
    - @Deprecated
    - @Override
    - @SuppressWarnings
    - @FunctionalInterface
    - @SafeVarargs
- Custom Annotations / User Defined (Our custom name)

Annotations used on Fast Code:

@Deprecated

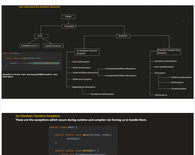
- Legacy and Deprecated Code, Backward Code, Backward compatible Code **DEPRECATED**
- Deprecated Code means, Backward Compatible Code, Legacy Code, Code that is not recommended to use in future versions, or Code that is no longer supported. Deprecated Code, Legacy Code, Backward Compatible Code, Legacy Code, Deprecated Code, Deprecated Code, Deprecated

Java: Exception Handling

"Concept && Coding" YT Video Notes

What is Exception?

- It's an event, that occurs during the execution of the program.
- It's an object that represents an error.
- It Creates the Exception Object, which contain information about the Error like
  1. Its Type of Exception and Message
  2. Stack trace etc.
- Runtime system see this Exception Object and find the class which can handle it.



Throwable
Error
Exception
RuntimeException
OutOfMemoryError
StackOverflowError
VirtualMachineError
ArithmeticException
ClassCastException
ArrayIndexOutOfBoundsException
NullPointerException
NumberFormatException
IndexOutOfBoundsException
ArrayStoreException
NegativeArraySizeException
IOException
FileNotFoundException
EOFException
SocketException
ClassNotFoundException
InterruptedException
CloneNotSupportedException
IllegalAccessException
InstantiationException
NoSuchMethodException
Checked Exceptions
Unchecked Exceptions
Runtime Exceptions

```java
public void myMethod(int schoolClassNumber) {

    try {
        int noOfStudents = getStudentCapacityofClass(schoolClassNumber);
        String[] names = new String[noOfStudents];
        names[0] = "new value";
    }
    catch (IndexOutOfBoundsException expObj){
        //do something
    }
    catch (Exception expObj){
        //do something
    }
}
```

But remember, exception handling is little expensive, if stack trance is huge and it is not handled or handled at parent class.

```java
public int myMethod(int a, int b) {
    int val;
    try{
         val = a/b;
    } catch (ArithmeticException exp){
        val = -1;
    }
    return val;
}

public int myMethod2(int a, int b) {

        if(b == 0){
            return -1;
        }

        int val = a/b;
        return val;
}
```

Try to avoid using exception handling, if you can. Like in this example, its not required

OPERATORS IN JAVA

* What are operators?
- This indicates what action to perform

* What is operand?
- This indicates the items, on which action has to apply on.

* What is expression?
- It consists of 1 or more operand and 0 or more operators.

Eg: 5 + 3
- 5 & 3 are operands
- + is operator
- 5+3 as a whole is kla expression

⇒ Categories Of Operators
There are 7 categories of operators

1) Arithmetic Operators
- / (Division)
- − (Subtraction)
- + (Addition)
- % (Modulus)
- * (Multiplication)

We already have information from mathematics of how arithmetic

operators work.

For eg:-

```java
public class Main {

    public static void main(String[] args) {

        int division = 5 / 2;
        System.out.println(division);

        int mod = 5 % 2;
        System.out.println(mod);

        int sum = 3 + 4;
        System.out.println(sum);

        int subtract = 4-3;
        System.out.println(subtract);

        int multiply = 3 * 4;
        System.out.println(multiply);
    }

}
```

Output:

```
2
1
7
1
12
```

2) Relational Operators

- These compares two operand relation and return true or false.
  - == (Equals to)
  - != (Not equals to)
  - > (Greater than)
  - < (Less than)
  - >= (Greater than or equals to)
  - <= (Less than or equals to)

For Eg:-

```java
public class Main {

    public static void main(String[] args) {

        int a = 4;
        int b = 7;
        System.out.println(a == b);
        System.out.println(a != b);
        System.out.println(a > b);
        System.out.println(a < b);
        System.out.println(a >= b);
        System.out.println(a <= b);
    }
}
```

Output:

```
false
true
false
true
false
true
```

3.) Logical Operators

- These combines two or more conditions and return true or false.
  - && (Logical AND) — Returns true only if all conditions are true.
  - || (Logical OR) - Returns true if atleast one condition is true

For Eg:-

```java
public class Main {

    public static void main(String[] args) {

        int a = 4;
        int b = 7;
        //AND operator
        System.out.println(a<3 && a!=b);
        System.out.println(a>3 && a!=b);
        //OR operator
        System.out.println(a<3 || a!=b);
        System.out.println(a>3 || a!=b);
    }
}
```

Output:

```
false
true
true
true
```

* As seen if one condition is false, && will not even evaluate further conditions.

4.) Unary Operators

- These require only a single operand
  - ++ (Increment)
  - -- (Decrement)
  - - (Unary Minus)
  - + (Unary Plus)
  - ! (Logical NOT)

* ++ & -- can be used before or after operand.

So Operand ++ ⇒ Postfix Increment , ++ operand ⇒ Prefix Increment

Operand -- ⇒ Postfix Decrement , -- operand ⇒ Prefix Decrement

- Postfix will return whatever value is first & then increments / decrements the value
- Prefix will first increment / decrement the value & then returns the value.

- Logical NOT operator will reverse the value i.e if current value is true, it'll change it to false & if the current value is false, it'll change the value to true.

- Unary + and - makes the value positive or negative.

For Eg :-

```java
public class Main {

    public static void main(String[] args) {
        int a = 5;
        boolean flag = true;

        //Increment operator
        System.out.println(a++);
        System.out.println(++a);

        //Decrement operator
        System.out.println(a--);
        System.out.println(--a);

        //Logical NOT operator
        System.out.println(!flag);

        //Unary Minus operator
        System.out.println(-a);

        //Unary Plus operator
        System.out.println(+a);
    }
}
```

Output:

```
5
7
7
5
false
-5
5
```

5.) Assignment Operators

- These are used to assign new value to the variable.
  - =
  - *=
  - +=
  - /=
  - -=
  - %=
- Assignment Operator assigns a value (on right side of operator) to a variable (on left side of the operator)

For Eg:-

```java
public class Main {

    public static void main(String[] args) {
        int a = 5;
        int variable;

        variable = a;
        System.out.println(variable);

        variable = 0;
        variable+=a;
        System.out.println(variable);

         variable-=3;
         System.out.println(variable);

         variable*=a;
         System.out.println(variable);

         variable/=a;
         System.out.println(variable);
    }
}
```

Output:

```
5
5
2
10
2
```

6) Bitwise Operators

- These works on bits i.e 1 and 0 and are very fast.
  - & (Bitwise AND)
  - | (Bitwise OR)
  - ^ (Bitwise XOR)
  - ~ (Bitwise NOT)

    (It can come under Unary too)

* Bitwise AND (&)

a	b		
0	0	⇒	0
0	1	⇒	0
1	0	⇒	0
1	1	⇒	1

* Bitwise OR (|)

a	b		
0	0	⇒	0
0	1	⇒	1
1	0	⇒	1
1	1	⇒	1

* Bitwise XOR (^)

a	b		
0	0	⇒	0
0	1	⇒	1
1	0	⇒	1
1	1	⇒	0

For Eg:-

```java
public class Main {

    public static void main(String[] args) {

        int a=4;
        int b=6;

        //Bitwise AND
        System.out.println(a & b);

        //Bitwise OR
        System.out.println(a | b);

        //Bitwise XOR
        System.out.println(a ^ b);

        //Bitwise NOT,  bitwise complement of any integer n is  -(n + 1)
        System.out.println(~a);
    }
}
```

Output:

```
4
6
2
-5
```

* How does bitwise NOT work?

— It basically reverses the bit

So 0 ⇒ 1

1 ⇒ 0

Now how does it works on numbers.

Let's compute ~4 (NOT of 4)

As we know in Java numbers are signed i.e Most Significant Bit (MSB) tells the sign of the number

So binary of 4 will be

2^3	2^2	2^1	2^0
0	1	0	0

(MSB) – Since MSB is 0 so it is positive.

Now NOT will reverse all the bits, so binary of ~4 will be

	2^3	2^2	2^1	2^0
(-)	1	0	1	1
	(MSB)			

Since MSB is 1 ie result is -ve, so the value of MSB will be negative

Now converting it to decimal gives us

$-8 + 0 + 2 + 1$

i.e -5

So $0100 \dashrightarrow \sim 0100 = 1011$

We can calculate it directly using the formula $-(N+1)$. So in our case it'll be $-(4+1)$ ie -5.

* How can we confirm if -5 is 1011

- To get -5, we know that we have to find its 2nd Complement

So for 4 ie 0101 $\Rightarrow$

1st Complement = 1010

2nd Complement = 1st complement + 1

i.e $1010 + 1 \Rightarrow 1011$

So bitwise NOT, bitwise complement of any integer n is $-(n+1)$

7) Bitwise Shift Operators

→ These are used to shift the bits of a number left or right.

- << (Signed Left Shift)
- >> (Signed Right Shift)
- >>> (Unsigned Right Shift

- There is no unsigned left shift as << and <<< are equal.

* >> : It's signed right shift

- It fills the most significant bit with the sign of the number

Eg:- 1) 11000110's >> will be

$$1\underbrace{1100011}_{\text{shifted right}}$$

MSB added as same sign of the original number

2) 01000110's >> will be

$$0\underbrace{0100011}_{\text{shifted right}}$$

MSB added as same sign of the original number

* >>> : It's unsigned right shift

- It fills the MSB with the 0;

Eg: 1) 11000110's >>> will be

$$0\underbrace{1100011}_{\text{shifted right}}$$

MSB filled with 0

2) 01000110's >>> will be

$$0\underbrace{0100011}_{\text{shifted right}}$$

MSB added as 0

Demo for both left & right shift :-

```java
public class Main {

    public static void main(String[] args) {

        int a=4;

        //left shift
        System.out.println(a<<1);
        System.out.println(a<<2);

        //right shift
        System.out.println(a>>1);
        System.out.println(a>>2);
    }
}
```

Output:

```
8
16
2
1
```

* For Left Shift LSB will always filled with 0.
* Left Shift once doubles the number
* Right Shift once halves the number
* There is no sense of <<< as LSB don't have any value

8) Ternary Operators

- It mimics the if else condition
- So, it evaluates the condition, it'll execute first expression otherwise it'll execute second expression.

- (Condition) ? Expression 1 : Expression 2

For Eg:-

```java
public class Main {

    public static void main(String[] args) {

        int a=4;
        int b=5;

        int maxValue = (a>b) ? a : b;
        System.out.println(maxValue);
    }
}
```

Output:

```
5
```

9) Type Comparison Operator

- It is used to do the type check, whether particular object is of a certain class or not.
- instanceOf

For Eg:-

```java
public class ParentClass {
}
```

```java
public class ChildClass1 extends ParentClass{
}
```

```java
public class ChildClass2 extends ParentClass{
}
```

```java
public class Main {

    public static void main(String[] args) {

        ParentClass obj = new ChildClass2();
        System.out.println( obj instanceof ChildClass2);
        System.out.println(obj instanceof ChildClass1);

        ChildClass1 childObj = new ChildClass1();
        System.out.println( childObj instanceof ParentClass);

        String val = "hello";
        System.out.println( val instanceof String);

        Object unknownObject = new RandomClass();
        System.out.println( unknownObject instanceof ChildClass2);
    }
}
```

Output:

```
true
false
true
true
false
```

* It'll return true when we perform the type check of an object of child class with its parent class.

⇒ Operator Precedence :-

- Associativity: If 2 operators have the same precedence, then it is evaluated based on its associativity (Left to Right or Right to Left).

High

Priority

Low

Operators	Precedence	Associativity
Parentheses	(), []	Left to right
Unary: Postfix	expr++ , expr--	Left to right
Unary: Prefix	++expr, --expr, +expr, -expr, ~, !	Right to Left
Multiplicative	*, /, %	Left to right
Additive	+, -	Left to right
Bitwise Shift	<<, >>, >>>	Left to right
Relational	<, >, <=, >=, instanceOf	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
Ternary	?:	Right to Left
Assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	Right to Left

* Let's solve an example expression

Int a = 4

Solve a = a + a++ + ++a * --a + a--

Now replace operands with values

a = ~~4~~ ~~5~~ ~~6~~ ~~5~~ 4

∴ a = 4 + 4 + 6 * 5 + 5

a = 4 + 4 + 30 + 5

a = 39

JAVA: Control flow statements

"Concept && Coding" YT Video Notes

Report Issue

Decision Making Statements

1. If-then (Simple if): If condition is true, then if block will get executed.

```
if(condition1 condition2 )
    {//code here get execute when condition is true
    }
```

```
public class Main {
    public static void main(String[] args) {
        int a = 5;
        if(a<5)
        System.out.println("a is smaller than 5");
        System.out.println("This line will execute");
    }
}
```

Output:

```
This line will execute
Process finished with exit code 0
```

2. If-else: If condition is true, if block will get executed else if condition is false, then else block will get executed.

```
public class Main {
    public static void main(String[] args) {
        int a = 5;
        if(a<5)
        {
            System.out.println("a is smaller than 5");
            System.out.println("This line will execute");
        }
        else
        {
            System.out.println("a is greater than 5");
            System.out.println("This line will execute");
        }
    }
}
```

Output:

```
This line will execute
a is greater than 5
This line will execute
Process finished with exit code 0
```

3. If-else if Ladder: It contains if statement with multiple (short off) else if statements. It evaluates the condition from top and goes down and as any condition gets true, its block will get executed.

```
public class Main {
    public static void main(String[] args) {
        int val = 15;
        if(val<10)
        System.out.println("val is less than 10");
        else if(val<20)
        System.out.println("val is less than 20");
        else if(val<30)
        System.out.println("val is less than 30");
        else
        System.out.println("val is greater than 30");
    }
}
```

Output:

```
val is less than 20
Process finished with exit code 0
```

4. Nested if Statement: If-else statement within if block or else block.

```
public class Main {
    public static void main(String[] args) {
        int a = 5;
        if(a<5)
        {
            System.out.println("a is smaller than 5");
            if(a<2)
            System.out.println("a is greater than 2 but smaller than 5");
            else
            System.out.println("a is greater than 2");
        }
        else
        {
            System.out.println("a is greater than 5");
            System.out.println("This line will execute");
        }
    }
}
```

Output:

```
a is smaller than 5
a is greater than 2
This line will execute
Process finished with exit code 0
```

5. Switch Statement: It is similar to if-else if ladder, based on condition particular block will get executed out of many alternatives.

Syntax:

```
switch(expression)
case value_1:
    //code statements
    break;
case value_2:
    //code statements
    break;
.
.
.
case value_N:
    //code statements
    break;
default:
    //default statements
    break; // optional
```

Alternative version

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

Output:

```
This line will execute
Process finished with exit code 0
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

Output:

```
This line will execute
Process finished with exit code 0
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

Output:

```
This line will execute
Process finished with exit code 0
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

Output:

```
This line will execute
Process finished with exit code 0
```

Few things we need to take care:

1. Two Cases can not have the same value.
2. Switch expression data type and case value/constant data type should be same.
3. Case value should be either LITERAL or CONSTANT.

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        int x = 2;
        switch(val)
        {
            case x:
            System.out.println("This line will execute");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        int x = 2;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

4. All use case need not to be handled:

```
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        int x = 2;
        switch(val)
        {
            case MONDAY:
            System.out.println("This line will execute");
            break;
            case TUESDAY:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            break;
        }
    }
}
```

Output:

```
Process finished with exit code 0
```

5. Nested Switch statement is possible:

```
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

```
public class Main {
    public static void main(String[] args) {
        int val = 5;
        int x = 2;
        switch(val)
        {
            case 1:
            System.out.println("This line will execute");
            break;
            case 2:
            System.out.println("This line will execute");
            break;
            default:
            System.out.println("This line will execute");
            switch(Day.MONDAY)
            {
                case MONDAY:
                System.out.println("This line will execute");
                break;
                default:
                System.out.println("This line will execute");
                break;
            }
        }
    }
}
```

Output:

```
This line will execute
This line will execute
Process finished with exit code 0
```

6. Supported data types:

- **Primitive types:** int, short, byte, char
- **Wrapper Types of above primitive data types** i.e. Integer, Short, Byte, Character.
- **Enum**
- **String**

7. Return is not possible within Switch case.

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

        String day = "";
        int val = 1;
        String result = switch (val){

            case 1 -> "One";
            case 2 -> "Two";
            default -> "None";
        };
        System.out.println(result);
    }
}
```

There are 2 ways to do it:

1. **Using "case N ->" Label.**
2. **Using "yield" statement.**

Use of "case N ->" :

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

        String day = "";
        int val = 1;
        String outputVal = switch (val){

            case 1 -> "One";
            case 2 -> "Two";
            default -> "None";
        };
        System.out.println(outputVal);
    }
}
```

8. All possible use cases need to be handled for the expression.

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

        String day = "";
        int val = 1;
        String result = switch (val){

            case 1 -> "One";
            case 2 -> "Two";
            default -> "None";
        };
        System.out.println(result);
    }
}
```

Switch expression does not cover all possible inputs
Switch label values: int, char, byte, short, enum, String

9. Using this "->" we can not have block of statements. If we want block of statements and then return the value, we need to use "yield" .

Use of "yield" :

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

        String day = "";
        int val = 1;
        String outputVal = switch (val){

            case 1 -> {
                //some code logic here
                yield "One";
            }
            case 2 -> {
                //some code logic here
                yield "Two";
            }
            default -> "None";
        };

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

Iterative Statements:

1. For Loop:

```
for(initialization of variable; condition check; increment/decrement of variable) {
    //statement block
}
```

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

        for(int i=1; i<5; i++)
        System.out.print(i);
        System.out.print("\n");
        for(int i=5; i>1; i--)
        System.out.print(i);
    }
}
```

Output:

```
1 2 3 4
5 4 3 2
Process finished with exit code 0
```

```
while(condition) {
    //statement block
    //increment/decrement variable
}
```

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

        int i = 1;
        while(i<5){
            System.out.print(i);
            i++;
        }
        System.out.print("\n");
        int j = 5;
        while(j>1){
            System.out.print(j);
            j--;
        }
    }
}
```

```
1 2 3 4
5 4 3 2
Process finished with exit code 0
```

2. While Loop:

Initialize variable;
While(condition check){
//block of statements
//increment/decrement variable
}

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

        int i = 1;
        while(i<5){
            System.out.print(i);
            i++;
        }
        System.out.print("\n");
        int j = 5;
        while(j>1){
            System.out.print(j);
            j--;
        }
    }
}
```

Output:

```
1 2 3 4
5 4 3 2
Process finished with exit code 0
```

3. Do-While Loop:

Initialize variable;
do{
//block of statements
//increment/decrement variable
}while(condition check);

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

        int i = 1;
        do{
            System.out.print(i);
            i++;
        }while(i<5);
        System.out.print("\n");
        int j = 5;
        do{
            System.out.print(j);
            j--;
        }while(j>1);
    }
}
```

Output:

```
1 2 3 4
5 4 3 2
Process finished with exit code 0
```

4. For each loop:

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

        int[] arr = {1, 2, 3, 4, 5};
        for(int ele : arr){
            System.out.print(ele);
        }
    }
}
```

Output:

```
1 2 3 4 5
Process finished with exit code 0
```

Branching Statements:

1. break statement:

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

        for(int val=1; val<=10 ; val++){

            if(val == 3){
                break;
            }
            System.out.println(val);
        }
    }
}
```

Output:

```
1
2

Process finished with exit code 0
```

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

        for (int outerLoop = 1; outerLoop <= 5; outerLoop++)
            for (int innerLoop = 1; innerLoop <= 5; innerLoop++) {

                if (innerLoop == 2) {
                    break;
                }
                System.out.println(outerLoop + "," + innerLoop);
            }
    }
}
```

Output:

```
1,1
2,1
3,1
4,1
5,1

Process finished with exit code 0
```

2. continue statement:

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

        for(int val=1; val<=10 ; val++){

            if(val == 3){
                continue;
            }
            System.out.println(val);
        }
    }
}
```

Output:

```
1
2
4
5
6
7
8
9
10

Process finished with exit code 0
```

(Concept && Coding) Java Collections: Page1

Java Collections Framework
Introduction

- **What is Java Collections Framework?**
  - Added in Java version 1.2
  - Collection is a building block of Objects.
  - Collection is a group of Objects.
  - Framework provide us the architecture to manage those "group of objects" i.e. add, update, delete, search etc.

Why we need Java Collections Framework?

- Prior to JDK1 we have Array, Vector, Hash tables.
  But problem with them is, there is no commonly interface, so its difficult to remember the methods for each.

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

        int a[] = {4,6,1,9,2,11};
        Arrays.sort(a, (int[][] val1, int[][] val2) -> val1[-val1[1]],
        val2[-val2[1]]);

        for(int v : a){
            System.out.println(v);
        }
    }
}
```

Output:

```
11
9
6
4
2
1
```

```
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}

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

        Car[] cars = new Car[3];
        cars[0] = new Car("Ford", "SUV");
        cars[1] = new Car("BMW-750xi", "SUV");
        cars[2] = new Car("Ford", "Hatchback", "Sub-Type:S");

        Arrays.sort(cars, (Car v1[], Car v2[]) -> v1[0].carType.compareTo(v2[0].carType));

        for(Car car : cars){
            System.out.print(car.carName + " - " + car.carType);
        }
    }
}
```

Output:

```
suv..petrol
Sedan..diesel
Hatchback..cng
```

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

        Car[] cars=new Car[3];
        cars[0]=new Car("sedan","petrol","2000");
        cars[1]=new Car("sedan","diesel","1500");
        cars[2]=new Car("hatchback","CNG","1000");

        Arrays.sort(cars, (Car o1, Car o2) -> o1.carName.compareTo(o2.carName));

        for(Car car : cars){
            System.out.print(car.carName + " " + car.carType);
        }
    }
}
```

Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

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

        List<Car> cars = new ArrayList<>();
        cars.add(new Car("suv","petrol","2000"));
        cars.add(new Car("sedan","diesel","1500"));
        cars.add(new Car("hatchback","CNG","1000"));

        Collections.sort(cars, (Car o1, Car o2) -> o1.carName.compareTo(o2.carName));

        cars.forEach(Car car0 -> System.out.print(car0.carName + " " + car0.carType));
    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}

public class CarNameComparator implements Comparator<Car> {

    @Override
    public int compare(Car o1, Car o2) {
        return o1.carName.compareTo(o2.carName);
    }
}

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

        List<Car> cars = new ArrayList<>();
        cars.add(new Car("suv","petrol","2000"));
        cars.add(new Car("sedan","diesel","1500"));
        cars.add(new Car("hatchback","CNG","1000"));

        Collections.sort(cars, new CarNameComparator());

        cars.forEach(Car car0 -> System.out.print(car0.carName + " " + car0.carType));
    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```
public class Car implements Comparator<Car> {

    String carName;
    String carType;

    Car(){

    }
    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }

    @Override
    public int compare(Car o1, Car o2) {
        return o1.carName.compareTo(o2.carName);
    }
}

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

        List<Car> cars = new ArrayList<>();
        cars.add(new Car("suv","petrol","2000"));
        cars.add(new Car("sedan","diesel","1500"));
        cars.add(new Car("hatchback","CNG","1000"));

        Collections.sort(cars, new Car());

        cars.forEach(Car car0 -> System.out.print(car0.carName + " " + car0.carType));
    }
}
```

Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

Deque:
Stands for Double Ended Queue. Means addition and removal can be done from both the sides of the queue.

Methods available in Deque Interface

Methods available in Collection, Queue interface + Methods available in Deque interface

Queue
add(), offer(), poll(), remove(), peek(), element()

Deque

	Throws exception	Returns special value	Throws exception	Returns special value
Insert to front	addFirst(e)	offerFirst(e)	removeFirst()	pollFirst()
Remove from front	removeFirst()	pollFirst()	getFirst()	peekFirst()
Examine front	getFirst()	peekFirst()		
Insert to rear	addLast(e)	offerLast(e)	removeLast()	pollLast()
Remove from rear	removeLast()	pollLast()	getLast()	peekLast()
Examine rear	getLast()	peekLast()		

Using this methods, we can also use Deque to implement FIFO(First In First Out) and LIFO(Last In First Out).
Deque can be used as a Queue and Stack instead as an element.
add() = removeLast() add(e) = removeFirst() offer(e) = offerFirst(e)
remove() = removeFirst() poll() = pollFirst()
peek() = peekFirst() element() = getFirst()
offerFirst(e) = offerLast(e) offerLast(e) = offerFirst(e)
removeFirst() = pollFirst() removeLast() = pollLast()
getFirst() = peekFirst() getLast() = peekLast()

ArrayDeque

ArrayDeque (Concrete class), implements the methods which are available in Queue and Deque interface.

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

        ArrayDeque<Integer> arrDeque=new ArrayDeque<>();
        arrDeque.add(1);
        arrDeque.offer(2);
        arrDeque.addFirst(3);
        arrDeque.offerFirst(4);
        arrDeque.addLast(5);
        arrDeque.offerLast(6);
        System.out.
        for(int i : arrDeque){
            System.out.print(i+" ");
        }
        System.out.
        arrDeque.peek();
        arrDeque.peekFirst();
        arrDeque.peekLast();
        arrDeque.element();
        arrDeque.getFirst();
        arrDeque.getLast();
        System.out.
        arrDeque.remove();
        arrDeque.removeFirst();
        arrDeque.removeLast();
        arrDeque.poll();
        arrDeque.pollFirst();
        arrDeque.pollLast();
        System.out.
    }
}
```

add() = removeLast() offer(e) = offerFirst(e)
addFirst(e) = offerLast(e) offerLast(e) = offerFirst(e)
peekFirst() = getFirst() peekLast() = getLast()
element() = getFirst() remove() = removeFirst()
removeFirst() = pollFirst() removeLast() = pollLast()
poll() = pollFirst() pollFirst() = removeFirst()
pollLast() = removeLast()

Time Complexity:

Insertion: Amortized(time of the cities is changed) complexity is O(1) except from case this
Deletion: Amortized(time of the cities is changed) complexity is O(1) except from case this
Search: O(n)
Space Complexity:
O(n)

Queue Implementations

	Unbounded	Bounded	Blocking	Blocking	Priority ordering	Priority ordering
	ArrayDeque	ArrayBlockingQueue	LinkedBlockingQueue	LinkedBlockingDeque	PriorityQueue	PriorityBlockingQueue
Not thread safe	No	NO	NO	No	yes	Yes

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

        ArrayDeque<Integer> arrDeque=new ArrayDeque<>();
        arrDeque.add(1);
        arrDeque.offer(2);
        arrDeque.addFirst(3);
        arrDeque.offerFirst(4);
        arrDeque.addLast(5);
        arrDeque.offerLast(6);
        System.out.
        for(int i : arrDeque){
            System.out.print(i+" ");
        }
        System.out.
        arrDeque.peek();
        arrDeque.peekFirst();
        arrDeque.peekLast();
        arrDeque.element();
        arrDeque.getFirst();
        arrDeque.getLast();
        System.out.
        arrDeque.remove();
        arrDeque.removeFirst();
        arrDeque.removeLast();
        arrDeque.poll();
        arrDeque.pollFirst();
        arrDeque.pollLast();
        System.out.
    }
}
```

Output:

```
4 3 1 2 5 6
4 3 1 2 5 6
4 3 1 2 5 6
```

Comparison

Concurrent	No	Yes	Yes	Yes	

```
public class Test {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
        map.put(1, "A");
        System.out.println(map.get(1));
    }
}
```

Vector

Vector is a legacy synchronized implementation of List.
Every method is synchronized, meaning only one thread can access the vector at a time.
This makes it safe for multi-threaded environments.
Slow in single-threaded applications because of the overhead of synchronization.
Use Collections.synchronizedList() or CopyOnWriteArrayList instead for better performance.

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

        Vector<Integer> v = new Vector<>();
        v.add(10);
        v.add(20);

        System.out.println(v.get(0));

    }
}
```

Operation	Time Complexity	Thread Safe	Allows Duplicates	Allows Null Elements	Preserves Insertion Order	Implementation
Insert	O(1)	Yes	Yes	Yes	Yes	N/A

Stack

- **Represents LIFO (Last In First out) operations**
- **Since it extends Vector, its method is also Synchronized.**

How its different from Deque : Deque is not thread safe, stack is.

```
public class StackExample {

    public static void main(String[] args) {

        Stack<Integer> stack = new Stack<>();
        stack.push(10);
        stack.push(20);

        System.out.println(stack.pop());

    }
}
```

Time Complexity:

- Push(add) : O(1)
- Pop(remove) : O(1)
- Peek(get) : O(1)
- Search: O(n)

Space Complexity:
O(n)

Operation	Time Complexity	Thread Safe	Allows Duplicates	Allows Null Elements	Preserves Insertion Order	Implementation
Push	O(1)	Yes	N/A	Yes	Yes	N/A

Java Collection: Page2

"Concept && Coding" YT Video Notes

Map Properties:

- Its an interface and its implementations are:
  - HashMap: do not maintains the order
  - HashTable: Synchronized version of HashMap
  - LinkedHashMap: Maintains the insertion order.
  - TreeMap: sorts the data internally.
- Object that maps keys to values.
- Can not contain duplicate key.

Methods available in Map Interface:

Methods available in Map Interface

No.	Method Name	Usage
1.	size()	returns the number of key-value mapping present in the map
2.	isEmpty()	returns true if map contains no key-value mapping else false
3.	containsKey(Object key)	if given key is already present in the map returns true else false
4.	containsValue(Object value)	return true if map contains one or more key-value pairs with given value, return false if no value to which this key is mapped
5.	put(Object key, value)	if key is already has the same key as given key, it will overwrite the value now possible
6.	putIfAbsent(Object key, value)	do not have the key present, it will add new key-value mapping, if key already present with some value then it will not add the new value
7.	remove(Object key)	remove the mapping from the specified key in the map, removes all the mappings from the map
8.	clear()	clear the map, removes all the key-value mappings from the map
9.	keySet()	returns a set view of the keys contained in this map, the set is backed by the map, so changes in the Map will be reflected in the Set and vice versa.
10.	Collection<V> values()	returns a collection view of the values contained in this map.
11.	Set<Map.Entry<K,V>> entrySet()	returns the set view of the mappings present in the Map.
12.	putIfAbsent(K key, V value)	if key already exists return the value already associated, else add the specified key-value pair to the map.
13.	get(Object key, defaultValue)	if key not present in map, return the specified default value.
14.	Entry sub-interface	Entry is the sub-interface of the Map. So, it is accessed as Map.Entry

Method Name	Usage
getKey()	returns the key
getValue()	returns the value
setValue(value)	returns the old value
boolean equals(Object o)	compare the 2 objects

HashMap:

- put() permit null key or value (HashTable do not contains null key or value)
- Hash Map do not maintains the insertion order
- Its not thread safe (instead use ConcurrentHashMap or HashTable for thread safe HashMap implementation)

Time complexity:

add : amortized O(1)
remove: Amortized O(1)
get: Amortized O(1)

LinkedHashMap:

1. Helps in Maintain Insertion order
   or
   Helps in Maintain Access order
2. Similar to HashMap, but also uses Double LinkedList.

```
map.put("K","L");
map.put("L","B");
map.put("B","C");
map.put("C","D");
map.put("D","L");
```

TreeMap:

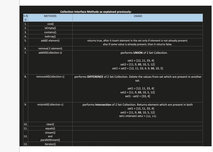
Methods Available in SORTEDMAP Interface:

Methods Available in NAVIGABLEMAP Interface:

S.NO	METHOD NAME	USAGE
1.	Map.Entry<K,V> lowerEntry(K key)	return the Entry(key,value) entry with the first key that is strictly less than given key.
2.	K lowerKey(K key)	returns the Key only which is less, or equal than key. If there is no key returns null
3.	Map.Entry<K,V> floorEntry(K key)	return the Entry(key,value) entry with the greatest key that is less or equal than key. If there is no value returns null
4.	K floorKey(K key)	returns the Key only which is greater or equal than key. If there is no value returns null
5.	Map.Entry<K,V> ceilingEntry(K key)	return the Entry(key,value) entry with the least key that is greater than given key. If there is no value returns null
6.	K ceilingKey(K key)	returns the Key only which is greater or equal than key. If there is no value returns null
7.	Map.Entry<K,V> higherEntry(K key)	return the Entry(key,value) entry with the least key that is greater than key. If there is no value returns null
8.	K higherKey(K key)	returns the Key only which is greater than key. If there is no value returns null

S.NO	METHOD NAME	USAGE
9.	Map.Entry<K,V> firstEntry()	map.firstEntry(); output: 1=1
10.	K firstKey()	map.firstKey(); output: 1
11.	Map.Entry<K,V> lastEntry()	map.lastEntry(); output: 5=5
12.	K lastKey()	map.lastKey(); output: 5
13.	Map.Entry<K,V> pollFirstEntry()	map.pollFirstEntry(); output: 1=1 removes and returns the first entry from the map. Now 1=1 would be removed from the map
14.	Map.Entry<K,V> pollLastEntry()	map.pollLastEntry(); output: 5=5 removes and returns the last entry from the map. Now 5=5 would be removed from the map
15.	NavigableMap<K,V> descendingMap()	map.descendingMap(); {5=5, 2=2, 1=1} if nothing is sorted if ascending, key will come in ascending, if descending, key will come in descending
16.	NavigableSet<K> descendingKeySet()	map.descendingKeySet(); [5, 2, 1] if nothing is sorted if ascending, key will come in ascending, if descending, key will come in descending
17.	NavigableMap<K,V> headMap(K toKey, boolean inclusive)	map.headMap(5, true); {1=1, 2=2, 5=5}
18.	NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)	map.tailMap(5, true); {5=5, 21=21}

SET:



Java provides SET:

1. Collections of Objects, but it does not contains duplicate value (any only one "null" value you can insert).
2. Unlike List, Set is not an Ordered collection, hence objects inside set does not follow the insertion order.
3. Set is not a List.

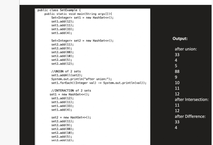
Few questions should have come to our mind:

1. What Data Structure is used in Set internally, so it does not allow duplicate values?
2. In order to not guarantee, then what if we want to get the ordered set?

We will try to find the answer of both the above questions, as we go further.....

What all methods SET interface contains:

All methods which are declared in Collections interface, generally that only is available in SET interface. No new method specific to SET is added.



Collection interface Methods & explained previously	
Method	Usage
1. add(E e)	
2. clear()	
3. contains(Object o)	
4. isEmpty()	
5. iterator()	
6. remove(Object o)	
7. size()	
8. addAll(Collection c)	returns true, after it over-ulmost is the same as it doesn't is only allowed unique elements
9. removeAll(Collection c)	deletes the common elements from the Set Collections
10. retainAll(Collection c)	performs INTERSECT of 2 Set Collections. Returns true if set1 was changed set1.retainAll(set2); set1 = {1, 2, 3, 4, 5} set2 = {4, 5, 6, 7, 8} set1.retainAll(set2); set1 = {4, 5} set2 = {4, 5, 6, 7, 8}
11. removeAll(Collection c)	performs **DIFFERENCE** of 2 Set Collections. Returns true if set1 was changed set1.removeAll(set2); set1 = {1, 2, 3, 4, 5} set2 = {4, 5, 6, 7, 8} set1.removeAll(set2); set1 = {1, 2, 3}
12. equals(Object o)	returns true, if both the sets are same, order does not matter
13. hashCode()	
14. toArray()	
15. toArray(T[] a)	

Set Interface:

- Data structure used : HashSet
- HashSet(E...Objects/elements) = new HashMap<>();

- During Add/ Remove/ Modification, it stored the element in the key part and it a value it stores the dummy object (**static private/private, new Object()**
  when E if values get the same hash value/ Free to handled/ when is load factor

- No guarantee that the order will remain consistent.
- Hashset is not threadSafe. **newHashSet** method present in **ConcurrentHashMap** class is used to create threadSafe Set.

Collection	IsThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion
HashSet	No	NO	Yes (only one)	NO	***newKeySet** method* present in ***ConcurrentHashMap*** class

```java
public class ThreadSafeSet {

    public static void main(String[] args){

        ConcurrentHashMap concurrentHashMap = new ConcurrentHashMap<>();
        Set<Integer> threadSafeSet = concurrentHashMap.newKeySet();
        threadSafeSet.add(1);
        threadSafeSet.add(2);

        Iterator<Integer> iterator = threadSafeSet.iterator();
        while (iterator.hasNext()){

            int val = iterator.next();
            if(val == 1){
                // iterator.remove(); //we can  remove
                threadSafeSet.add(5); //we should be able to add in the set as its thread safe
            }
        }
        threadSafeSet.forEach((Integer val) -> System.out.println(val));
    }
}
```

Without Thread safe, addition of element while iterating

```java
public class HashSetExample {

    public static void main(String[] args){
        Set<Integer> set1 = new HashSet<>();
        set1.add(1);
        set1.add(2);
        set1.add(3);
        set1.add(4);

        Iterator<Integer> iterator = set1.iterator();
        while (iterator.hasNext()){

            int val = iterator.next();
            if(val == 1){
                // iterator.remove(); //we can also remove
                set1.add(5); //we should set be able to add in the set as its not thread safe.
            }
        }
        set1.forEach((Integer val) -> System.out.println(val));
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1469)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1493)
    at HashSetExample.main(HashSetExample.java:25)
```

Time complexity:

add : O(1)
remove: Amortized O(1)
contains: Amortized O(1)

LinkedHashSet:

- Internally it uses: LinkedHashMap
- Maintains the insertion Order of the element
- Its not thread safe:

Set<Integer>set=Collections.*synchronizedMap*(new LinkedHashSet<>());

```java
public class LinkedHashSetExample {

    public static void main(String args[]){

        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

       Iterator<Integer> iterable =  intSet.iterator();
       while(iterable.hasNext()){
           int val = iterable.next();
           System.out.println(val);
       }
    }
}
```

Output:

```
2
77
82
63
5
```

TreeSet:

- Internally it uses: TreeMap
- It can not store null value

```java
public class SetExample {

    public static void main(String args[]){

       Set<Integer> treeSet = new TreeSet<>();
        treeSet.add(2);
        treeSet.add(77);
        treeSet.add(82);
        treeSet.add(63);
        treeSet.add(5);
      Iterator<Integer> iterable2 =  treeSet.iterator();

        while(iterable2.hasNext()){
            int val = iterable2.next();
            System.out.println(val);
        }
    }
}
```

Output:

```
2
5
63
77
82
```

STREAMS in Java8

"Concept && Coding" YT Video Notes

Stream (Java8)

Stream<T>, Stream<Integer>

What is Stream?

- We can consider Stream as a pipeline, through which our collection elements passes through.
- While elements passes through pipelines, it perform various operations like sorting, filtering etc.
- Useful when deals with bulk processing. (can do parallel processing)

Different ways to create a stream:

1. From Collection:

```
List<Integer> salaryList = Arrays.asList(3000, 4100, 5000, 2800, 3500);
Stream<Integer> streamFromCollection = salaryList.stream();
```

2. From Array:

```
int[] salaryArray = {3000, 4100, 5000, 2800, 3500};
Stream<Integer> streamFromArray(Arrays) = Arrays.stream(salaryArray);
```

3. From Static Method:

```
Stream<Integer> streamFromStaticMethod = Stream.of(1, 2, 3, 4, 5);
```

4. From Stream Builder:

```
Stream.Builder<Integer> streamBuilder = Stream.builder();
streamBuilder.add(2001).add(2002).add(2003);

Stream<Integer> streamFromStreamBuilder = streamBuilder.build();
```

5. From Stream Iterate:

```
Stream<Integer> streamFromIterate = Stream.iterate(2001-1000, Integer x) -> x + 1000).limit(3);
```

Different Intermediate Operations

We can chain multiple intermediate operations together to perform more complex processing before applying terminal operation to produce the result.

	Intermediate Operations	Description
1.	filter(Predicate<T> p-predicate)	Filters the element.
2.	map(Function<T, R> mapper)	Used to transform each element.
3.	flatMap(Function<T, Stream<R>> mapper)	Removes duplicates from the stream.
4.	distinct()	Sorts the elements.
5.	sorted()	Sorts the elements.
6.	peek(Consumer<T> action)	Helps you to see the elements of the stream as it is getting processed.
7.	limit(long maxSize)	Truncate the stream, so have no longer than given maxSize.
8.	skip(long n)	Skip the first n elements of the stream.
9.	mapToInt(ToIntFunction<T> mapper)	Helps to work with primitive "int" data type.
10.	mapToLong(ToLongFunction<T> mapper)	Helps to work with primitive "long" data type.
11.	mapToDouble(ToDoubleFunction<T> mapper)	Helps to work with primitive "double" data type.

Different Terminal Operations

Terminal operations are the ones that produces the result. It triggers the processing of the stream.

	Terminal Operations	Description
1.	forEach(Consumer<T> c)	Iterate over the elements of the stream.
2.	collect(Collector<T, A, R> collector)	Helps to store the elements of the stream into a collection.
3.	count()	Gives the count of elements in the stream.
4.	min(Comparator<T> c)	Gives the minimum element in the stream.
5.	max(Comparator<T> c)	Gives the maximum element in the stream.
6.	reduce(BinaryOperator<T> accumulator)	Gives reduction in the elements of the stream by applying the given aggregation function.
7.	findFirst()	Gets the first element of the stream.
8.	findAny()	Gets the first element of the stream.
9.	anyMatch(Predicate<T> p-predicate)	Checks if any value in the stream satisfies the given predicate condition and return true/false.
10.	allMatch(Predicate<T> p-predicate)	Checks if all values in the stream satisfies the given predicate condition and return true/false.
11.	noneMatch(Predicate<T> p-predicate)	Checks if no value in the stream satisfies the given predicate condition and return true/false.
12.	toArray()	Stores the final element of the stream.
13.	toArray(IntFunction<A[]> generator)	Stores the final element of the stream.
14.	findFirst()	Gives any random element of the stream.

How many times we can use a single stream:

- Java Terminal Operation is used on a stream, it is closed/consumed and can not be used again for another terminal operation.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> firstStreamObject = numbers.stream();
                .forEach(System.out::println);

firstStreamObject.forEach(System.out::println); // consumed the firstStreamObject stream

List<Integer> secondStreamObject = numbers.stream();
                .collect(Collectors.toCollection(ArrayList::new));
```

Output:

```
1
2
3
4
5
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.base/java.util.AbstractPipeline.sourceStagePut(
```

Sequence of Stream Operations:

Different Terminal Operations:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream()
        .forEach(Integer -> {
            System.out.print(Integer + " ");
        });
// Output: 1 2 3 4 5

numbers.stream()
        .forEach(Integer -> {
            System.out.print(Integer + " ");
            if (Integer == 3)
                return;
        });
// Output: 1 2 3 4 5

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Object[] firstEvenNumberArray = numbers.stream()
        .filter(Integer -> Integer % 2 == 0)
        .toArray();
// Output: [2, 4]

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Object[] firstEvenNumberArray = numbers.stream()
        .filter(Integer -> Integer % 2 == 0)
        .toArray(size -> new Integer[size]);
// Output: [2, 4]
```

```
int[] numbers = {1, 2, 3, 4, 5};
int sum = Arrays.stream(numbers).reduce(0, (a, b) -> a + b);
System.out.println("Sum: " + sum);
// Output: Sum: 15

int[] numbers = {1, 2, 3, 4, 5};
int product = Arrays.stream(numbers).reduce(1, (a, b) -> a * b);
System.out.println("Product: " + product);
// Output: Product: 120

int[] numbers = {1, 2, 3, 4, 5};
int maxNumber = Arrays.stream(numbers).reduce(Integer.MIN_VALUE, (a, b) -> Math.max(a, b));
System.out.println("Max number: " + maxNumber);
// Output: Max number: 5

int[] numbers = {1, 2, 3, 4, 5};
int minNumber = Arrays.stream(numbers).reduce(Integer.MAX_VALUE, (a, b) -> Math.min(a, b));
System.out.println("Min number: " + minNumber);
// Output: Min number: 1

int[] numbers = {1, 2, 3, 4, 5};
int firstNumber = Arrays.stream(numbers).reduce((a, b) -> a);
System.out.println("First number: " + firstNumber);
// Output: First number: 1

int[] numbers = {1, 2, 3, 4, 5};
int lastNumber = Arrays.stream(numbers).reduce((a, b) -> b);
System.out.println("Last number: " + lastNumber);
// Output: Last number: 5
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
boolean hasEvenNumber = numbers.stream()
        .anyMatch(Integer -> Integer % 2 == 0);
System.out.println("Has even number: " + hasEvenNumber);
// Output: Has even number: true

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
boolean allPositive = numbers.stream()
        .allMatch(Integer -> Integer > 0);
System.out.println("All positive: " + allPositive);
// Output: All positive: true

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
boolean noNegative = numbers.stream()
        .noneMatch(Integer -> Integer < 0);
System.out.println("No negative: " + noNegative);
// Output: No negative: true
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Object[] arrayFromStream = numbers.stream().toArray();
System.out.print("Array from stream: ");
for (Object obj : arrayFromStream) {
    System.out.print(obj + " ");
}
// Output: Array from stream: 1 2 3 4 5

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Integer[] arrayFromStream = numbers.stream().toArray(size -> new Integer[size]);
System.out.print("Array from stream: ");
for (Integer obj : arrayFromStream) {
    System.out.print(obj + " ");
}
// Output: Array from stream: 1 2 3 4 5
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstElement = numbers.stream().findFirst();
System.out.print("First element: ");
if (firstElement.isPresent()) {
    System.out.print(firstElement.get());
} else {
    System.out.print("No elements");
}
// Output: First element: 1

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> anyElement = numbers.stream().findAny();
System.out.print("Any element: ");
if (anyElement.isPresent) {
    System.out.print(anyElement.get());
} else {
    System.out.print("No elements");
}
// Output: Any element: 1
```

How many times we can use a single stream:

- Java Terminal Operation is used on a stream, it is closed/consumed and can not be used again for another terminal operation.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> firstStreamObject = numbers.stream();
                .forEach(Integer -> System.out.print(Integer + " "));

firstStreamObject.forEach(Integer -> System.out.print(Integer + " "));

List<Integer> secondStreamObject = numbers.stream();
                .collect(Collectors.toCollection(ArrayList::new));
```

Output:

```
1 2 3 4 5
1 2 3 4 5
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.base/java.util.AbstractPipeline.sourceStagePut(
```

Parallel Streams:

Parallel streams enable multiple threads to work on different parts of the stream simultaneously, leveraging multi-core processors. This can significantly improve performance for large datasets or computationally intensive operations.

- **Parallel processing:** Divides the stream into multiple chunks, each processed by a separate thread.
- **Fork and Join:** Uses the "Fork-Join" framework to split the data into multiple chunks, process them independently, and then combine the results.
- **Thread Safety:** Ensures that shared data is handled safely to prevent race conditions.
- **Overhead:** Consider the overhead of creating and managing threads when using parallel streams.
- **Ordering:** Maintains the original order of elements unless explicitly specified otherwise.
- **Stateful operations:** Be careful with stateful operations, as they can introduce complexity in parallel processing.
- **Performance:** Measure the performance to ensure that parallel streams provide a benefit in your specific scenario.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
System.out.print("Output: ");
numbers.parallelStream()
        .forEach(Integer -> {
            System.out.print(Integer + " ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
// Output: Output: 1 2 3 4 5 6 7 8 9 10

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
System.out.print("Output: ");
numbers.parallelStream()
        .forEach(Integer -> {
            System.out.print(Integer + " ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
// Output: Output: 1 2 3 4 5 6 7 8 9 10
```

Multithreading - Part1

"Concept && Coding" YT Video Notes

Introduction of Multithreading

- **Before we understand what is Multithreading, lets first understand Thread and Process**

Process:

Process is an instance of a program that is getting executed.
It has its own resource like memory, thread etc. OS allocate these resources to process when its created.

Compilation (*javac Test.java*) : generates bytecode that can be executed by JVM.

→ Execution (*java Test*) : at this point, JVM starts the new Process, here ***Test*** is the class which has ***"public static void main(String args[])"*** method.

How much memory does process gets?
While creating the process "**java MainclassName**" command, a new JVM instance will get created and we can tell how much heap memory need to be allocated.

```
java -Xms256m -Xmx2g MainClassName
```

-Xms<size>:
This will set the initial heap size, above, i allocated 256MB

-Xmx<size>:
Max heap size, process can get, above, i allocated 2GB, if tries to allocate more memory, "OutOfMemoryError" will occur

Thread:

- Thread is know as ligtweight process
  OR
  Smallest sequence of instructions that are executed by CPU independently.

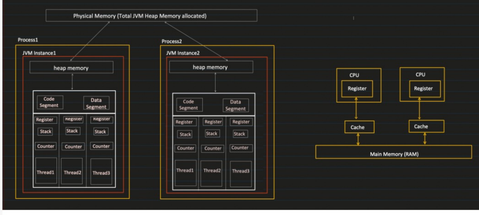
- And 1 process can have multiple threads.

- When a Process is created, it start with 1 thread and that initial thread know as 'main thread' and from that we can create multiple threads to perform task concurrently.

```
public class MultithreadingLearning {

    public static void main(String args[]){
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

Output: `Thread Name: main`

Let's understand little bit more about Process and Threads:



Code Segment:

- Contains the compiled **BYTECODE** *(i.e machine code)* of the Java Program.
- Its read only.
- All threads within the same process, share the same code segment.

Data Segment:

- Contains the GLOBAL and STATIC variables.
- All threads within the same process, share the same data segment.
- Threads can read and modify the same data.
- Synchronization is required between multiple threads.

Heap :

- Objects created at runtime using "new" keyword are allocated in the heap.

- Heap is shared among all the threads of the same process. (but NOT WITHIN PROCESS)
  (let say in Process1, X8000 heap memory pointing to some location in physical memory, same X8000 heap memory point to differet location for Process2)

- Threads can read and modiy the heap data.

- Synchronization is required between multiple threads.

Stack:

- **Each thread has its own STACK.**
- **It manages, method calls, local variables.**

Register:

- **When JIT (Just-in time) compiles converts the Bytecode into native machine code, its uses register to optimized the generated machine code.**
- **Also helps in context switching.**
- **Each thread has its own Register.**

Counter:

- **Also know as Program Counter, it points to the instruction which is getting executed.**
- **Increments its counter after successfully exectuion of the instruction.**

All these are managed by JVM.

Definition of Multithreading:

- **Allows a program to perform multiple task at the same time.**
- **Multiple threads share the same resource such as memory space but still can perform task independently.**

Benefits and Challenges of Multithreading:

Benefits :

- **Imporved performance by task parallelism**
- **Responsiveness**
- **Resource sharing**

Challenges:

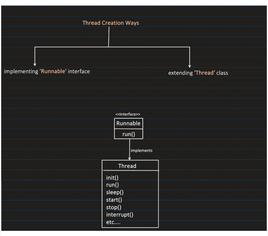
- **Concurrency issue like deadlock, data inconsistency etc.**
- **Synchronized overhead.**
- **Testing and Debugging is difficult.**

Multitasking vs Multithreading

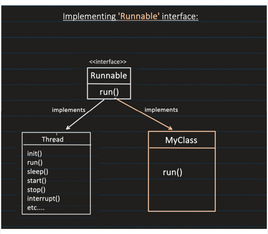
Report Abuse

Multithreading : Part2

"Concept && Coding" YT Video Notes



Report Abuse



Step1: Create a Runnable Object

- Create a class that implements 'Runnable' interface.
- Implement the "run()" method to tell the task which thread has to do.

```
public class MultiThreadingLearning implements Runnable{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

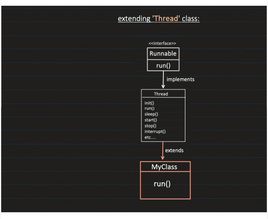
Step2: Start the thread

- Create an instance of class that implement 'Runnable'.
- Pass the Runnable object to the Thread Constructor.
- Start the thread.

```
public class Main {
    public static void main(String args){

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultiThreadingLearning runnableObj = new MultiThreadingLearning();
        Thread myThread = new Thread(runnableObj);
        myThread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```



Step1: Create a Thread Subclass

- Create a class that extends 'Thread' class.
- Override the "run()" method to tell the task which thread has to do.

```
public class MultiThreadingLearning extends Thread{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Step2: Initiate and Start the thread

- Create an instance of the subclass.
- Call the start() method to begin the execution.

```
public class Main {
    public static void main(String args){

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultiThreadingLearning myThread = new MultiThreadingLearning();
        myThread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

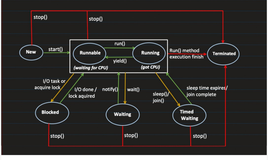
Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```

Why we have 2 ways to create threads?

- A class can implement more than 1 interface
  but
- A class can extend only 1 class.

Thread Lifecycle



Lifecycle State	Description
New	• Thread has been created but not started • Its just an Object in memory
Runnable	• Thread is ready to run. • Waiting for CPU time.
Running	• When thread start executing its code.
Blocked	Different scenarios where runnable thread goes into the Blocking state: - I/O : like reading from a file or database. - Lock aquired: if thread want to lock on a resource which is locked by other thread, it has to wait. • Releases all the MONITOR LOCKS
Waiting	• Thread goes into this state when we call the wait() method, makes it non runnable. • Its goes back to runnable, once we call notify() or notifyAll() method. • Releases all the MONITOR LOCKS
Timed Waiting	• Thread waits for specific period of time and comes back to runnable state, after specific conditions met. like sleep(), join(). • Do not Releases any MONITOR LOCKS
Terminated	• Life of thread is completed, it can not be started back again.

Before seeing an example, lets first understand "MONITOR LOCK"

MONITOR LOCK:
It helps to make sure that only 1 thread goes inside the particular section of code (a synchronized block or method)

```
public class MyCounterLockExample {

    public synchronized void test1() {
        try {
            Thread.sleep(1);
            System.out.println("inside test1");
            Thread.sleep(100);
        } catch (Exception e) {
            //exception handling here
        }
    }

    public void test2() {
        System.out.println("test2, but outside synchronized");
        synchronized (this) {
            System.out.println("test2, inside synchronized");
        }
    }

    public void test3() {
        System.out.println("test3");
    }
}
```

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

    MyCounterLockExample obj = new MyCounterLockExample();

    Thread t1 = new Thread(() -> obj.test1());
    Thread t2 = new Thread(() -> obj.test2());
    Thread t3 = new Thread(() -> obj.test3());

    t1.start();
    t2.start();
    t3.start();
}
```

How lets see an Example

```
public class ProducerConsumerExample {

    private static int[] buffer = new int[5];
    private static int count = 0;
    private static int in = 0;
    private static int out = 0;

    public static void main(String[] args) {
        Thread producerThread = new Thread(() -> {
            try {
                while (true) {
                    synchronized (buffer) {
                        while (count == buffer.length) {
                            System.out.println("Buffer is full, producer waiting...");
                            buffer.wait();
                        }
                        int item = (int) (Math.random() * 100);
                        buffer[in] = item;
                        in = (in + 1) % buffer.length;
                        count++;
                        System.out.println("Produced: " + item);
                        buffer.notifyAll();
                    }
                    Thread.sleep(500);
                }
                } catch (InterruptedException e) {
                e.printStackTrace();
                }
            });

        Thread consumerThread = new Thread(() -> {
            try {
                while (true) {
                    synchronized (buffer) {
                        while (count == 0) {
                            System.out.println("Buffer is empty, consumer waiting...");
                            buffer.wait();
                        }
                        int item = buffer[out];
                        out = (out + 1) % buffer.length;
                        count--;
                        System.out.println("Consumed: " + item);
                        buffer.notifyAll();
                    }
                    Thread.sleep(500);
                }
                } catch (InterruptedException e) {
                e.printStackTrace();
                }
            });

        producerThread.start();
        consumerThread.start();
    }
}
```

```
public class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        try {
            int temp = balance;
            Thread.sleep(100);
            balance = temp + amount;
            System.out.println("Deposited: " + amount + ", Balance: " + balance);
            notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized void withdraw(int amount) {
        try {
            while (balance < amount) {
                System.out.println("Insufficient funds, waiting...");
                wait();
            }
            int temp = balance;
            Thread.sleep(100);
            balance = temp - amount;
            System.out.println("Withdrawn: " + amount + ", Balance: " + balance);
            notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized int getBalance() {
        return balance;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread t1 = new Thread(() -> {
            try {
                account.deposit(100);
                account.withdraw(50);
                } catch (InterruptedException e) {
                e.printStackTrace();
                }
            });

        Thread t2 = new Thread(() -> {
            try {
                account.withdraw(200);
                account.deposit(150);
                } catch (InterruptedException e) {
                e.printStackTrace();
                }
            });

        t1.start();
        t2.start();
        System.out.println("Final balance: " + account.getBalance());
    }
}
```

```
Deposited: 100, Balance: 100
Withdrawn: 50, Balance: 50
Insufficient funds, waiting...
Deposited: 150, Balance: 200
Final balance: 200
```

Or use lambda expression, instead of creating Produce() and Consume() set class

```
Thread producerThread = new Thread(() -> {
    try {
        produce();
        } catch (InterruptedException e) {
        e.printStackTrace();
        }
    });
```

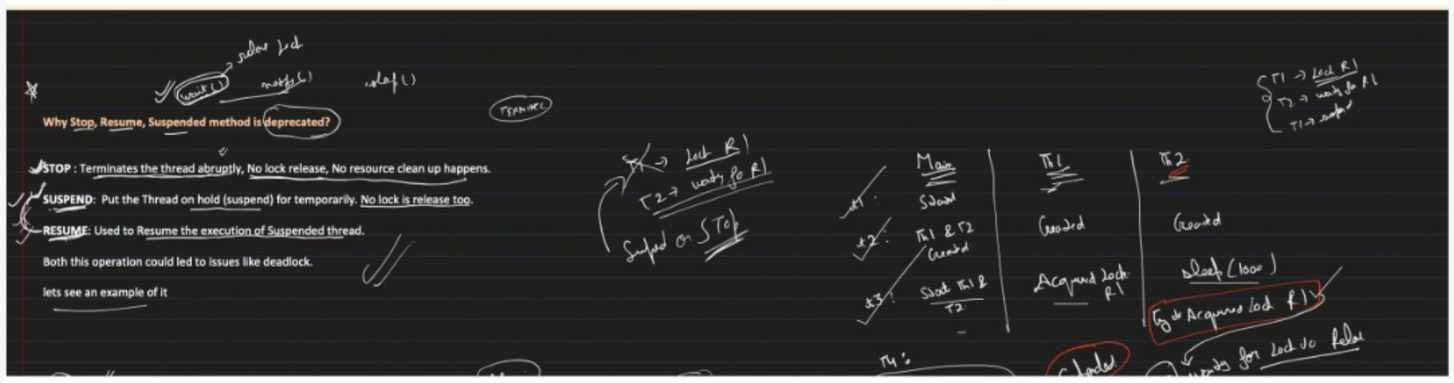
Assignment: Implement PRODUCER CONSUMER Problem

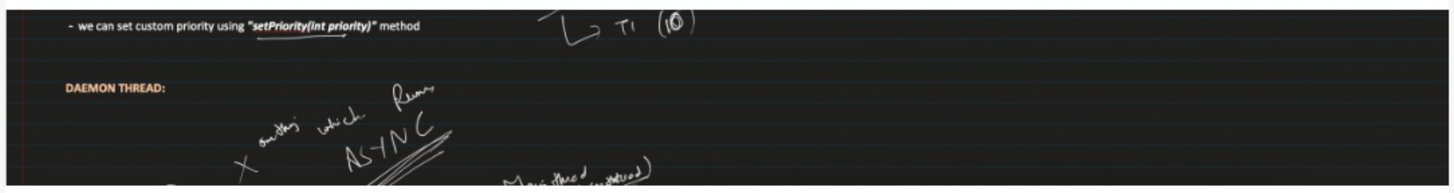
Question:

Two threads, a producer and a consumer, share a common, fixed-size buffer as a queue.
The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer.
The problem is to make sure that the producer won't produce data if the buffer is full, and the consumer won't consume data if the buffer is empty.

Java:Thread Joining, Daemon Thread, Thread Priority

"Concept && Coding" YT Video Notes





Report Abuse

Java: Lock and Condition (Multithreading Part4)

"Concept && Coding" YT Video Notes

ReentrantLock:

```java
public class Main {

    public static void main(String args[]) {

        SharedResource resource = new SharedResource();
        Thread th1 = new Thread(() -> {
            resource.producer();
        });

        Thread th2 = new Thread(() -> {
            resource.producer();
        });

        th1.start();
        th2.start();
    }
}
```

```java
public class SharedResource {

    boolean isAvailable = false;
    ReentrantLock lock = new ReentrantLock();

    public void producer(){

        try {
            lock.lock();
            System.out.println("Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = true;
            Thread.sleep( millis: 4000);
        }
        catch (Exception e) {

        }
    finally{
            lock.unlock();
            System.out.println("Lock release by: " + Thread.currentThread().getName());
        }
    }
}
```

ReadWriteLock:

ReadLock: More than 1 thread can acquire the read lock
WriteLock: Only 1 thread can acquire the write lock.

```java
public class Main {

    public static void main(String args[]) {

        SharedResource resource = new SharedResource();
        ReadWriteLock lock = new ReentrantReadWriteLock();

        Thread th1 = new Thread(() -> {
            resource.producer(lock);
        });

        Thread th2 = new Thread(() -> {
            resource.producer(lock);
        });

        SharedResource resource1 = new SharedResource();
        Thread th3 = new Thread(() -> {
            resource1.consume(lock);
        });

        th1.start();
        th2.start();
        th3.start();
    }
}
```

```java
public class SharedResource {
    boolean isAvailable = false;
    public void producer(ReadWriteLock lock){
        try {
            lock.readLock().lock();
            System.out.println("Read Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = true;
            Thread.sleep( millis: 8000);
        }
        catch (Exception e) {
        }
        finally{
            lock.readLock().unlock();
            System.out.println("Read Lock release by: " + Thread.currentThread().getName());
        }
    }
    public void consume(ReadWriteLock lock){
        try {
            lock.writeLock().lock();
            System.out.println("Write Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = false;
        }
        catch (Exception e) {
        }
        finally{
            lock.writeLock().unlock();
            System.out.println("Write Lock release by: " + Thread.currentThread().getName());
        }
    }
}
```

StampedLock:

1. Support Read/Write Lock functionality like ReadWriteLock

```java
public class Main {

    public static void main(String args[]) {

        SharedResource resource = new SharedResource();

        Thread th1 = new Thread(() -> {
            resource.producer();
        });

        Thread th2 = new Thread(() -> {
            resource.producer();
        });

        Thread th3 = new Thread(() -> {
            resource.consume();
        });

        th1.start();
        th2.start();
        th3.start();
    }
}
```

```java
public class SharedResource {
    boolean isAvailable = false;
    StampedLock lock = new StampedLock();

    public void producer(){
        long stamp = lock.readLock();
        try {
            System.out.println("Read Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = true;
            Thread.sleep( millis: 6000);
        }
        catch (Exception e) {
        }
    finally{
            lock.unlockRead(stamp);
            System.out.println("Read Lock release by: " + Thread.currentThread().getName());
        }
    }

    public void consume(){
        long stamp = lock.writeLock();
        try {
            System.out.println("Write Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = false;
        }
        catch (Exception e) {
        }
        finally{
            lock.unlockWrite(stamp);
            System.out.println("Write Lock release by: " + Thread.currentThread().getName());
        }
    }
}
```

2. Support Optimistic Lock functionality too

```java
public class Main {

    public static void main(String args[]) {

        SharedResource resource = new SharedResource();

        Thread th1 = new Thread(() -> {
            resource.producer();
        });

        Thread th2 = new Thread(() -> {
            resource.consumer();
        });

        th1.start();
        th2.start();
    }
}
```

```java
public class SharedResource {
    int a = 10;
    StampedLock lock = new StampedLock();
    public void producer(){
        long stamp = lock.tryOptimisticRead();
        try {
            System.out.println("taken optimistic lock");
            a = 11;
            Thread.sleep( millis: 6000);
            if(lock.validate(stamp)){
                System.out.println("updated a value successfully");
            }
            else {
                System.out.println("rollback of work");
                a = 10; //rollback
            }
        }
        catch (Exception e) {
        }
    }

    public void consumer(){
        long stamp = lock.writeLock();
        System.out.println("write lock acquired by : " + Thread.currentThread().getName());

        try {
            System.out.println("performing work");
            a = 9;
        }
        finally {
            lock.unlockWrite(stamp);
            System.out.println("write lock released by : " + Thread.currentThread().getName());
        }
    }
}
```

Semaphore Lock:

```java
public class Main {

    public static void main(String args[]) {

        SharedResource resource = new SharedResource();
        Thread th1 = new Thread(() -> {
            resource.producer();
        });

        Thread th2 = new Thread(() -> {
            resource.producer();
        });

        Thread th3 = new Thread(() -> {
            resource.producer();
        });

        Thread th4 = new Thread(() -> {
            resource.producer();
        });

        th1.start();
        th2.start();
        th3.start();
        th4.start();

    }
}
```

```java
public class SharedResource {

    boolean isAvailable = false;
    Semaphore lock = new Semaphore( permits: 2);

    public void producer(){

        try {
            lock.acquire();
            System.out.println("Lock acquired by: "  + Thread.currentThread().getName());
            isAvailable = true;
            Thread.sleep( millis: 4000);
        }
        catch (Exception e) {

        }
    finally{
            lock.release();
            System.out.println("Lock release by: " + Thread.currentThread().getName());
        }
    }
}
```

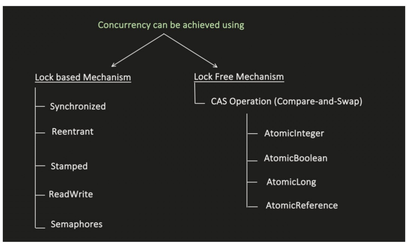
Condition

```
await() = wait()
signal() = notify()
```

Report Abuse

Lock Free Concurrency (CAS) : Multithreading Part5

"Concept && Coding" YT Video Notes



Report Abuse

Lock Free Mechanism

It uses CAS (compare and Swap) technique:

- It's a Low level operation.
- Its Atomic.
- And all modern Processor supports it.

It involves 3 main parameters:

- Memory location: location where variable is stored.
- Expected Value: value which should be present at the memory.
  - *ABA* problem is solved using version or timestamp.
- New Value: value to be written to memory, if the current value matches the expected value.

Atomic Variables:

What ATOMIC means:

- It means Single or "all or nothing"

```java
public class Main {

    public static void main(String[] args) {

        SharedResource resource = new SharedResource();
        for(int i=0; i<400; i++) {
            resource.increment();
        }
        System.out.println(resource.get());
    }
}
```

```java
public class SharedResource {

    2 usages
    int counter;

    no usages
    public void increment() {
        counter++;
    }

    no usages
    public int get() {
        return counter;
    }
}
```

Output:

```
400

Process finished with exit code 0
```

```java
public class Main {

    public static void main(String[] args) {

        SharedResource resource = new SharedResource();

        Thread t1 = new Thread(() -> {

            for(int i=0; i< 200; i++) {
                resource.increment();
            }
        });

        Thread t2 = new Thread(() -> {

            for(int i=0; i< 200; i++) {
                resource.increment();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (Exception e) {

        }

        System.out.println(resource.get());
    }
}
```

```java
public class SharedResource {

    2 usages
    int counter;

    no usages
    public void increment() {
        counter++;
    }

    no usages
    public int get() {
        return counter;
    }
}
```

Output:

```
371

Process finished with exit code 0
```

2 solutions:

1. Using lock like synchronized
2. Using lock free operation like AtomicInteger

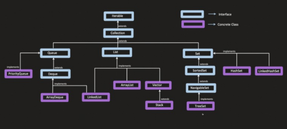
1. Using lock like synchronized

```java
public class SharedResource {

    2 usages
    int counter;

    2 usages
    public synchronized void increment() {
        counter++;
    }

    1 usage
    public int get() {
        return counter;
    }
}
```

2. Using lock free operation like AtomicInteger

```java
public class SharedResource {

    2 usages
    AtomicInteger counter = new AtomicInteger( initialValue: 0);

    2 usages
    public void increment() {
        counter.incrementAndGet();
    }

    1 usage
    public int get() {
        return counter.get();
    }
}
```

Concurrent Collection



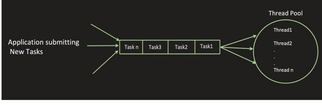
Collection	Concurrent Collection	Lock
PriorityQueue	PriorityBlockingQueue	ReentrantLock
LinkedList	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayDeque	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayList	CopyOnWriteArrayList	ReentrantLock
HashSet	newKeySet method inside ConcurrentHashMap	Synchronized
TreeSet	Collections.synchronizedSortedSet	Synchronized
LinkedHashSet	Collections.synchronizedSet	Synchronized
Queue Interface	ConcurrentLinkedQueue	Compare-and-swap operation

Thread Pools and ThreadPoolExecutor

"Concept && Coding" YT Video Notes

What is ThreadPool:

- It's a collection of threads (aka workers), which are available to perform the submitted tasks.
- Once task completed, worker thread get back to Thread Pool and wait for new task to assigned.
- Means threads can be reused.



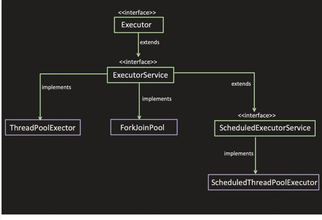
What's the Advantage of Thread Pool?

- **Thread Creation time can be saved:**
  - When each thread created, space is allocated to it (stack, heap, program counter etc..) and this takes time.
  - With thread, this can be avoided by reusing the thread.

- **Overhead of managing the Thread lifecycle can be removed:**
  - Thread has different state like Running, Waiting, terminate etc. And managing thread state includes complexity.
  - Thread pool abstract away this management.

- **Increased the performance:**
  - More threads means, more **Context Switching** time, using control over thread creation, excess context switching can be avoided.

package java.util.concurrent;



ThreadPoolExector:

It's helps to create a customizable ThreadPool.

```
public ThreadPoolExecutor
        ( int corePoolSize
          int maximumPoolSize,
          long keepAliveTime,
          TimeUnit unit,
          BlockingQueue<Runnable> workQueue,
          ThreadFactory threadFactory,
        RejectedExecutionHandler handler)
```

- ***corePoolSize***:
  Number of threads are initially created and keep in the pool, even if they are idle.

- **allowCoreThreadTimeOut**:
  If this property is set to TRUE (by default its FLASE), idle thread kept Alive till time specified by 'KeepAliveTime'.

- **KeepAliveTime**:
  Thread, which are idle get terminated after this time.

- **maxPoolSize**:
  Maximum number of thread allowed in a pool.
  If no. of thread are == *corePoolSize* and queue is also full, then new threads are created (till its less than 'maxPoolSize').

Excess thread, will remain in pool, this pool is not shutdown or if *allowCoreThreadTimeOut* is set to true, then excess thread get terminated after remain idle for *KeepAliveTime*.

- **TimeUnit**:
  TimeUnit for the keepAliveTime, whether ***Millisecond*** or ***Second*** or ***Hours*** etc.

- **BlockingQueue**:
  Queue used to hold task, before they got picked by the worker thread.
  - **Bounded Queue:** Queue with FIXED capacity.
    - Like: *ArrayBlockingQueue*
  - **Unbounded Queue:** Queue with NO FIXED capacity.
    - Like: *LinkedBlockingQueue*

- **ThreadFactory**:
  Factory for creating new thread. ThreadPoolExecutor use this to create new thread, this Factory provide us an interface to:

- To give custom Thread name
- To give custom Thread priority
- To set Thread Daemon flag etc.

- **RejectedExecutionHandler**:
  Handler for tasks that can not be accepted by thread pool.
  Generally logging logic can be put here. For debugging purpose.
  - **new ThreadPoolExecutor.AbortPolicy**
    - Throws *RejectedExecutionException*
  - **new ThreadPoolExecutor.CallerRunsPolicy**
    - Executed the rejected task in the caller thread (thread that attempted to submit the task)
  - **new ThreadPoolExecutor.DiscardPolicy**
    - Silently discard the Rejected task, without throwing any exception.
  - **new ThreadPoolExecutor.DiscardOldestPolicy**
    - Discard the oldest task in the queue, to accommodate new task.

Example:

```java
public class Main {

    public static void main(String args[]) {


        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(corePoolSize: 2, maximumPoolSize: 5, keepAliveTime:
                TimeUnit.HOURS, new ArrayBlockingQueue<>(capacity: 10), new CustomThreadFactor(),
                new CustomRejectedHandler());

        poolExecutor.allowCoreThreadTimeOut(value: true);

        //submit task
        for(int i=1; i< 25 ; i++){
            poolExecutor.submit(() -> {
                try {
                    Thread.sleep(5000L 5000);
                    System.out.println("Thread name:" + Thread.currentThread().getName());
                }catch (Exception e) {
                }
            });
        }

        poolExecutor.shutdown();

    }
}


class CustomRejectedHandler implements RejectedExecutionHandler{

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        //logging
        System.out.println("Task denied:" + r.toString());
    }
}

class CustomThreadFactor implements ThreadFactory{

    @Override
    public Thread newThread(Runnable r) {

        Thread th = new Thread(r);
        return th;
    }
}
```

Interview question:

Why you have taken corePoolSize as 2, why not 10 or 15 or another number, what's the logic?

My Answer:

Generally, the ThreadPool min and max size are depend on various factors like:
- CPU Cores
- JVM Memory
- Task Nature (CPU Intensive or I/O Intensive)
- Concurrency Requirement (Want high or medium or low concurrency)
- Memory Required to process a request
- Throughput etc.

And its an iterative process to update the min and max values based on monitoring.

Formula to find the no. of thread:

Max No of thread = No. of CUP Core * (1 + Request waiting time/processing time)

No. of CUP Core = 4
Request waiting time = 10ms
Processing time = 100ms

But this formula, do not consider Memory yet, which need to be consider...

Future, CompletableFuture and Callable

"Concept && Coding" YT Video Notes

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

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<Runn
        Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

//the thread will be created and it will perform the task
poolExecutor.execute(() -> {
    System.out.println("this is the task, which thread will execute");
});

//main thread will continue processing
}
```

New thread will be created here: lets say thread1

Now, what if caller want to know the status of the thread1. Whether its completed or failed etc.

Future:

- Interface which Represents the result of the Async task.
- Means, it allow you to check if:
  - Computation is complete
    - Get the result
    - Take care of exception if any etc.

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

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<Run
        Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

//the thread will be created and it will perform the task
Future<?> futureObj = poolExecutor.submit(() -> {
    System.out.println("this is the task, which thread will execute");
});

//after it executes the status of the thread it created
System.out.println(futureObj.isDone());
}
```

S.No.	Method Available in Future Interface	Purpose
1.	boolean cancel(boolean mayInterruptIfRunning)	• Attempts to cancel the execution of the task. • Returns false, if task can not be cancelled (typically bcze task already completed); returns true otherwise.
2.	boolean isCancelled()	• Returns true, if task was cancelled before it get completed.
3.	boolean isDone()	• Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return true.
* 4.	V get()	• Wait if required, for the completion of the task. • After task completed, retrieve the result if available.
5.	V get(long timeout, TimeUnit unit)	• Wait if required, for at most the given timeout period. • Throws 'TimeoutException' if timeout period finished and task is not yet completed.

Example:

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

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<Run
        Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    Future<?> futureObj = poolExecutor.submit(() -> {
        try {
            Thread.sleep(7000);
        } catch (InterruptedException e) {
        }
    });

    System.out.println("Is Done: " + futureObj.isDone());
    try {
        futureObj.get(1, TimeUnit.SECONDS);
    } catch (InterruptedException | ExecutionException | TimeoutException e) {
        System.out.println("Interruption happened");
    }
    catch (Exception e) {
    }

    try {
        futureObj.get();
    } catch (Exception e) {
    }

    System.out.println("Is Done: " + futureObj.isDone());
    System.out.println("Is Cancelled: " + futureObj.isCancelled());
}
```

Callable:

- Callable represents the task which need to be executed just like Runnable.
- But difference is:
  - Runnable do not have any Return type.
  - Callable has the capability to return the value.

Runnable Interface:	Callable Interface:

@FunctionalInterface
public interface Runnable {
public abstract void run();
}
	@FunctionalInterface
	public interface Callable<V> {
	V call() throws Exception;
	}

Example:

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

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(corePoolSize: 1, maximumPoolSize: 1, keepAliveTime: 1, TimeUnit.HOURS,
            new ArrayBlockingQueue<Runnable>(capacity: 10), Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    //Runnable
    Future<?> futureObject = poolExecutor.submit(() -> {
        System.out.println("hello with Runnable");
    });

    try {
        Object object = futureObject.get();
        System.out.println(object == null);
    } catch (InterruptedException e) {
    }

    //Callable1
    List<Integer> output = new ArrayList<>();
    Future<List<Integer>> futureObject1 = poolExecutor.submit(() -> {
        output.add(200);
        System.out.println("hello with Runnable and Return object");
        return output;
    });

    try {
        List<Integer> outputFromFutureObject1 = futureObject1.get();
        System.out.println(outputFromFutureObject1.get(0));
    } catch (InterruptedException e) {
    }

    //Callable2
    Future<List<String>> futureObject2 = poolExecutor.submit(() -> {
        System.out.println("hello with Callable");
        List<String> listObj = new ArrayList<>();
        listObj.add("200);
        return listObj;
    });

    try {
        List<String> outputFromFutureObject2 = futureObject2.get();
        System.out.println(outputFromFutureObject2.get(0));
    } catch (InterruptedException e) {
    }

    //Callable3
    Future<List<String>> futureObject3 = poolExecutor.submit(() -> {
        System.out.println("hello with Callable");
        List<String> listObj = new ArrayList<>();
        listObj.add("200);
        return listObj;
    });

    try {
        List<String> outputFromFutureObject3 = futureObject3.get();
        System.out.println(outputFromFutureObject3.get(0));
    } catch (InterruptedException e) {
    }
}
```

CompletableFuture:

- Introduced in Java8
- To help in async programming.
- We can considered it as an advanced version of Future provides additional capability like chaining.

How to use this:

1. CompletableFuture.supplyAsync:

   public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier)

   public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier, Executor executor)

   - *supplyAsync* method initiates an Async operation.
   - 'supplier' is executed asynchronously in a separate thread.
   - If we want more control on Threads, we can pass Executor in the method.
   - By default its uses, shared **Fork-Join Pool** executor. it dynamically adjust its pool size based on processors.

```
public class CompletableFutureExample {

    public static void main(String args[]) {

        try {
            ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(corePoolSize: 1, maximumPoolSize: 1,
                    TimeUnit.HOURS, new ArrayBlockingQueue<Runnable>(capacity: 10),
                    Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

            CompletableFuture<String> asyncTask = CompletableFuture.supplyAsync(() -> {
                //after this thread need to be executed by thread
                return "task completed";
            }, poolExecutor);

            System.out.println(asyncTask.get());

        } catch (InterruptedException e) {

        }
    }
}
```

2. *thenApply* & *thenApplyAsync*:

   - Apply a function to the result of previous Async computation.
   - Return a new *CompletableFuture* object.

```
CompletableFuture<String> asyncTask2 = CompletableFuture.supplyAsync(() -> {
    //time which thread need to execute
    return "Concept and";
}, poolExecutor).thenApply(s1->{String val2 = {
    //functionality which can work on the result of previous async task
    return val2 + "Coding";
});
```

'thenApply' method:

- Its a Synchronous execution.
- Means, it uses same thread which completed the previous Async task.

'thenApplyAsync' method:

- Its a Asynchronous execution.
- Means, it uses different thread (*from fork-join' pool, if we do not provide the executor in the method*), to complete this function.
- If Multiple '*thenApplyAsync*' is used, ordering can not be guarantee, they will run concurrently.

```
public class ThenAndThenAsyncDemo {

    public static void main(String[] args) {

        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(corePoolSize: 1, maximumPoolSize: 1, keepAliveTime: 1,
                TimeUnit.HOURS, new ArrayBlockingQueue<Runnable>(capacity: 10),
                Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

        ThreadPoolExecutor.execute(() -> {
            System.out.println("Thread Name which runs 'execute': " + Thread.currentThread().getName());
            System.out.println("Thread Name which runs 'execute': " + Thread.currentThread().getName());
        });
        CompletableFuture<String> completableFuture = poolExecutor.submit(() -> {
            System.out.println("Thread Name which runs 'submit': " + Thread.currentThread().getName());
            return "Hello";
        });
        System.out.println("Thread Name which runs 'CompletableFuture': " + Thread.currentThread().getName());
        System.out.println("Thread Name which runs 'thenApply': " + completableFuture.thenApply(s -> {
            System.out.println("Thread Name which runs 'thenApply': " + Thread.currentThread().getName());
            return s + " World";
        });
        System.out.println("Thread Name which runs 'thenApplyAsync': " + completableFuture.thenApplyAsync(s -> {
            System.out.println("Thread Name which runs 'thenApplyAsync
```

Java - Fork/Join Pool, Single, Fixed, CachedPool

"Concept && Coding" YT Video Notes

Fixed ThreadPoolExecutor:

'newFixedThreadPool' method creates a thread pool executor with a fixed no. of threads.

Min and Max Pool	Same
Queue Size	Unbounded Queue
Thread Alive when idle	Yes
When to use	Exact Info, how many Async task is needed
Disadvantage	Not good when workload is heavy, as it will lead to limited concurrency

```
//fixed thread pool executor
ExecutorService poolExecutor1 = Executors.newFixedThreadPool( nThreads: 5);
poolExecutor1.submit(() -> "this is the async task");
```

Report Abuse

Cached ThreadPoolExecutor:

'newCachedThreadPool' method creates a thread pool that creates a new thread as Needed (dynamically).

Min and Max Pool	Min : 0 Max: Integer.MAX_VALUE
Queue Size	Blocking Queue with Size 0
Thread Alive when idle	60 SECONDS
When to use	Good for handling burst of short lived tasks.
Disadvantage	Many long lived tasks and submitted rapidly, ThreadPool can create so many threads which might lead to increase memory usage.

```
//cached thread pool executor
ExecutorService poolExecutor = Executors.newCachedThreadPool();
poolExecutor.submit(() -> "this is the async task");
```

Single Thread Executor:

'newSingleThreadExecutor' creates Executor with just single Worker thread.

Min and Max Pool	Min : 1 Max: 1
Queue Size	Unblocking Queue
Thread Alive when idle	Yes
When to use	When need to process tasks sequentially
Disadvantage	No Concurrency at all

WorkStealing Pool Executor:

- It creates a *Fork-Join Pool Executor*.

- Number of threads depends upon the Available Processors or we can specify in the parameter

- There are 2 queues:
  - Submission Queue
  - Work-Stealing Queue for each thread (it's a Dequeue)

- Steps:
  - If all threads are busy, task would be placed in "Submission Queue". (or whenever we call submit() method, tasks goes into submission queue only)
  - Lets say task1 picked by ThreadA. And if 2 subtasks created using fork() method. Subtask1 will be executed by ThreadA only and Subtask2 is put into the ThreadA work-stealing queue.
  - If any other thread becomes free, and there is no task in Submission queue, it can "STEAL" the task from the other thread work-stealing queue.

- Task can be split into multiple small sub-tasks. For that Task should extend:
  - RecursiveTask
  - RecursiveAction

- We can create Fork-Join Pool using "newWorkStealingPool" method in ExecutorService.
  Or
  By calling ForkJoinPool.commonPool() method.

```
public class ExecutorsUtilityExample {

    public static void main(String args[]) {


        ForkJoinPool pool = ForkJoinPool.commonPool();
        Future<Integer> futureObj = pool.submit(new ComputeSumTask(0, 100));
        try {
            System.out.println(futureObj.get());
        }catch (Exception e){
        }
    }
}


class ComputeSumTask extends RecursiveTask<Integer> {

    int start;
    int end;
    ComputeSumTask(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {

        if (end - start <= 4) {
            int totalSum = 0;
            for (int i = start; i <= end; i++) {
                totalSum += i;
            }
            return totalSum;
        } else {
            //split the task
            int mid = (start + end) / 2;
            ComputeSumTask leftTask = new ComputeSumTask(start, mid);
            ComputeSumTask rightTask = new ComputeSumTask(mid + 1, end);

            // Fork the subtasks for parallel execution;
            leftTask.fork();
            rightTask.fork();

            // Combine the results of subtasks
                int leftResult = leftTask.join();
                int rightResult = rightTask.join();

            // Combine the results
            return leftResult + rightResult;
        }
    }
}
```

Java : ScheduledThreadPoolExecutor

"Concept && Coding" YT Video Notes

shutdown vs await Termination vs shutdownNow

Shutdown:

- Initiates orderly shutdown of the ExecutorService.
- After calling 'Shutdown', Executor will not accept new task submission.
- Already Submitted tasks, will continue to execute.

AwaitTermination:

- Its an Optional functionality. Return true/false.
- It is used after calling 'Shutdown' method.
- Blocks calling thread for specific timeout period, and wait for ExecutorService shutdown.
- Return true, if ExecutorService gets shutdown withing specific timeout else false.

Report Abuse

shutdownNow:

- Best effort attempt to stop/interrupt the actively executing tasks
- Halt the processing of tasks which are waiting
- Return the list of tasks which are awaiting execution.

Scenario1: Task submission after Shutdown

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

    ExecutorService poolObj = Executors.newFixedThreadPool( nThreads: 5);
    poolObj.submit(() -> {
        System.out.println("Thread going to start its work");
    });

    poolObj.shutdown();

    poolObj.submit(() -> {
        System.out.println("Thread going to start its work");
    });

}
```

```
Exception in thread "main" java.util.concurrent.RejectedExecutionException Create breakpoint : Task java.util.concurrent.FutureTask@4b409385 rejected
    at java.util.concurrent.AbstractExecutorService.submit(AbstractExecutorService.java:112)
```

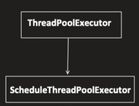
Scenario2: Shutdown do not impact the already submitted task

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

    ExecutorService poolExecutorObj = Executors.newFixedThreadPool( nThreads: 5);
    poolExecutorObj.submit(() -> {
        try {
            Thread.sleep( millis: 5000);
        }catch (Exception e){

        }
        System.out.println("new task");
    });

    poolExecutorObj.shutdown();
    System.out.println("Main thread unblocked and finished processing");
}
```

Scenario3: usage of 'awaitTermination'

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

    ExecutorService poolExecutorObj = Executors.newFixedThreadPool( nThreads: 5);
    poolExecutorObj.submit(() -> {
        try {
            Thread.sleep( millis: 6000);
        } catch (Exception e) {

        }
        System.out.println("new task");
    });

    poolExecutorObj.shutdown();
    try {
        boolean isExecutorTerminated = poolExecutorObj.awaitTermination( timeout: 3, TimeUnit.SECONDS);
        System.out.println("Main thread, isExecutorTerminated: " + isExecutorTerminated);
    } catch (Exception e) {

    }
}
```

ScheduledThreadPoolExecutor : Helps to schedule the tasks



S.No.	Method Name	Description
1.	schedule(Runnable command, long delay, TimeUnit unit)	Schedules a Runnable task after specific delay. Only one time task runs.
2.	schedule(Callable<V> callable, long delay, TimeUnit unit)	Schedules a Callable task after specific delay. Only one time task runs.

3.	scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)
	
	`public static void main(String args[]) {`
	
	`ScheduledExecutorService scheduledExecutorService0bj = new ScheduledThreadPoolExecutor( corePoolSize: 1);`
	`Future<?> scheduledObj = scheduledExecutorService0bj.scheduleAtFixedRate(() -> {`
	`System.out.println("task going to start 1s : " + Thread.currentThread().getName());`
	
	`try{`
	`Thread.sleep( millis: 1000);`
	`}catch (Exception e){`
	
	`}`
	`System.out.println("new task");`
	`}, initialDelay: 1, period: 1, TimeUnit.SECONDS);`
	
	`try {`
	`Thread.sleep( millis: 20000);`
	`} catch (InterruptedException e) {`
	`e.printStackTrace();`
	`}`
	`scheduledObj.cancel( mayInterruptIfRunning: true);`
	`}`
		We can use cancel method to stop this repeated task.
		
		Also lets say, if thread1 is taking too much time to complete the task and next task is ready to run, till previous task will not get completed, new task can not be start (it will wait in queue).

4.	scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Schedules a Runnable task for repeated execution with a fixed delay (Means next task delay counter start only after previous one task completed)

Java: VirtualThreads and ThreadLocal

"Concept && Coding" YT Video Notes

ThreadLocal

- ThreadLocal class provide access to Thread-Local variables.
- This 'Thread-Local' variable hold the value for particular thread.
- Means each Thread has its own copy of Thread-Local variable.
- We need only 1 object of ThreadLocal class and each thread can use it to set and get its own Thread-variable variable.

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

    ThreadLocal<String> threadLocalObj = new ThreadLocal<>();

    //main thread
    threadLocalObj.set(Thread.currentThread().getName());

    Thread thread1 = new Thread( () -> {
        threadLocalObj.set(Thread.currentThread().getName());
        System.out.println("Task1");
    });

    thread1.start();

    try{
        Thread.sleep( millis: 2000);
    }catch (Exception e){

    }

    //here we have main thread
    System.out.println("Main thread: " + threadLocalObj.get());
}
```

Remember to clean up, if reusing the thread

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

    ThreadLocal<String> threadLocalObj = new ThreadLocal<>();

    ExecutorService poolObj = Executors.newFixedThreadPool( nThreads: 5);

    poolObj.submit(() -> {
        threadLocalObj.set(Thread.currentThread().getName());
    });

    for(int i=1; i<15; i++){
        poolObj.submit(() -> {
            System.out.println(threadLocalObj.get());
        });
    }
}
```

Output:

```
null
pool-1-thread-1
null
pool-1-thread-1
null
null
null
null
null
pool-1-thread-1
null
null
null
```

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

    ThreadLocal<String> threadLocalObj = new ThreadLocal<>();

    ExecutorService poolObj = Executors.newFixedThreadPool( nThreads: 5);

    poolObj.submit(() -> {
        threadLocalObj.set(Thread.currentThread().getName());
        //my work completed, now clean up
        threadLocalObj.remove();
    });

    for(int i=1; i<15; i++){
        poolObj.submit(() -> {
            System.out.println(threadLocalObj.get());
        });
    }
}
```

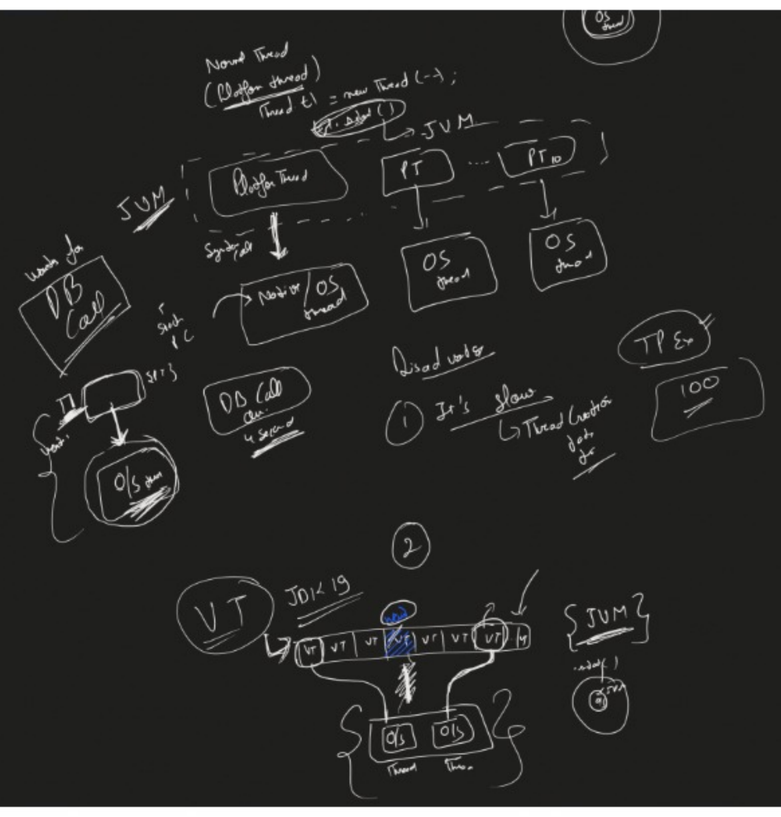
Output:

```
null
null
null
null
null
null
null
null
null
null
null
null
null
```

Virtual Thread vs Platform Thread

Moto of Virtual Thread:
To get Higher throughput not Latency

```
Thread th1 = Thread.ofVirtual().start(RunnableTask);
Or
 ExecutorService myExecutorObj = Executors.newVirtualThreadPerTaskExecutor()
myExecutorObj .submit(RunnableTask)
```



Report Abuse