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

```
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.

```
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.

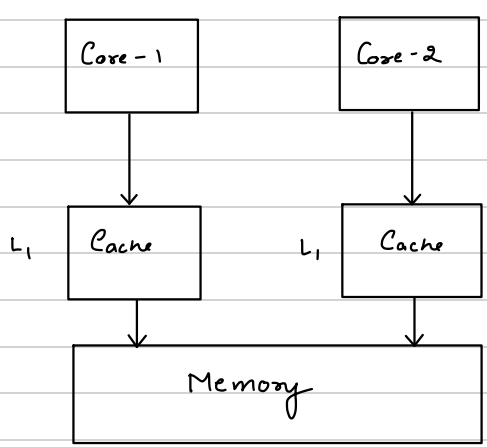
⇒ Double Checked Locking (This is used majorly)

- 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:

```
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.