- If you try to access any method of an object, but the object is null; then you'll get an **Null Pointer Exception**.
- **Remember: in java we initialize an array using {} curly braces; not [] square brackets.**
- Consider the following scenario:
  - I have 3 files:
    - Main.java
      - Contains the main method.
      - It has an object of type QuestionService
    - QuestionService.java
      - If has an array of objects.
      - Those objects are of type Question
    - Question.java
      - It is a normal class that contains some variables and getter, setter methods to build a question.
      - Like, id (int), question(String), options (String[]), answer (String)
  - Now, I implemented the **toString()** method inside Question class, so that whenever I print the object of type Question directly, it'll print something meaningful instead of the default *hashcode*.
  - Then I compiled **Main.java** and ran this. But I couldn't see the changes that I did inside **Question.java**.
  - When you compile a **.java** file, it'll generate **.class** file of all those Class which are linked to the **.java** file (in our case, all Classes i.e. Main, QuestionService, Question are linked).
  - Let I run **javac Main.java**
    - Now, it'll compile Main.java and create the Main.class file.
    - And it'll check if there is QuestionService.class and Question.class already present.
    - If present, then don't recompile those; otherwise compile those as well.
    - In my case, I had already QuestionService.class and Question.class present; so it was not re-compiling those classes.
  - So, every-time you do any changes, run the following command:
    - **javac *.java** (it'll re-compile all the .java files present in the current directory)
    - **java Main**

- To read input from users:
  - **Scanner** is used to read the input from the user.

```
Scanner sc = new Scanner(System.in); // initializing scanner to read user input
int i = 0;
for (Question q : questions) {
    System.out.println("Question No: " + q.getId());
    System.out.println(q.getQuestion());
    String[] options = q.getoptions();
    for (String option : options)
        System.out.println(" - " + option);

    this.selections[i++] = sc.nextLine(); // reading user input and storing in array
}
sc.close(); // closing scanner to prevent resource leak
```

  - Here, I created an object of Scanner, and passed **System.in**
    - **System.in** is a static input stream provided by JVM.
    - It represents standard input of your program.
    - Since it is **static final**, there is only **System.in** object per JVM process.
    - Once you close the scanner object using **sc.close()**, you can't read the input again.
  - Below is an example of wrong usage of Scanner:

```
int i = 0;
for (Question q : questions) {
    System.out.println("Question No: " + q.getId());
    System.out.println(q.getQuestion());
    String[] options = q.getoptions();
    for (String option : options)
        System.out.println(" - " + option);

    Scanner sc = new Scanner(System.in); // create a Scanner object
    this.selections[i++] = sc.nextLine(); // read user input
    sc.close(); // close the scanner to prevent resource leak
    // but closing here will close System.in, causing issues on next iteration
}
```

    - Here you'll get an exception after the first iteration, because System.in is already closed in the previous iteration.
  - **sc.close()** is optional by the way.
- **<u>Abstract Class and Abstract Method</u>**
  - In java, empty methods are valid.

```
class Car {
    // this method is valid but doesn't give error
    public void drive() {

    }
    public void playMusic() {
        System.out.println(x:"Playing music");
    }
}
```

- These are some conditions in Java OOP:
  - Abstract method inside Abstract class (✓)
  - Abstract method inside Normal class (✗)
  - Normal method inside Abstract class (✓)
  - Normal method inside Normal class (✓ (default only))
- In short:
  - Abstract method ⇒ class must be abstract
  - Normal methods ⇒ allowed anywhere
- An abstract class may have:
  - Only abstract methods
  - Only normal methods
  - A mix of abstract + normal methods
  - Even no methods at all

```java
abstract class Car {
    public abstract void drive();

    public void playMusic() {
        System.out.println(x:"Playing music");
    }
}

class Tesla extends Car {
    public void drive() {
        System.out.println(x:"Driving Tesla");
    }

    public void show() {
        System.out.println(x:"Show method in Tesla");
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Car car = new Tesla();
        car.drive();
        car.playMusic();
        // car.show(); // This will give an error because 'show' is not defined in Car

        Tesla myCar = new Tesla();
        myCar.drive();
        myCar.playMusic();
        myCar.show();
    }
}
```

- If a class is inheriting an abstract class
  - It must implements the abstract methods present inside the abstract class.
  - The normal methods present inside the abstract class need not to be overridden.

- **An abstract class can have constructor.**
  - The constructor can be called from the base classes using **super()**
- **NOTE**
  - An abstract class can inherit another abstract class as well.
  - And in this case, the child *abstract class* need not to implement the *abstract methods* inside the parent abstract class.

```
abstract class Car {
    public abstract void drive();

    public abstract void accelerate();

    public void playMusic() {
        System.out.println(x:"Playing music");
    }
}

abstract class FastCar extends Car {

    public void accelerate() {
        System.out.println(x:"Accelerating FastCar quickly");
    }
}

class Tesla extends FastCar {
    public void drive() {
        System.out.println(x:"Driving Tesla");
    }

    public void show() {
        System.out.println(x:"This is a Tesla car");
    }
}
```

  -

- **Inner Class**
  - An inner class is a class defined inside another class.
  - It is logically associated with its outer class and has access to its members (even private ones).
  - The inner class's type will be: OuterClassName.InnerClassName
  - And to instantiate the inner class, you need an instance of the outer class.
  - To instantiate the inner class, you need to call like.
    - **obj.new InnerClassName()**
  - There are 4 types of Inner Class

- Non-Static Nested Inner Class
- Static Nested Inner Class
- Local Inner Class
- Anonymous Inner Class

- **Non-Static Nested Inner Class:**

```java
class Outer {
    int age = 5;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show");
    }

    class Inner {
        // shadowing occured
        int age = 30;

        public void config() {
            System.out.println("in Inner's config: inner age = " + age); // inner's age
            System.out.println("in Inner's config: outer age = " + Outer.this.age); // outer's age
            // both are correct
            System.out.println("in Inner's config: name = " + name);
            System.out.println("in Inner's config: name = " + Outer.name);
        }
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer out = new Outer();
        Outer.Inner ob = out.new Inner();

        // Outer.Inner ob = new Outer().new Inner();
        // it is also correct; outer object would be anonymous

        ob.config();

        System.out.println(ob.age);
    }
}
```

- Just imagine a non-static method. You can access this only by an object.
- Just like that, you can access the Non-Static Inner Class using an object of Outer Class only.
- It can access all the instance and static variables of the outer class (even private variables are accessible).
- In the above example, the instance variable **age** got shadowed inside the Inner class. To access the Outer class's age
  - **OuterClassName.this.VariableName**
  - Because, **this.age** would have given InnerClass's variable **age**
- Why not **new ob1.Inner()** ?

- Think of it like, as the Inner class is non-static; so the Inner class's instance will be specific to the Outer class's instance.
- So, to instantiate Inner class's instance inside the Outer class's instance (here **ob1**), we need to call **ob1.new Inner()**

- **Static Nested Inner Class:**
  - Declared with the static keyword.
  - It does not need an instance of the outer class.
  - Can access only **static members** of the outer class directly.

```java
class Outer {
    int age;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show");
    }

    static class Inner {
        public void config() {
            // // error; as age is an instance variable
            // System.out.println("in Inner's config: age = " + age);
            // both are correct
            System.out.println("in Inner's config: name = " + name);
            System.out.println("in Inner's config: name = " + Outer.name);
        }
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer.Inner ob = new Outer.Inner();
        // new Outer().Inner() <= wrong

        ob.config();
    }
}
```

  - Just like static method, we can access the static inner class using the Outer class directly without instantiating it.
  - Here, **new Outer.Inner()** (*not* **Outer.new Inner()** *or* **new Outer().Inner()**)

- **Local Inner Class:**
  - When the Inner class is defined inside a method of Outer class, then it is Local Inner Class.

```java
class Outer {
    int age = 5;
    static String name = "Outer Static";

    private void show() {
        System.out.println(x:"In outer's show..");
    }

    public void show2() {
        System.out.println(x:"in Outer's show2..");

        class Inner {
            int val = 10;

            private void displayVal() {
                System.out.println("displayVal: val inside Inner is: " + val);
            }

            public void displayVal2() {
                System.out.println("displayVal2: val inside Inner is: " + val);
            }
        }
        // it can access both private and public methods of inner class
        Inner obj = new Inner();
        obj.displayVal();
        obj.displayVal2();
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        // ob.show(); // error as method is private
        ob.show2();
    }
}
```

- It is strange that, the **displayVal** method is private; but still it was able to get called from outside of it i.e. inside the **show()** method.
- As the Inner class comes inside the scope of Outer class, so in this case, **all private things of Outer class and Inner class are accessible to each-other**.
- But the private method **show()** of the class **Outer** is not accessible outside.
- Because Inner lives inside the scope of Outer, they can freely access each other's private members.
- But Main is outside, so it cannot access Outer.show() or Inner.displayVal().

٨ **Anonymous Inner Class:**

```java
class Test {
    public void greet() {
        System.out.println(x:"Hello from Test!");
    }
}

class Outer {
    int age = 5;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show...");

        // case-1
        class AdvTest extends Test {
            public void greet() {
                System.out.println(x:"Hello from AdvTest!");
            }
        }

        Test ob = new AdvTest();
        ob.greet();

        // case-2 (anonymous inner class)
        // --- for this case: Test can be abstract as well ---
        Test ob2 = new Test() {
            public void greet() {
                System.out.println(x:"Hello from Anonymous Test!");
            }
        }; // semicolon is necessary here
        ob2.greet();
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        ob.show();
    }
}
```

٩

```java
class Test {
    public void greet() {
        System.out.println(x:"Hello from Test!");
    }
}
class Outer {
    int age = 5;
    static String name = "Outer Static";
    public Test obj = new Test() {
        public void greet() {
            System.out.println(x:"Hello from Outer => Test!");
        }
    };
}
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        ob.obj.greet();
    }
}
```

١٠

- Its just like inheriting a Normal/Abstract class and instantiating directly without creating the inherited class.

- **Summary of Inner Classes**
  - In any type of inner class creation, both Outer and Inner classes can access each-other's private members.
  - **Non-Static Inner Class:**
    - Assumption: Inner class's name: **Inner**, Outer class's name: **Outer**
    - Just like non-static method, the Non-Static Inner Class can access both instance variables and static variables of the Outer class.
    - If there is any type of shadowing of Outer class's variable then (let variable name is: **val**)
      - **this.val** ⇒ Inner class's variable *val*
      - **Outer.this.val** ⇒ Outer class's variable *val*
    - Just like Non-Static Method, we need an instance of the class to access the Non-Static Inner Class.
    - As the Inner class's instance will be a part of the Outer class's instance, so to instantiate this:
      - **obOuter.new Inner()**
  - **Static Inner Class:**
    - Assumption: Inner class's name: **Inner**, Outer class's name: **Outer**
    - Just like the Static Methods, the Static Inner Class can only access the *static* members of the Outer class.
    - If there is any shadowing: (let the variale name is **val**)
      - val ⇒Inner class's static variable
      - Outer.val ⇒ Outer class's static variable
  - **Local Inner Class:**
    - The Inner class is defined inside a method of the Outer Class.
    - The scope to access this Inner class is only the scope of that Method.
  - **Anonymous Inner Class:**
    - Its just like extending a class (either Normal or Abstract) and creating an object out of that; without creating the Class.
    - The syntax is:
      - **ClassName** obj = new **ClassName**() { /* override method if want */ }
  -

- **Interface**
  - By default the variables inside interfaces are: **public static final**.
    - So, you need to initialize while declaring it.
    -
```
interface A {
    // by default variables are "public final static"
    // so, you need to initialize this
    int age = 23;
    String area = "Banglore";
```
    - You cannot override the variables that were declared and initialized in interface.
    -
```
public static void main() {
    System.out.println("A's area: " + A.area);
    System.out.println("A's age: " + A.age);

    // A.area = "Delhi"; // Error, because that is final
```
  - By default all the methods are **public abstract**; you don't need to explicitly write that.
    -
```
interface A {
        void show(); //same as => public abstract void show()

        void config();//same as => public abstract void config()
```
  - **implements** is the keyword that is used to implement a interface to a class.
  - Unlike classes, **multiple implementations** are allowed in case of interface.
    -
```
class B implements A, X {

    public void show() {
```
(multiple implementation)
  - Interfaces can inherit another interface.
    - In this case, **multiple inheritance** is allowed.
    -
```
interface Y extends X, A {

}
```
    - But the class which implements **Y**, has to override all the methods mentioned in interfaces **X** and **A**.
  - NOTE:
    - Interfaces cannot have constructors (because they can't be instantiated).
    - But you can create a reference of an interface type pointing to a class object.
    - Interfaces are used to achieve abstraction and multiple inheritance in Java.

```java
interface A {
    void show();
    void config();
}
interface X {
    void run();
}
class B implements A, X {
    public void show() {
        System.out.println(x:"overridden 'A: show'");
    }
    public void config() {
        System.out.println(x:"overridden 'A: config'");
    }
    public void run() {
        System.out.println(x:"overridden 'X: run'");
    }
}
public class Interface {
    public static void main() {
        A obj = new B();
        obj.show();
        obj.config();
        // obj.run(); // Error: A doesn't have run

        X obj2 = new B();
        // obj.show(); // Error: X doesn't have show
        // obj.config(); // Error: X doesn't have config
        obj2.run();
    }
}
```

- In case of implementing 2 interfaces, creating object of one interface type and calling the method mentioned in the other interface will not be possible.
- We had seen this during Upcasting and Downcasting.

➢ **Need of Interface**

- You can see the below example code.
- Here if we didn't have implemented an interface, only **Laptop** or **Desktop** type of objects would have been acceptable inside the **codeApplication** method of **Developer** class.
- Now we can think, instead of interface, **abstract class** can also be used;
- But, just to write a abstract method, why to create an abstract class.
- Interface is here simple and doing all the required things.

```java
interface Computer {
    void code();
}

class Laptop implements Computer {
    public void code() {
        System.out.println(x:"Coding started: little slow");
    }
}

class Desktop implements Computer {
    public void code() {
        System.out.println(x:"Coding started: faster");
    }
}

class Developer {
    public void codeApplication(Computer comp) {
        comp.code();
    }
}

class Company {
    Run | Debug
    public static void main(String[] args) {
        Developer alok = new Developer();
        Developer kanha = new Developer();
        // we are giving alok a laptop
        // and kanha a desktop to code
        Computer laptop = new Laptop();
        Computer desktop = new Desktop();

        alok.codeApplication(laptop);
        kanha.codeApplication(desktop);
    }
}
```

- **Enum**
  - enum is a special type of class in Java (its not same as Class; but similar).
  - It's a **final class** which cannot be inherited by any other class.

```java
enum Status {
    Running, Failed, Pending, Success;
}

public class Enum {
    Run | Debug
    public static void main(String[] args) {
        Status s = Status.Failed;
        System.out.println("s = " + s); // Failed

        // in Java, the indexing starts from 0 for the enums
        // Running: 0, Failed: 1, Pending: 2, Success: 3
        // the method "ordinal()" returns the index
        Status[] allVals = Status.values();
        for (Status val : allVals)
            System.out.println("index: " + val.ordinal() + ", value: " + val);
    }
}
```

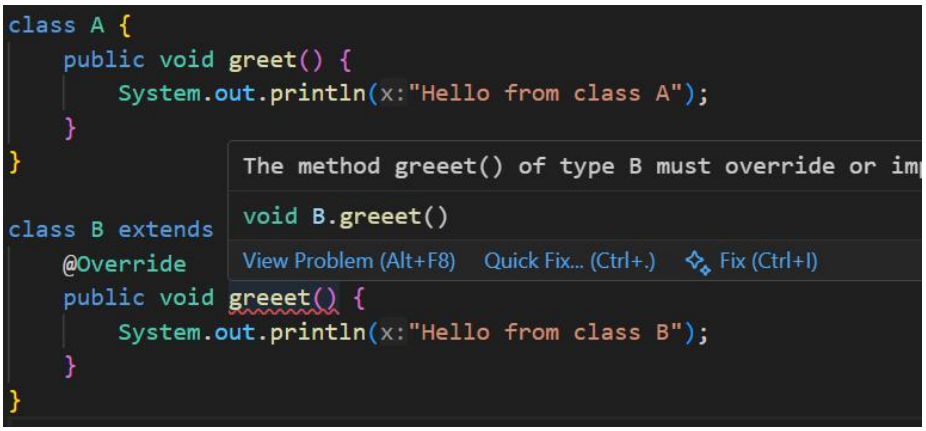
- **switch case** statement also supports **enum**, so it can be used to check the status.
- Consider the following example: (more than one constructor can be created)

```
enum Laptop {
    Macbook(price:2000), Dell(price:1200), Acer(price:1400);

    private int price;

    Laptop(int price) {
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}
```

```
final class Laptop extends Enum<Laptop> {
    public static final Laptop Macbook = new Laptop("Macbook", 0, 2000);
    public static final Laptop Dell = new Laptop("Dell", 1, 1200);
    public static final Laptop Acer = new Laptop("Acer", 2, 1400);

    private int price;

    private Laptop(String name, int ordinal, int price) {
        super(name, ordinal); // from java.lang.Enum
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}
```

- Behind the scene.

```
enum Laptop {
    // these are objects of Laptop class itself
    // as you are passing some value, so you need to create a constructor
    Macbook(price:2000), Dell(price:1200), Acer(price:1400);

    public int price;

    Laptop(int price) {
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}

public class Enum {
    Run | Debug
    public static void main(String[] args) {
        Laptop lap = Laptop.Macbook;
        System.out.println(lap.getClass()); // class Laptop
        System.out.println(lap.getClass().getSuperclass()); // class java.lang.Enum

        System.out.println("s = " + lap); // Macbook
        System.out.println("price = " + lap.getPrice()); // 2000

    }
}
```

- **Annotations**
  - Provides information to the compiler, tools, or runtime.
  - Think of it as a special marker/label you attach to classes, methods, variables, etc.
  - For example **@Override**
    - It tells the compiler: "this method is supposed to override a method from its superclass."
    - If it doesn't, the compiler will show an error.
  - 
    ```
    class A {
        public void greet() {
            System.out.println(x:"Hello from class A");
        }
    }

    class B extends A {
        public void greeet() {
            System.out.println(x:"Hello from class B");
        }
    }
    ```
    - Here you can see, I have made a *spelling error* in Class B.
    - Instead of **greet** I have written **greeet**
  - 

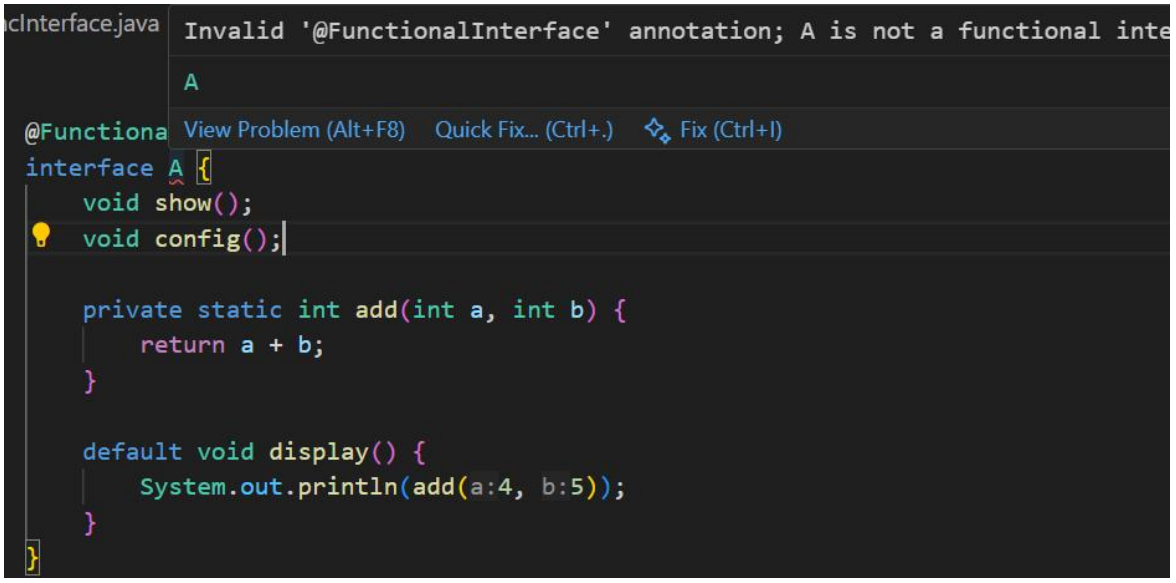
    - Now I used the annotation **@Override** , so now the compiler is showing me the error that this method doesn't exists in the superclass.
- **Types of Interface**
  - **Normal Interface**
    - Interface having **2 or more** methods
  - **Functional Interface / SAM** (Single Abstract Method)
    - Interface having only **1** method.
  - **Marker Interface**
    - Interface having **no** method.

- used for tagging or marking classes (e.g., Serializable).

- **Functional Interface:**

```java
@FunctionalInterface
interface A {
    void show();

    private static int add(int a, int b) {
        return a + b;
    }

    default void display() {
        System.out.println(add(a:4, b:5));
    }
}
```

  - Abstract method should be only **1**.
  - Remaining static or default methods can be there.
  - Annotation: **@FunctionalInterface**



  - I added one more Abstract method, so it is showing me error.

- **Lambda Expression**

```
@FunctionalInterface
interface A {
    void show();
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // anonymous inner class concept
        A obj = new A() {
            public void show() {
                System.out.println(x:"in show A");
            }
        };
        obj.show();
    }
}
```

- This code is proper and it'll work fine.

```
public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // anonymous inner class concept
        A obj = new A() {
            public void show() {
                System.out.println(x:"in show A");
            }
        };
        obj.show();

        // lambda expression
        A obj2 = () -> {
            System.out.println(x:"in show A");
        };
        obj2.show();

        // if there is only single expression
        A obj3 = () -> System.out.println(x:"in show A");
        obj3.show();
    }
}
```

```
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj3 = (int a) -> System.out.println("in show A: " + a);
        obj3.show(a:4);
    }
}
```

- You can also pass the arguments.

```java
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = (a) -> System.out.println("in show A: " + a);
        obj.show(a:4);
    }
}
```

- You don't even need to provide the data type; it'll take from the interface directly.

```java
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = a -> System.out.println("in show A: " + a);
        obj.show(a:4);
    }
}
```

- If you have only one argument, don't need to give the *parenthesis* as well.

```java
@FunctionalInterface
interface A {
    int add(int a, int b);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = (a, b) -> a + b;
        int res = obj.add(a:4, b:5);
        System.out.println("Sum = " + res);
    }
}
```

- You can directly return the values like this.

- **Lambda Expression** only works with the **Functional Interface.**
- Because if there are more than one method, which will be implemented.

- Dffd
- dfd

- Fdffdfdfdf