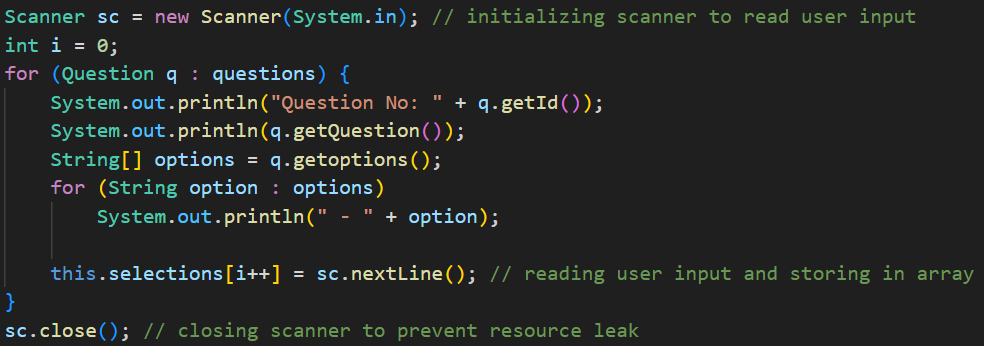
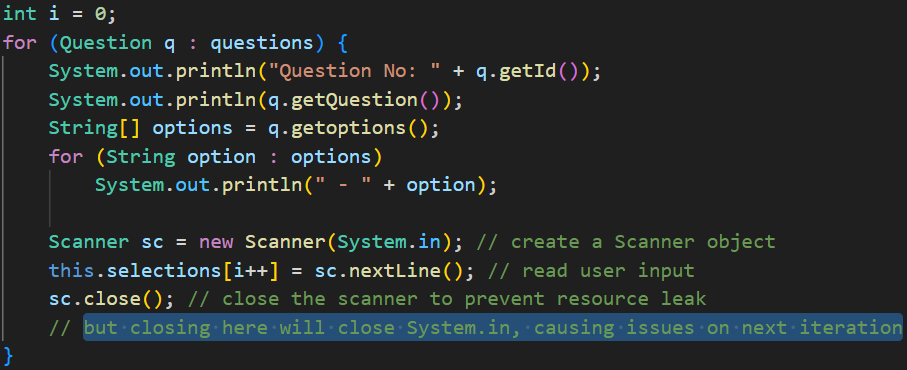
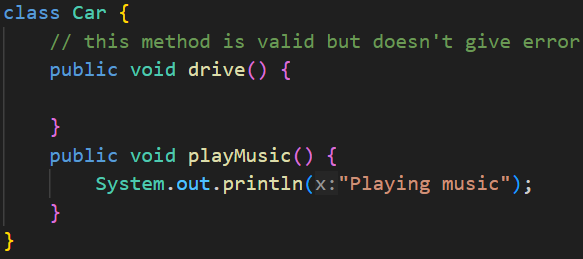
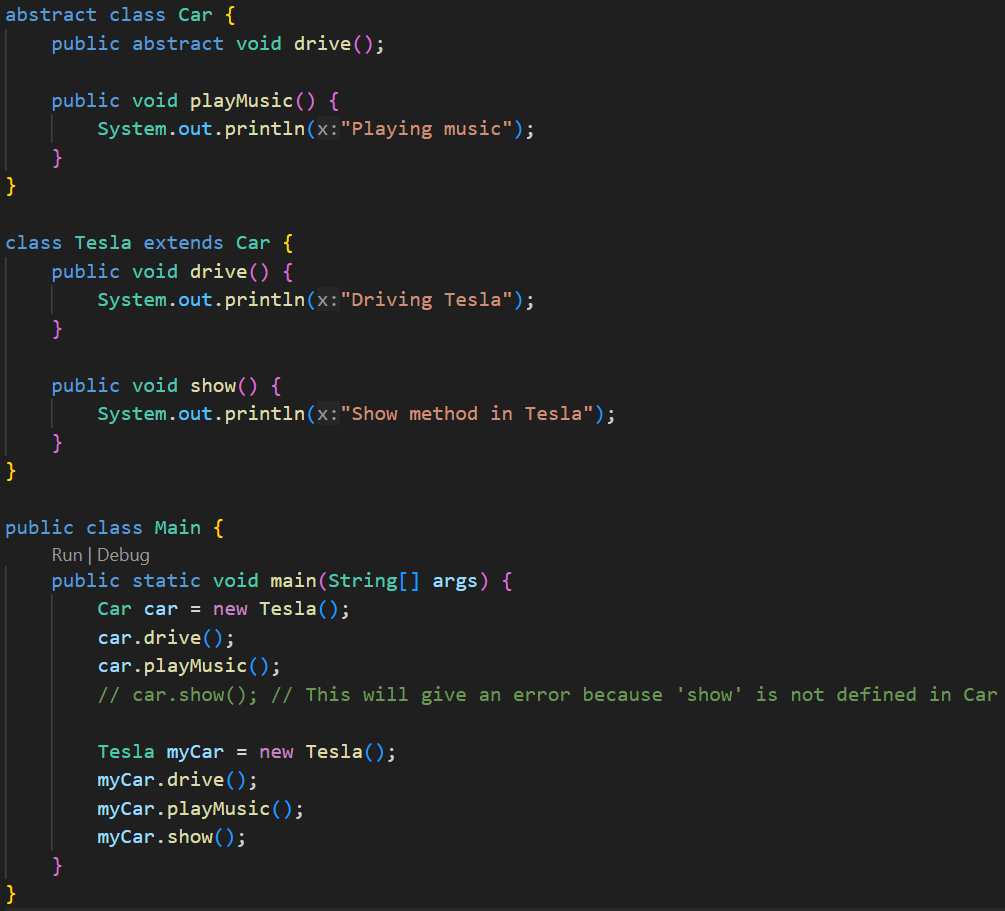
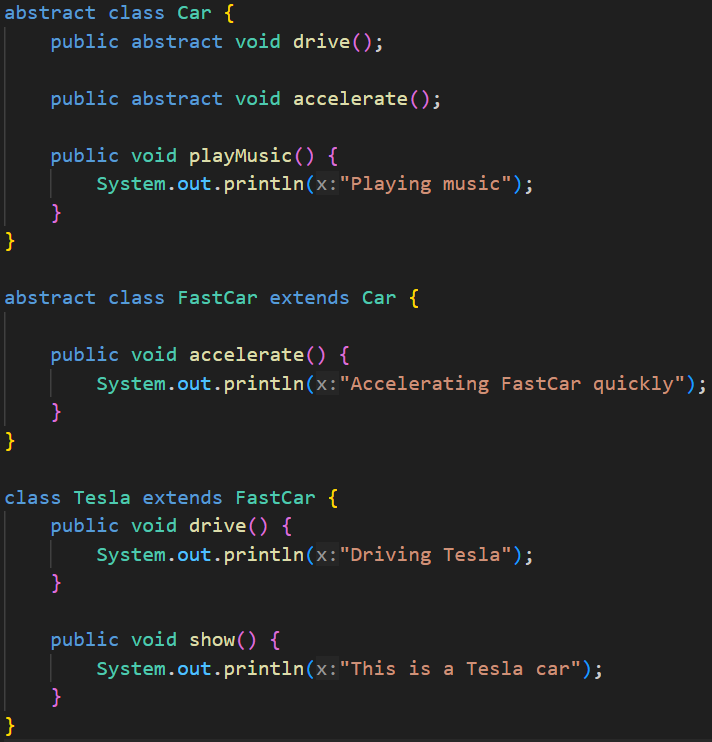
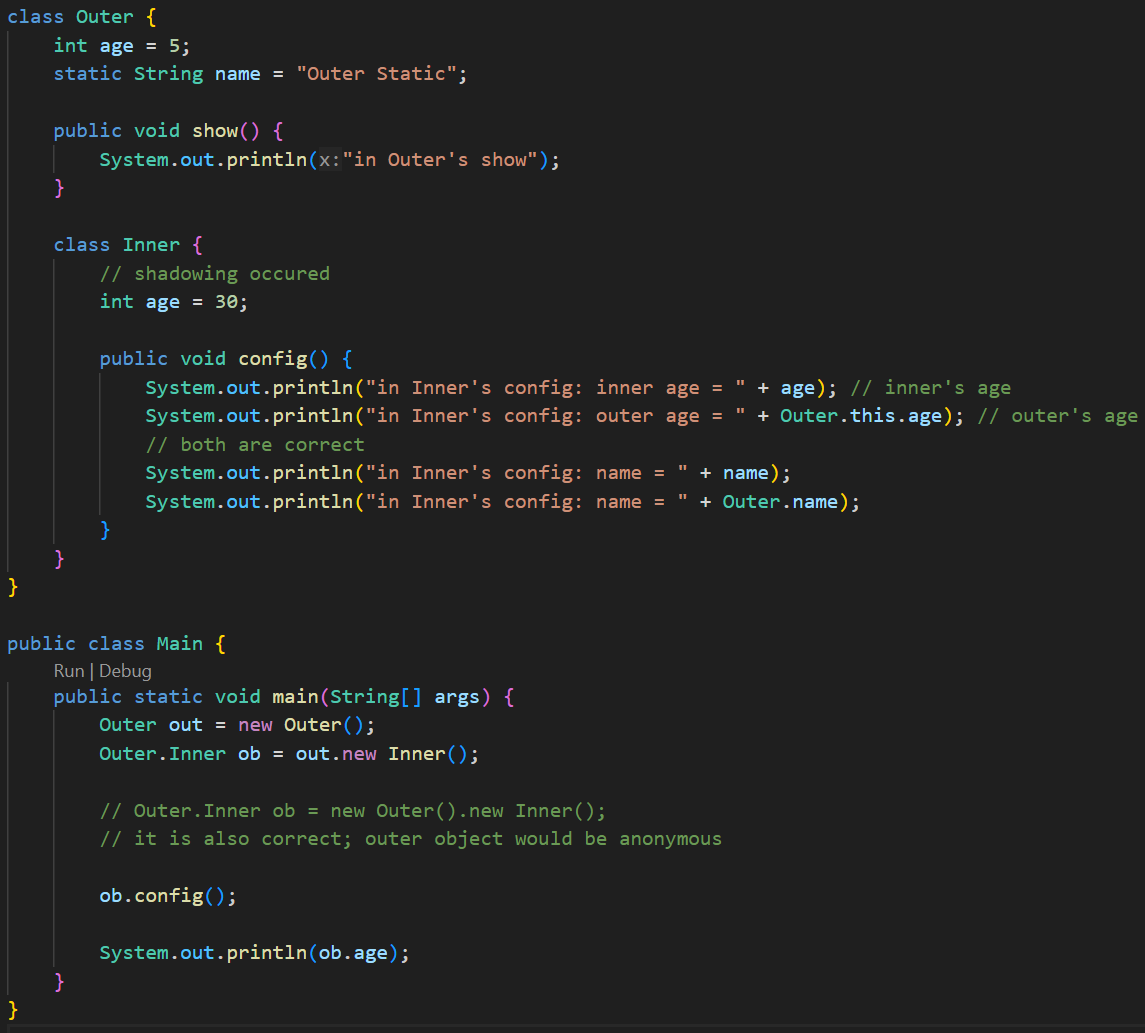
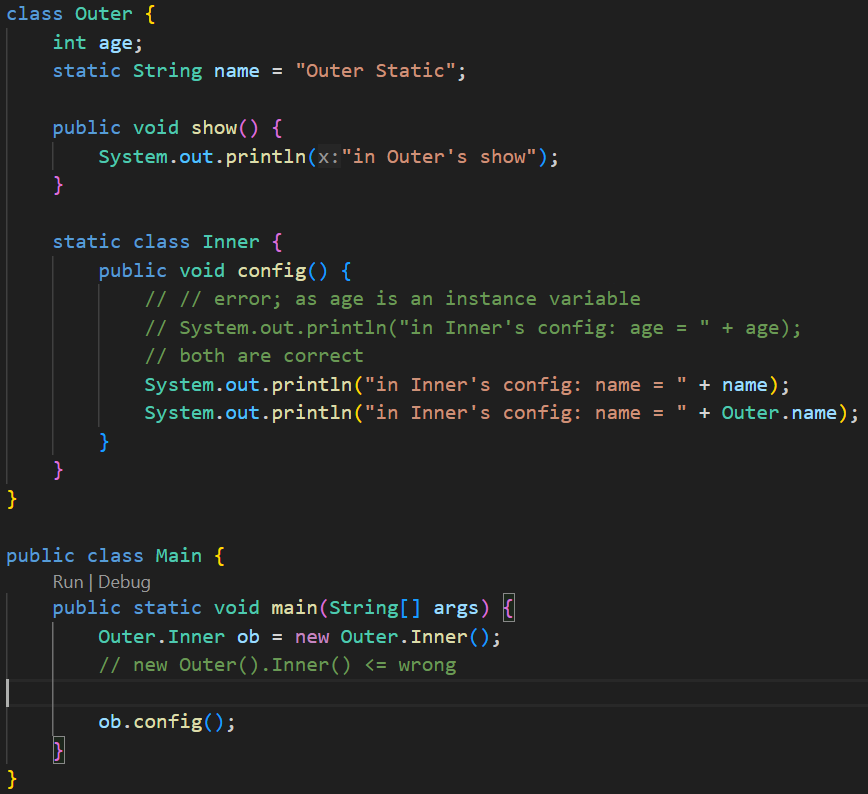
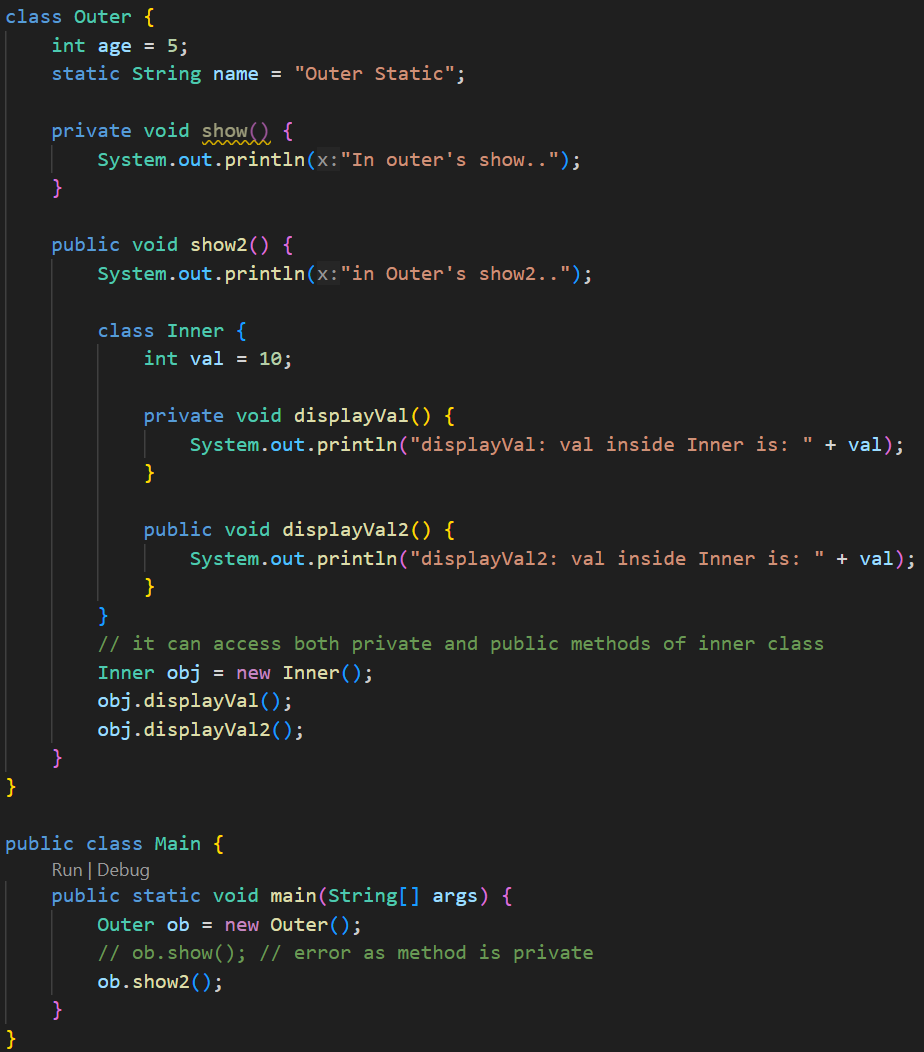
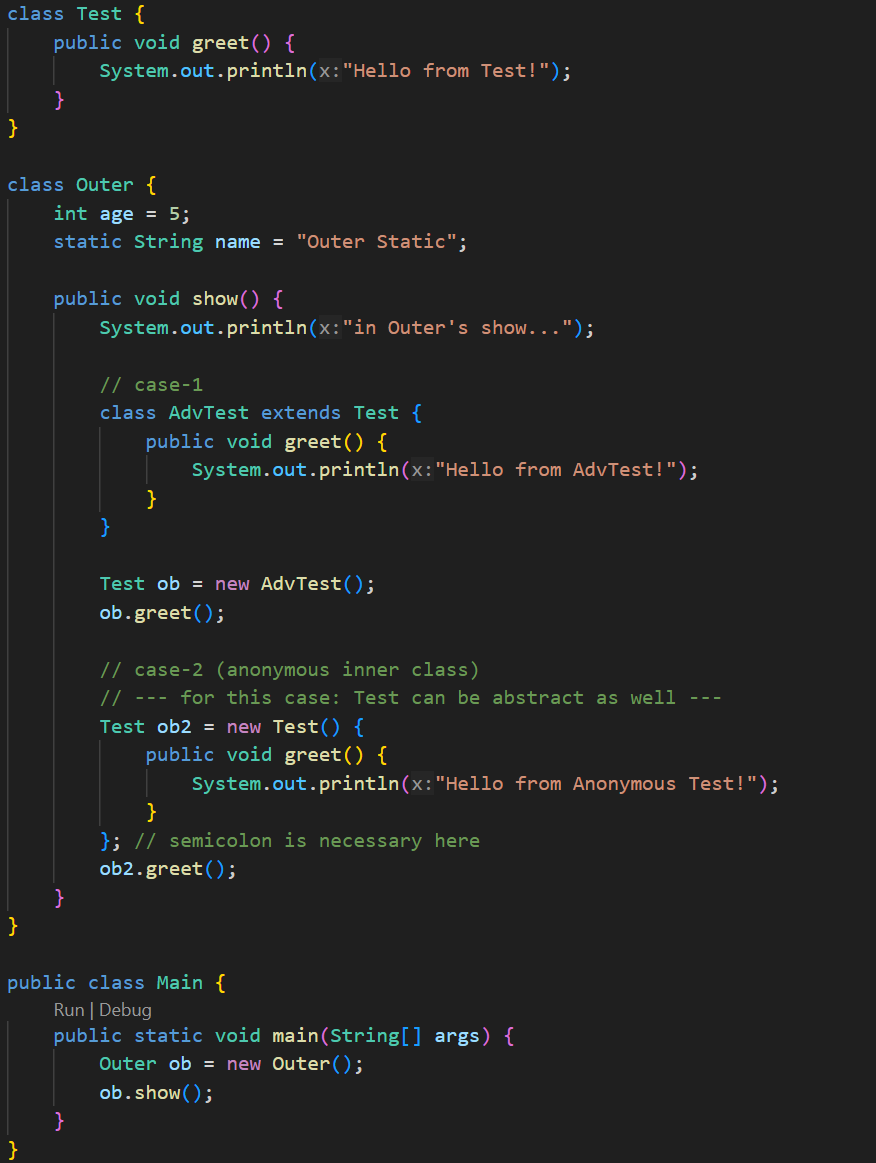
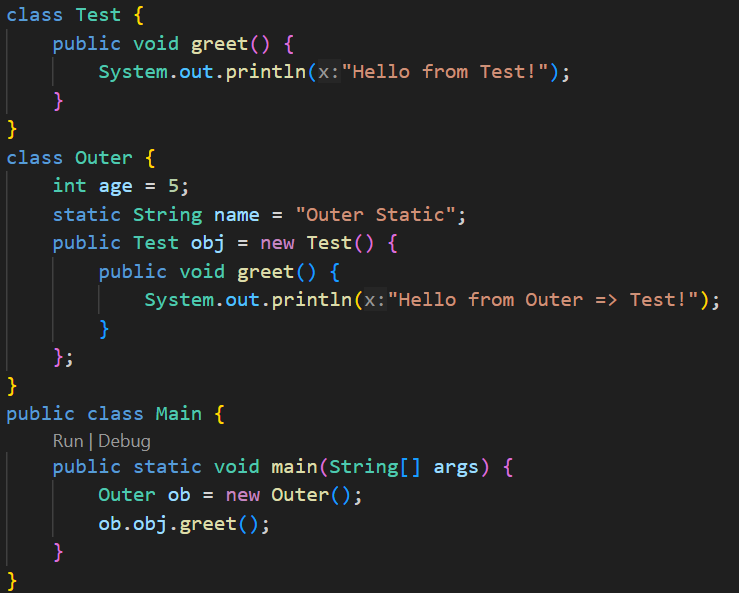
* If you try to access any method of an object, but the object is null; then you’ll get an **Null Pointer Exception**.
* **Remember: in java we initialize an array using {} curly braces; not [] square brackets.**
* Consider the following scenario:
  + I have 3 files:
    - Main.java
      * Contains 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.
  + 
  + 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:
    - 
    - 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.
* **Abstract Class and Abstract Method**
  + In java, empty methods are valid.
    - 
  + 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
  + 
  + 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.
  + **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.
    - 
* **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:**
    - 
    - 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.
    - 
    - 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.
    - 
    - 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().
  + **Anonymous Inner Class:**
    - 
    - 
    - 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 \*/ }
  + fdfdfdf