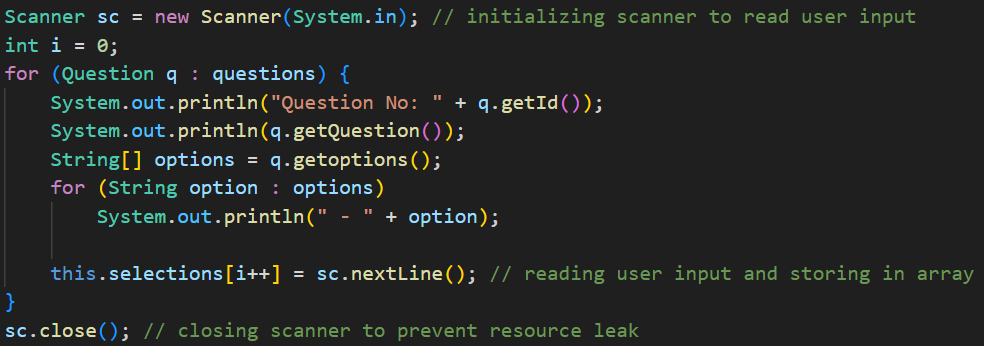
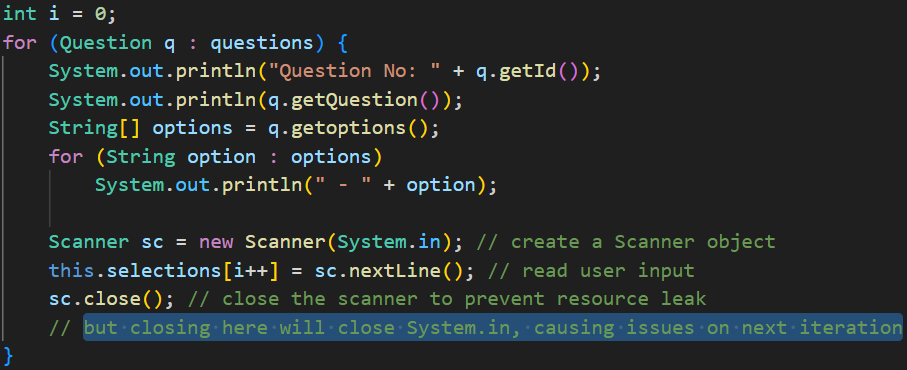
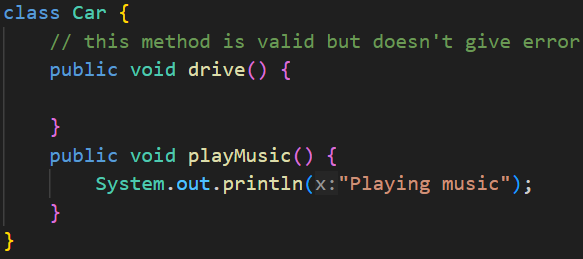
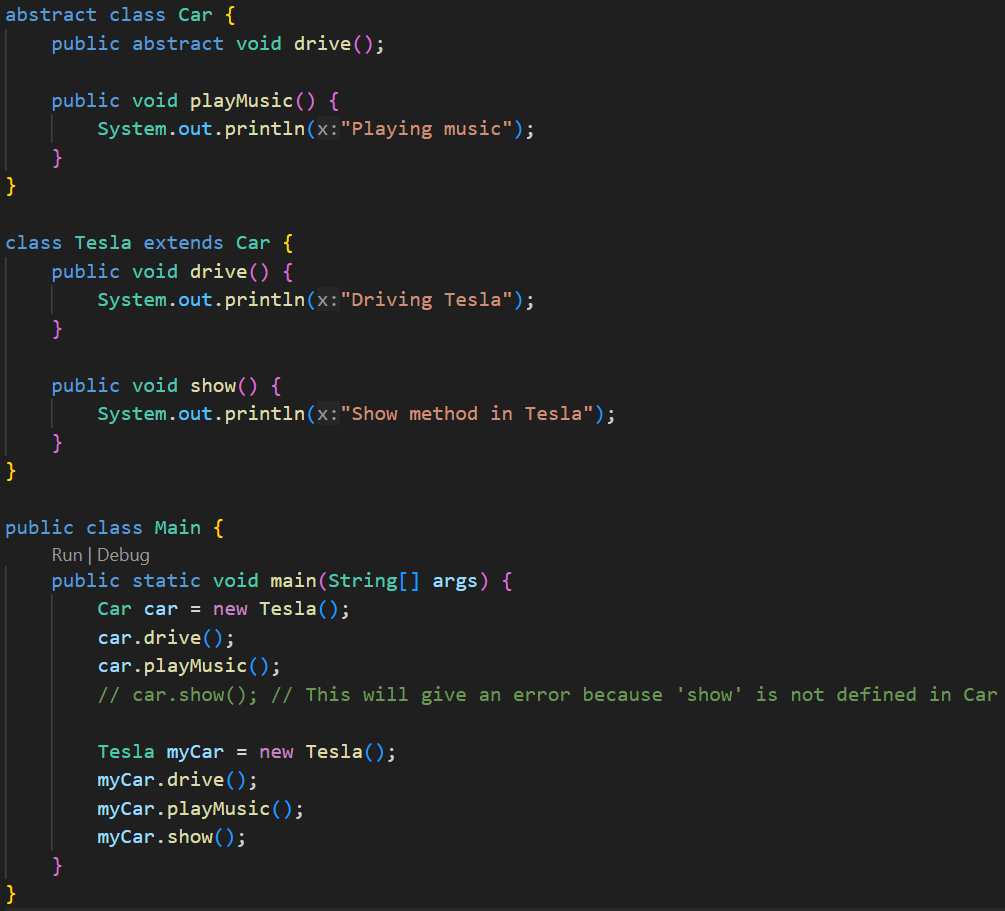
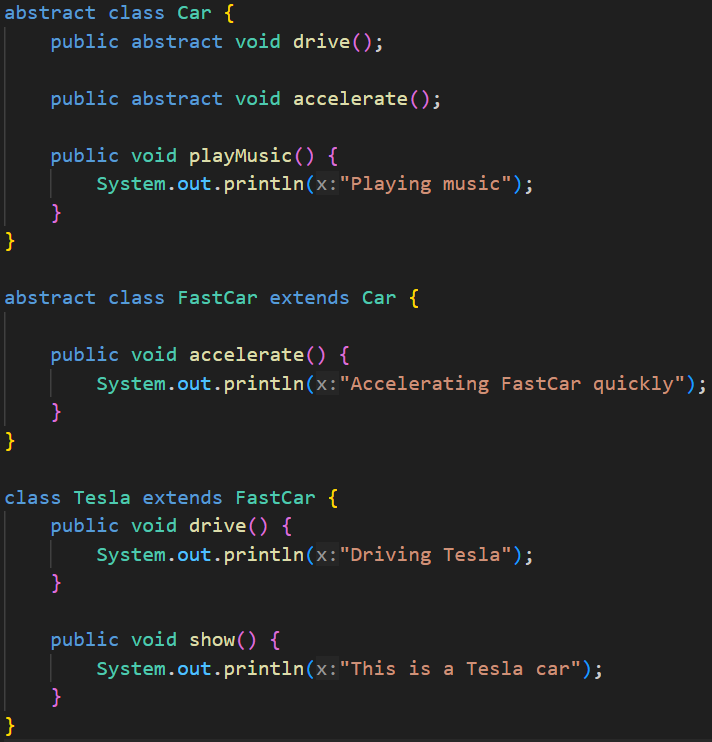
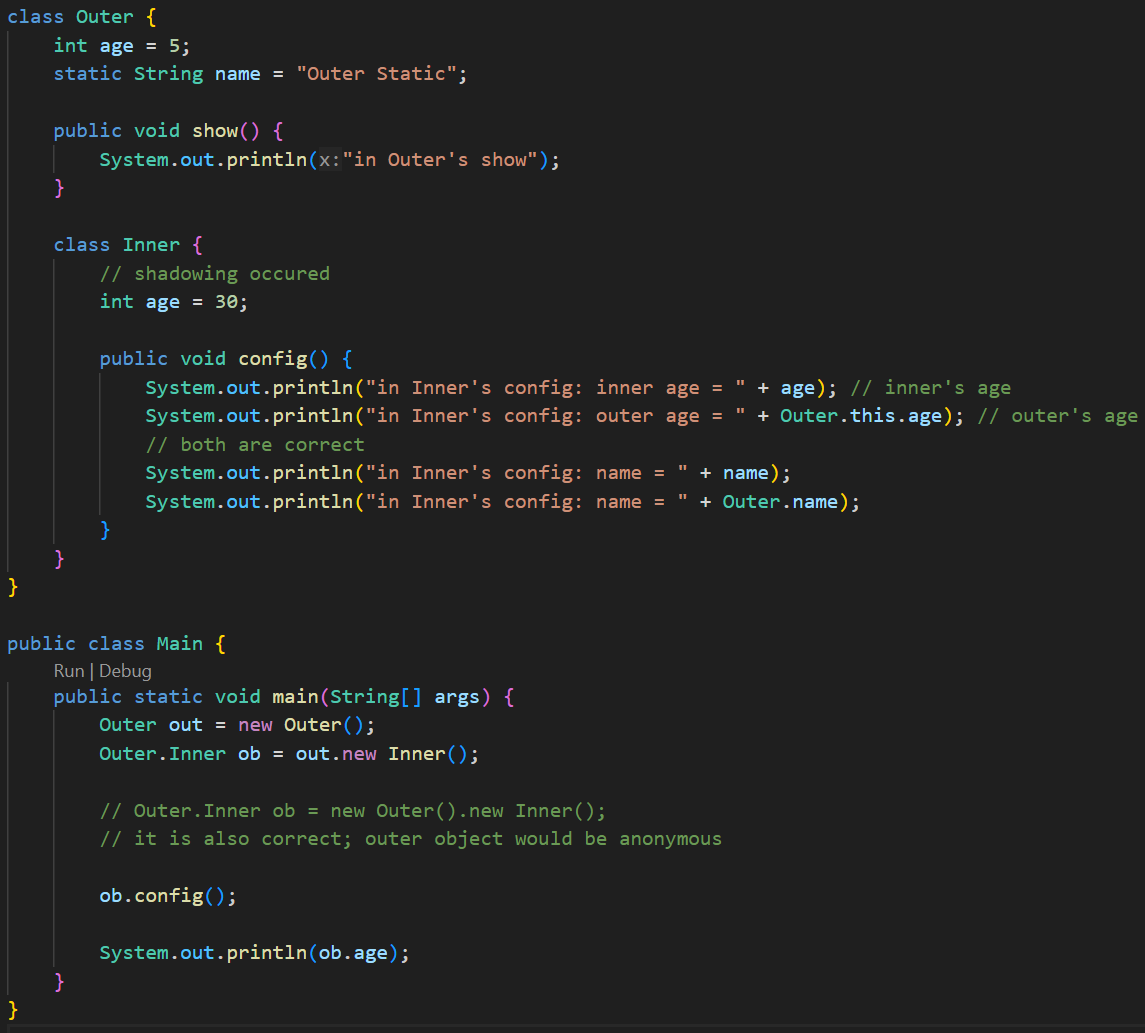
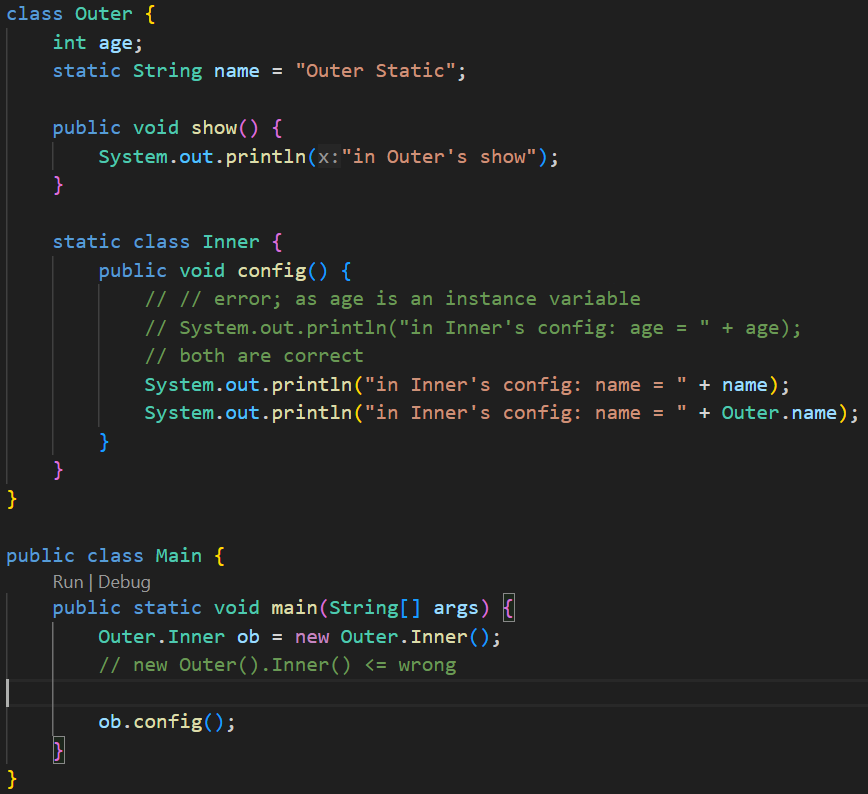
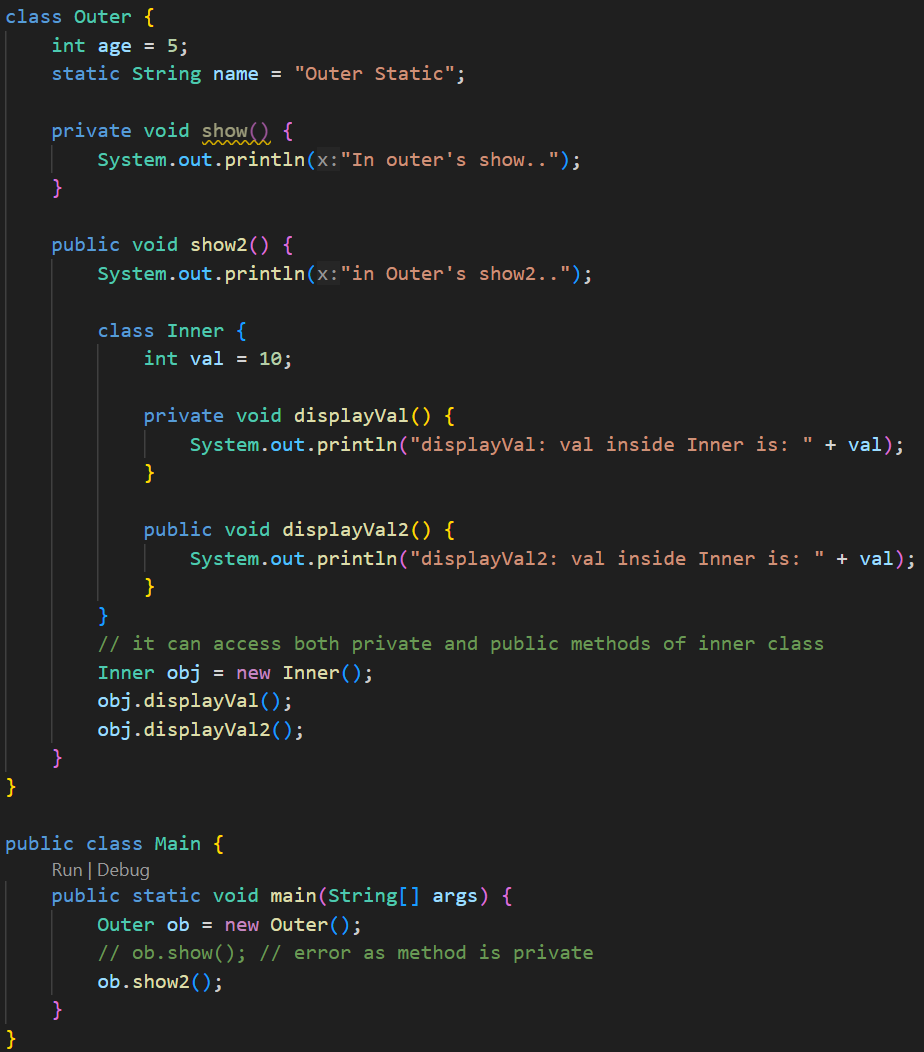
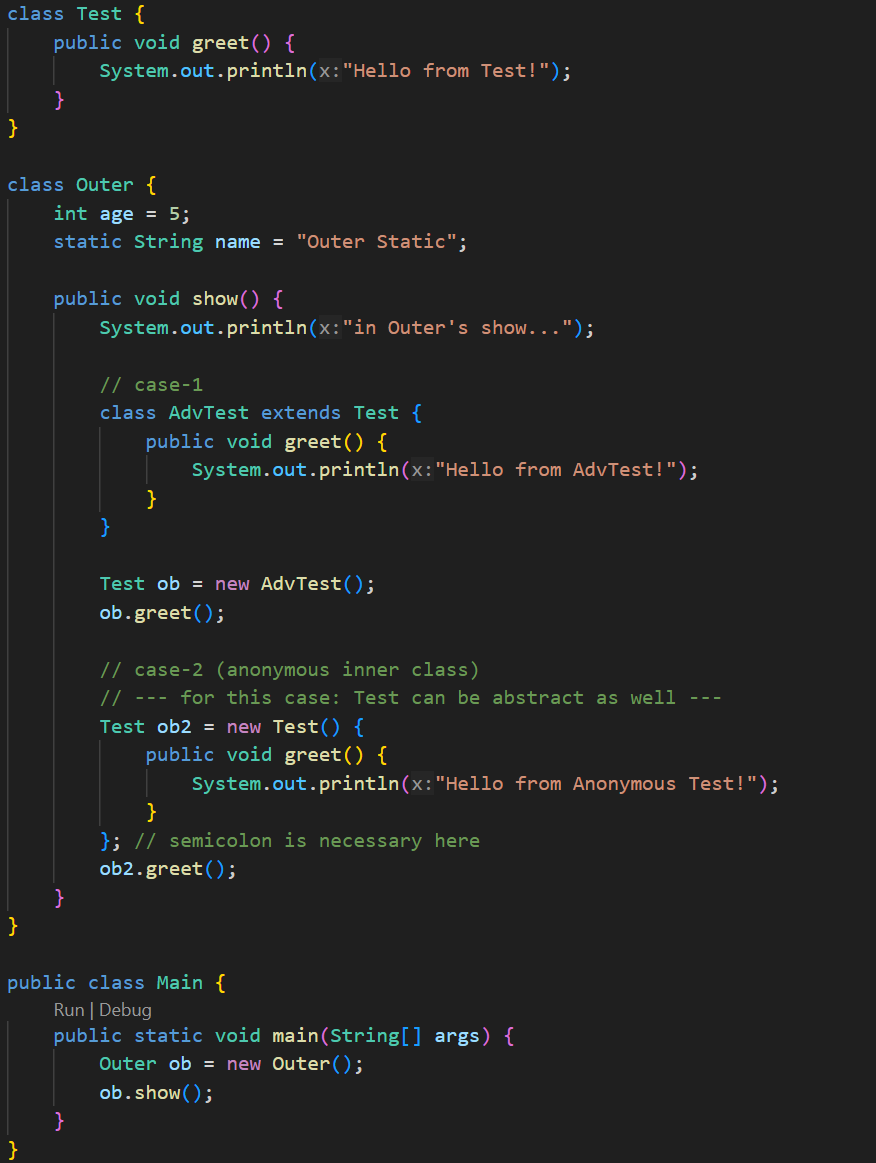
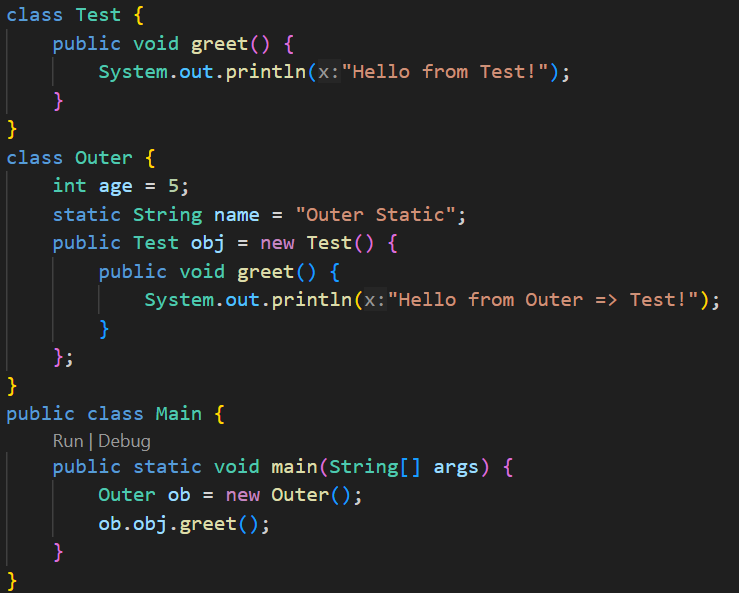
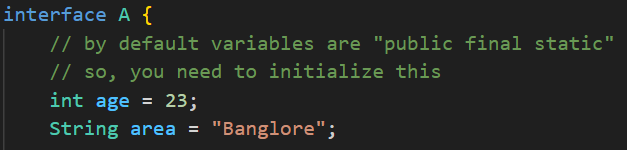
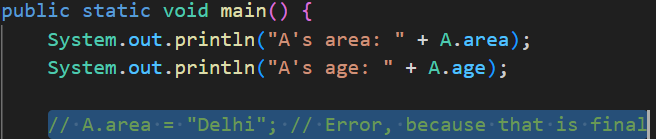
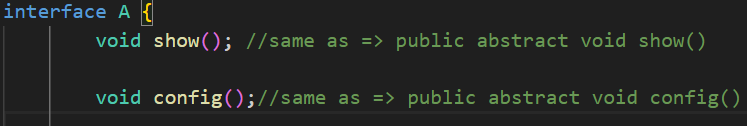
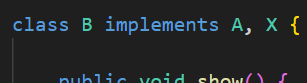
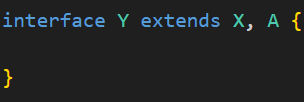
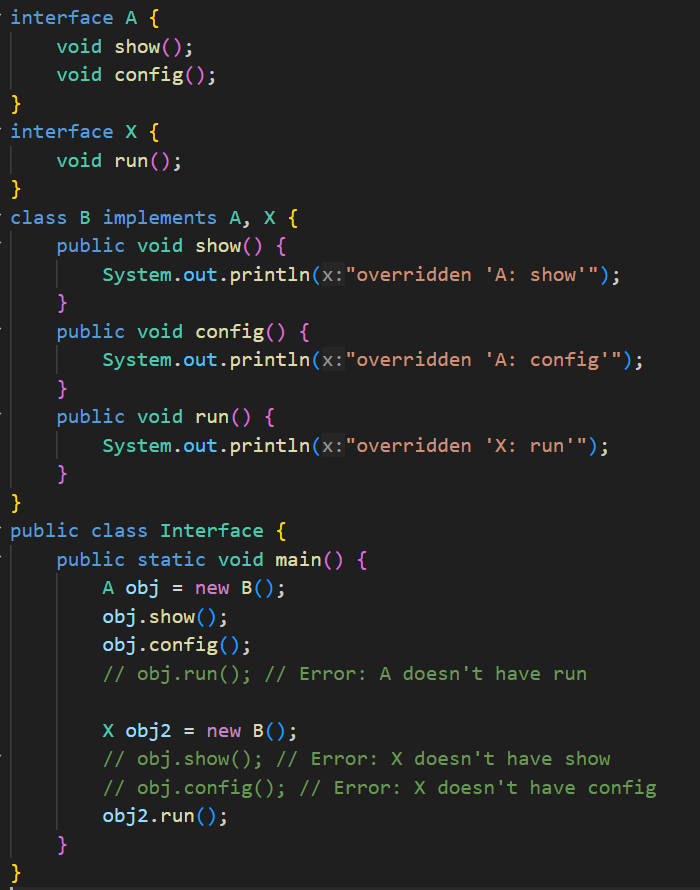
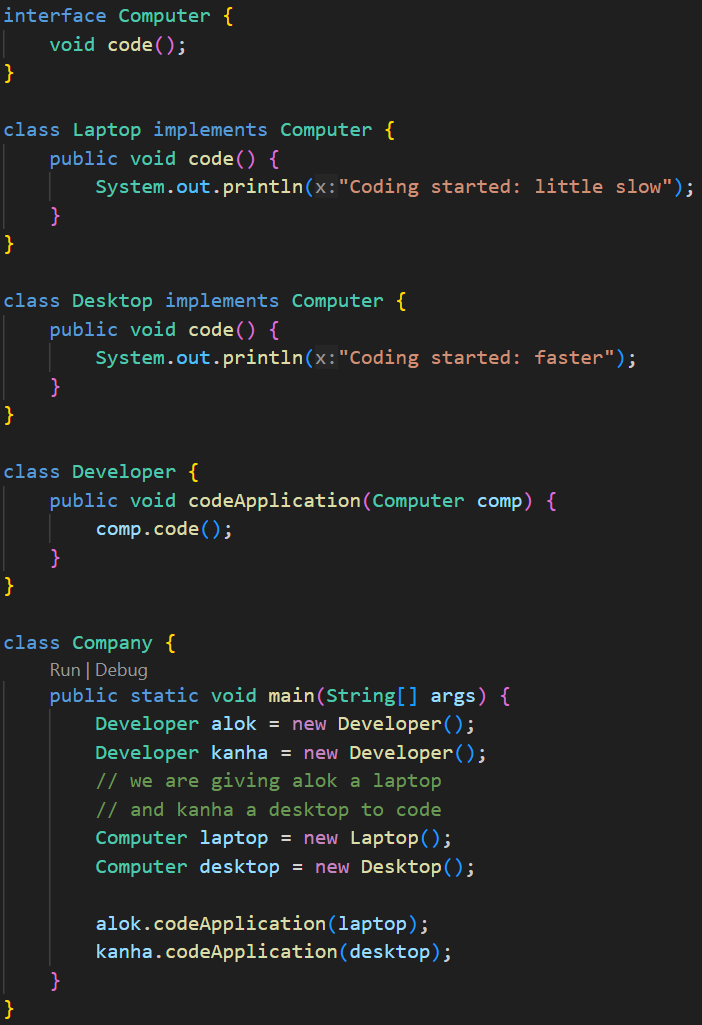
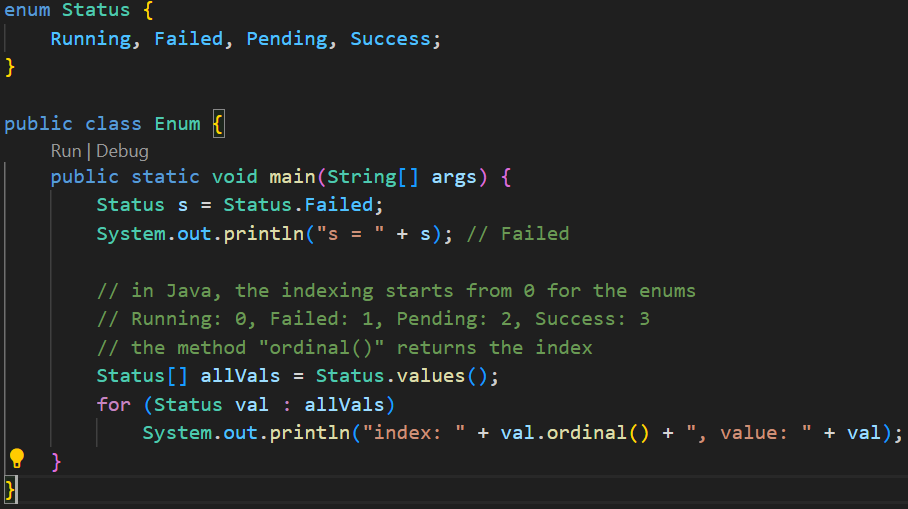
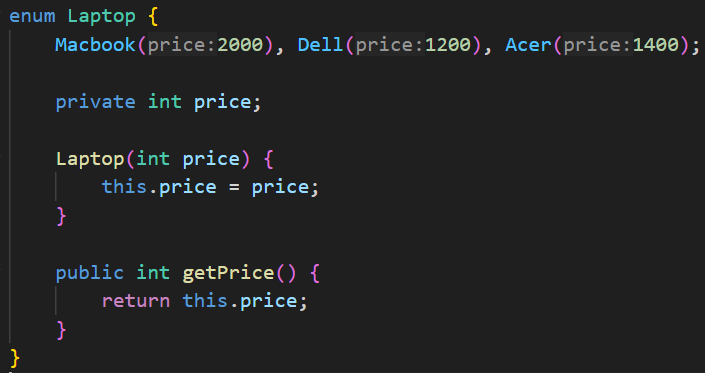
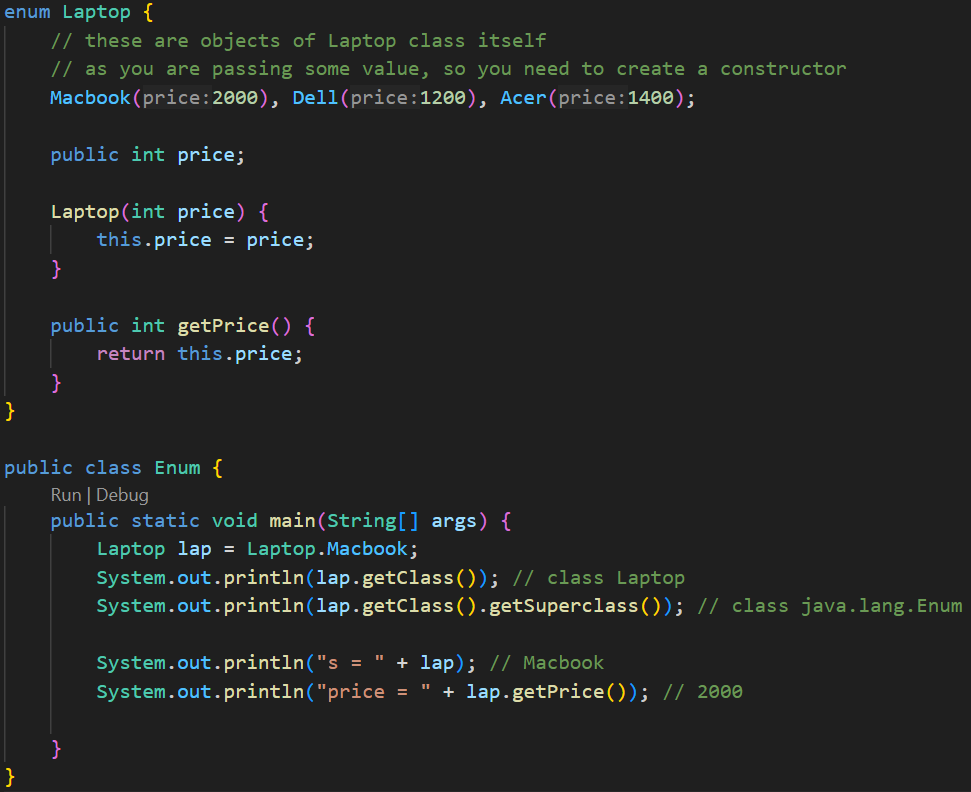
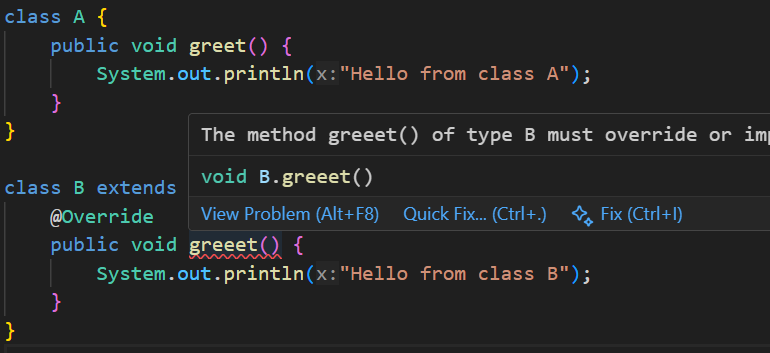
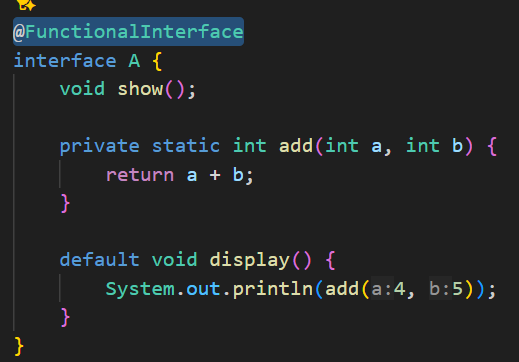
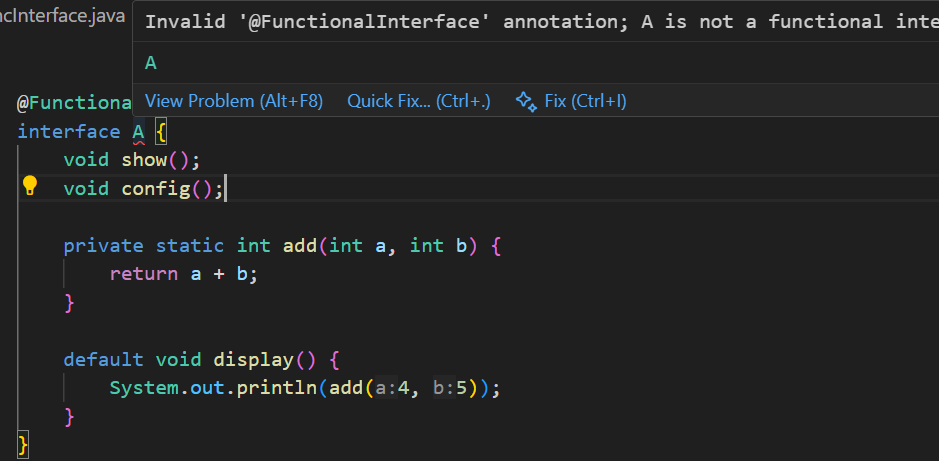
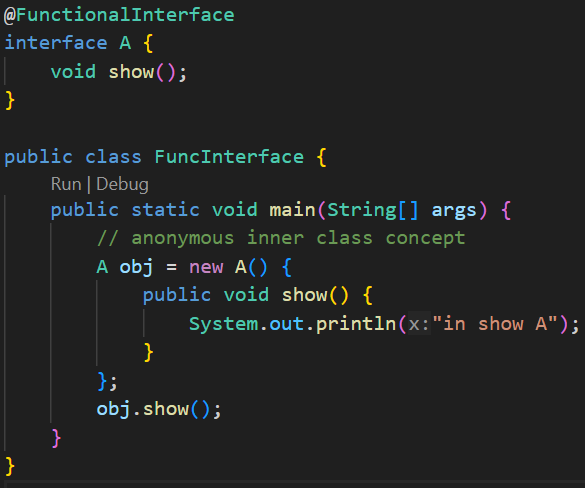
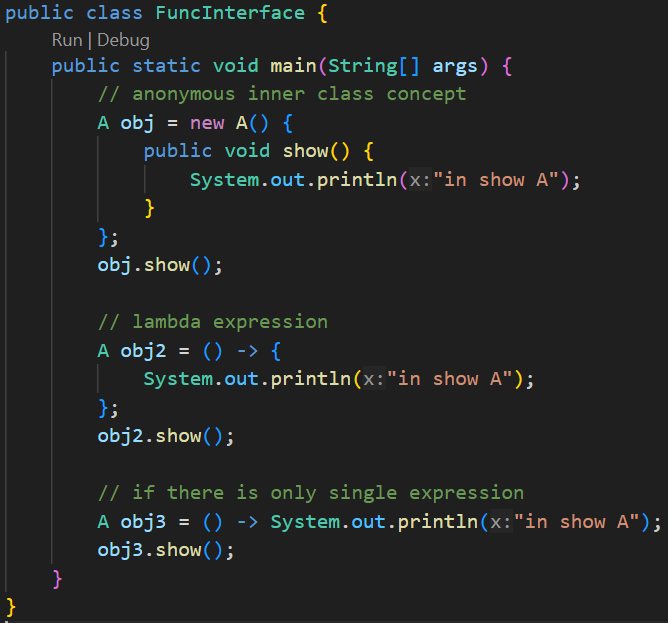
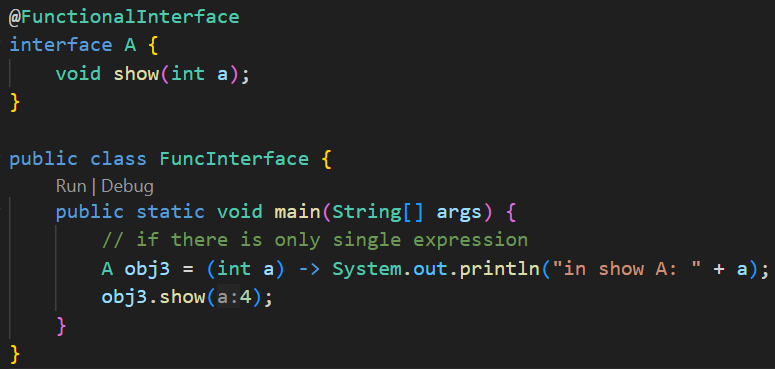
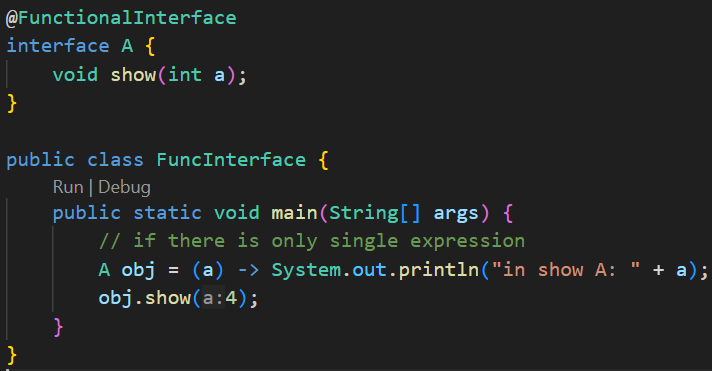
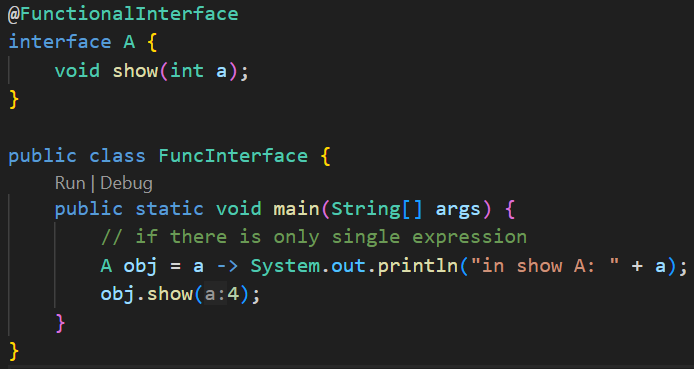
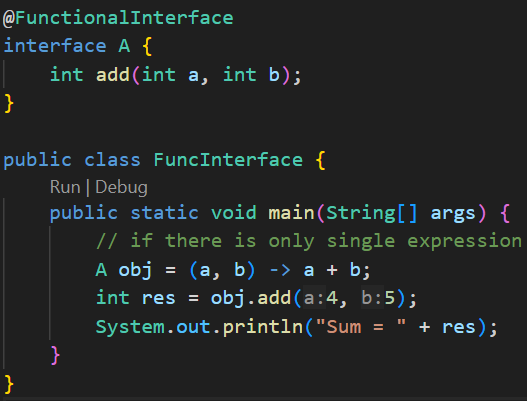
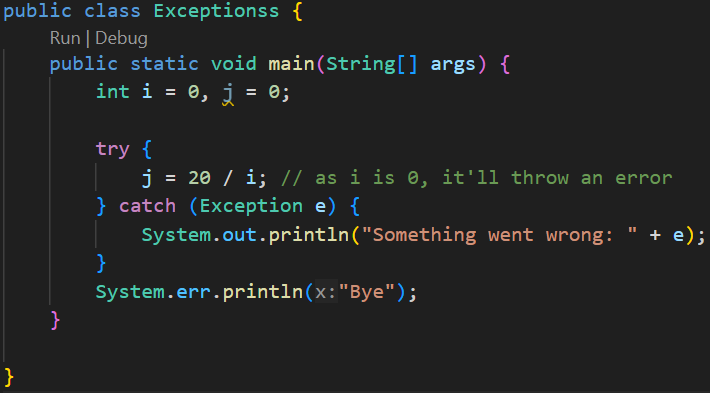
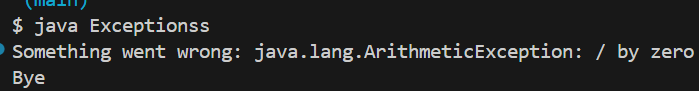
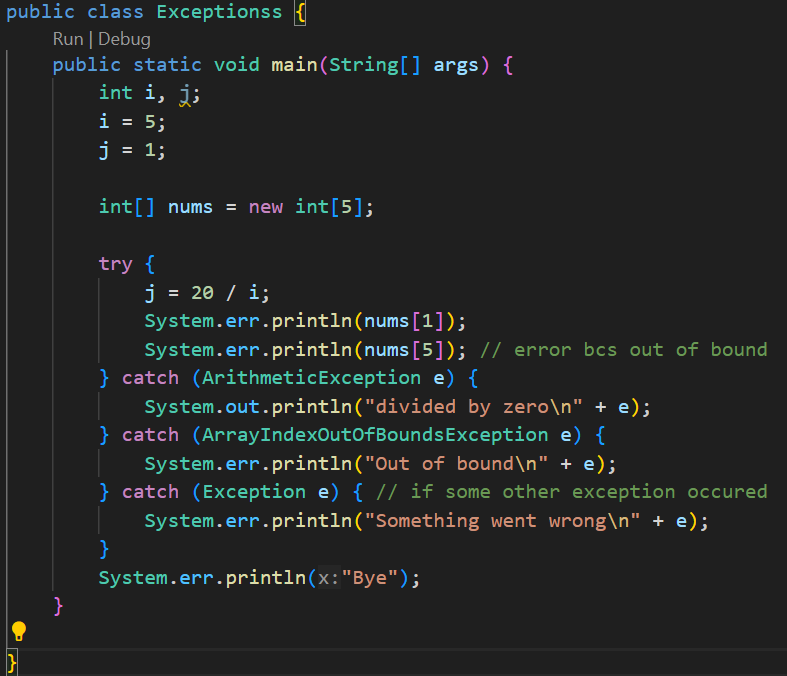
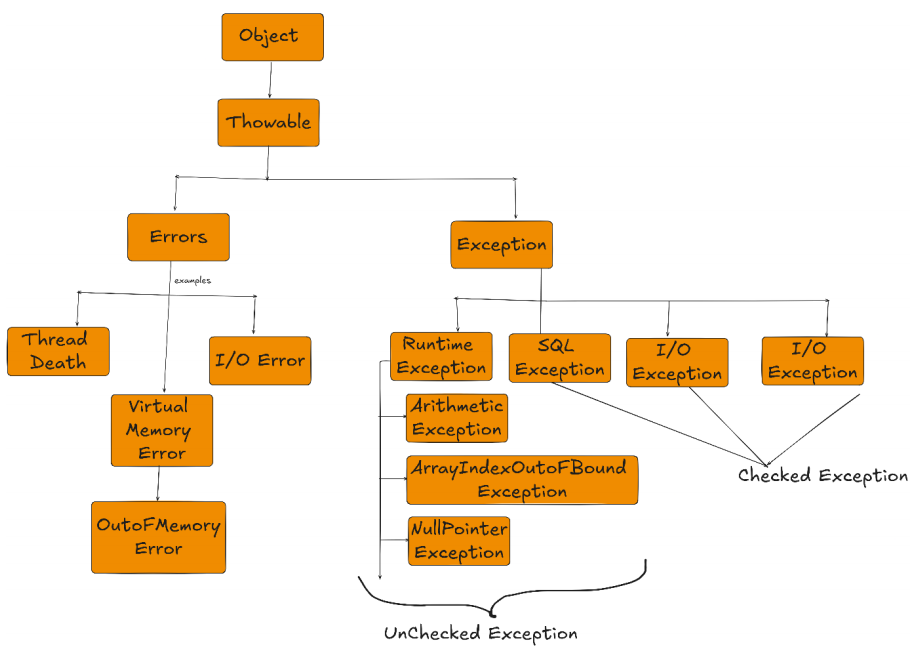
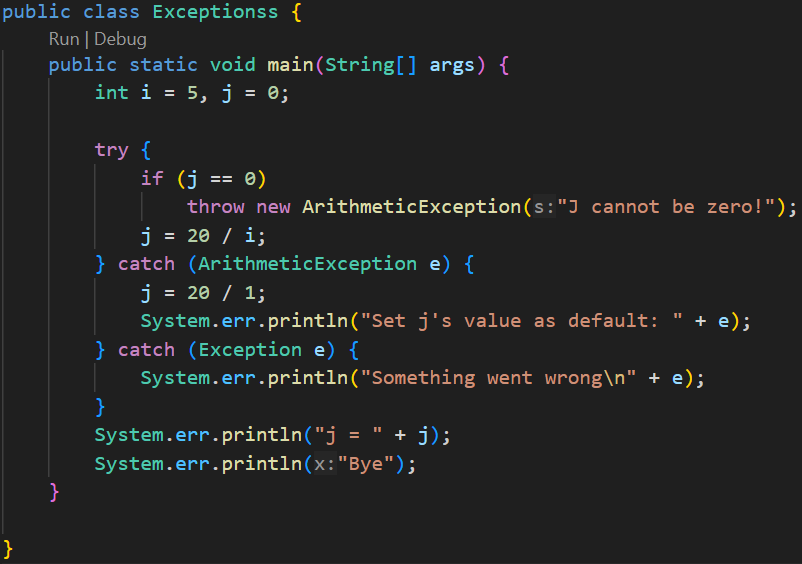
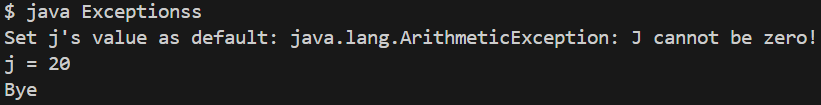
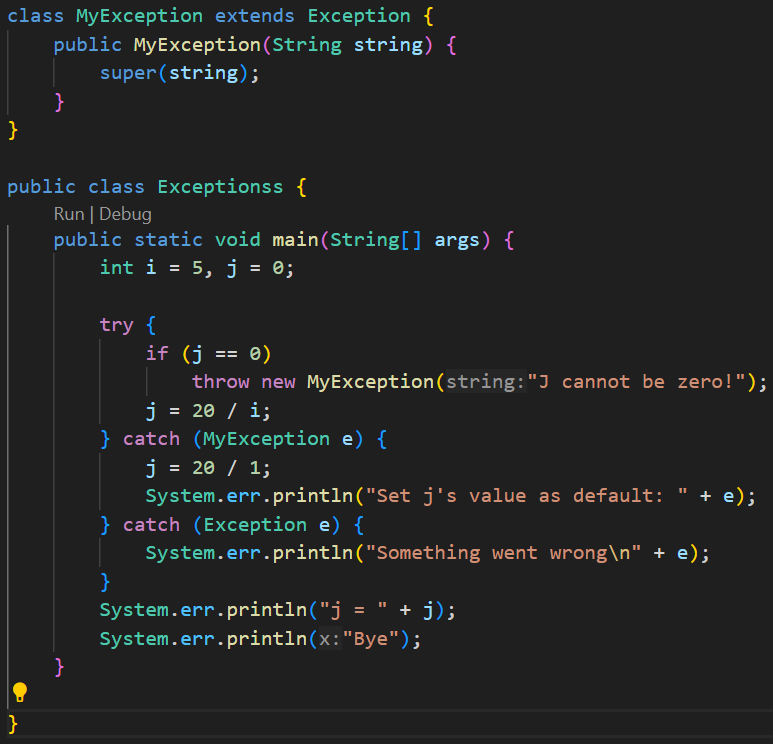
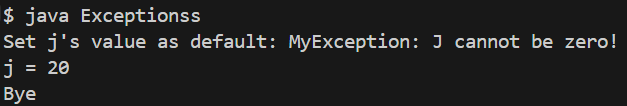
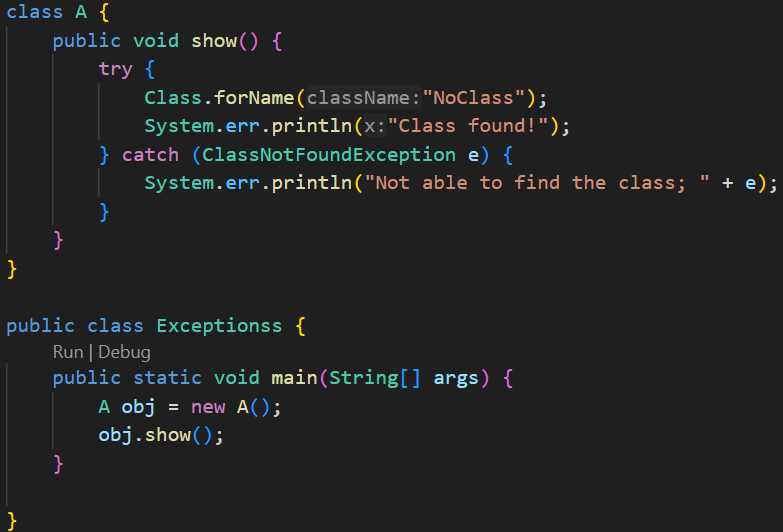
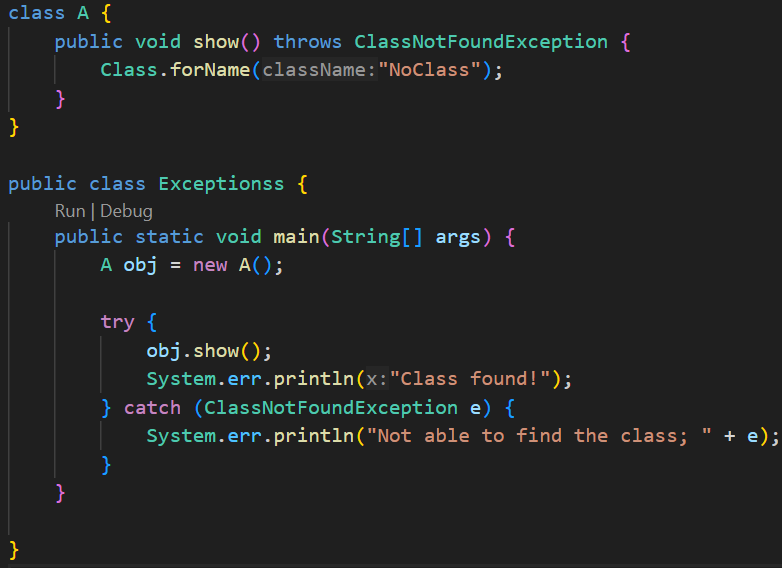
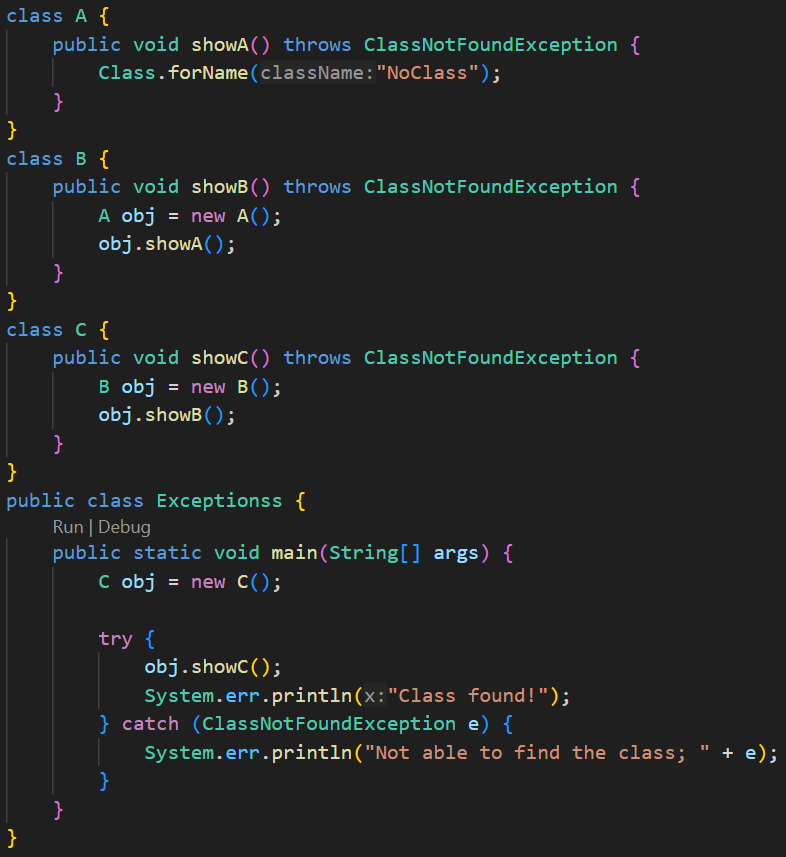
* If you try to access any method of an object, but the object is null; then you’ll get an **Null Pointer Exception**.
* **Remember: in java we initialize an array using {} curly braces; not [] square brackets.**
* Consider the following scenario:
  + I have 3 files:
    - Main.java
      * Contains 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.
  + **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.
    - 
* **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 \*/ }
* **Interface**
  + By default the variables inside interfaces are: **public static final**.
    - So, you need to initialize while declaring it.
    - 
    - You cannot override the variables that were declared and initialized in interface.
      * 
  + By default all the methods are **public abstract**; you don’t need to explicitly write that.
    - 
  + **implements** is the keyword that is used to implement a interface to a class.
  + Unlike classes, **multiple implementations** are allowed in case of interface.
    -  (multiple implementation)
  + Interfaces can inherit another interface.
    - In this case, **multiple inheritance** is allowed.
    - 
    - 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.
  + 
    - 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.
  + 
* **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.
  + 
  + **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)
    - 
    - 
      * Behind the scene.
    - 
* **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.
  + 
    - 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:**
    - 
      * 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**
  + 
    - This code is proper and it’ll work fine.
  + 
  + 
    - You can also pass the arguments.
  + 
    - You don’t even need to provide the data type; it’ll take from the interface directly.
  + 
    - If you have only one argument, don’t need to give the *parenthesis* as well.
  + 
    - 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.
* Exceptions:
  + Compile time error and Logical Errors can be fixed;
  + But Run Time error should be handled. So that the application won’t stop in between.
  + Exception Handling is nothing but handling these Run Time error.
  + 
    - Handling Exceptions using **try catch** block.
    - 
  + 
    - Using multiple **catch** blocks to catch different types of Exceptions.
  + 
    - This is the hierarchy of Exception classes.
    - **Checked** means the exceptions that are checked during compile-time. i.e. IOException, ClassNotFoundException, SQLException
    - **Unchecked** means the exceptions that are occur during the run-time i.e. NullPointerException, ArithmeticException, ArrayIndexOutOfBoundException ..etc etc
* **Throw Keyword**
  + 
    - We can throw any kind of exception we want by giving some customized error message.
    - 
  + 
    - It is a **custom exception**.
    - You need to inherit the **Exception** class or can also inherit **RuntimeException** class; and pass the string to the super class’s constructor because those Exception classes handles this message.
    - 
* **throws keyword**
  + Let suppose, in a method **A** , you are calling 2 methods **B** and **C**.
  + **B** and **C** both are having a critical expression that might throw the same Exception.
  + So, instead of handling those inside **B** and **C** both, we can handle those inside **A** directly.
  + For this, **throws** keyword will be used in **B** and **C**.
  + It is used to forward the Exception to the method where the current method is called.
  + Lets **A** is being called in some another method **X**, if you mention the keyword **throws** in the method **A** as well, then the Exception occurred from **B** and **C** will go to **A** then it’ll go to **X**.
  + 
    - Simple code only; exception will be thrown inside the **show()** method and will be handled there only.
  + 
    - So, like this you can use **throws** to forward the Exception to calling method.
  + 
    - Here the exception flows from A to Exceptionss class:
      * A’s showA ⇒ B’s showB ⇒ C’s showC ⇒ Exceptionss’s main
  + dfd
* Fdffdfdfdf