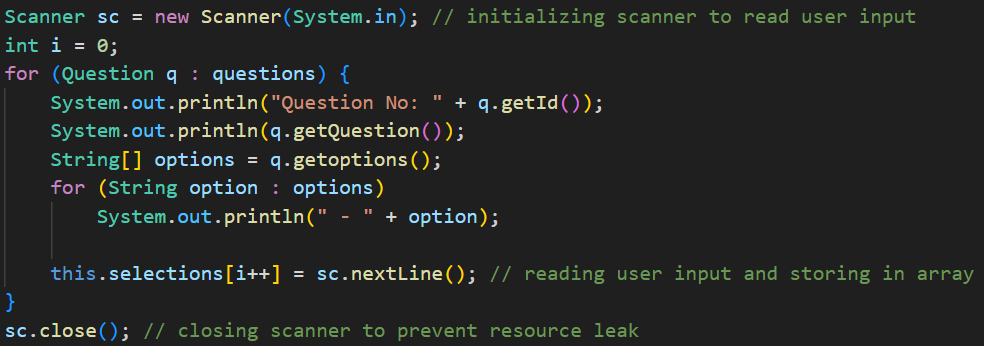
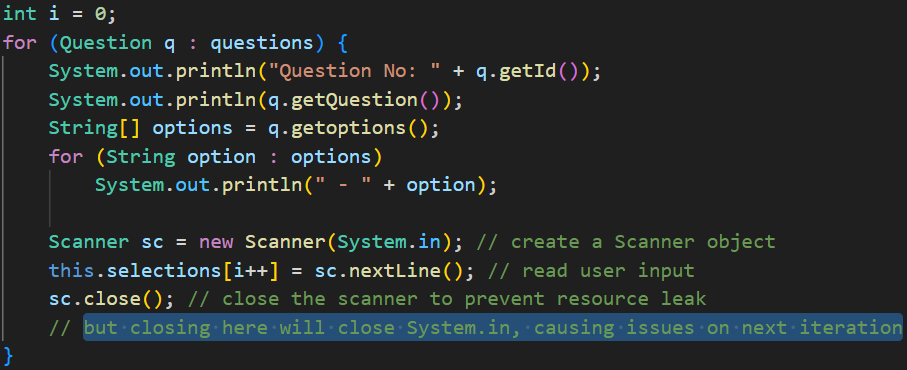
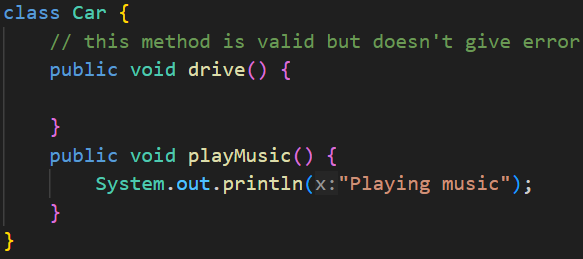
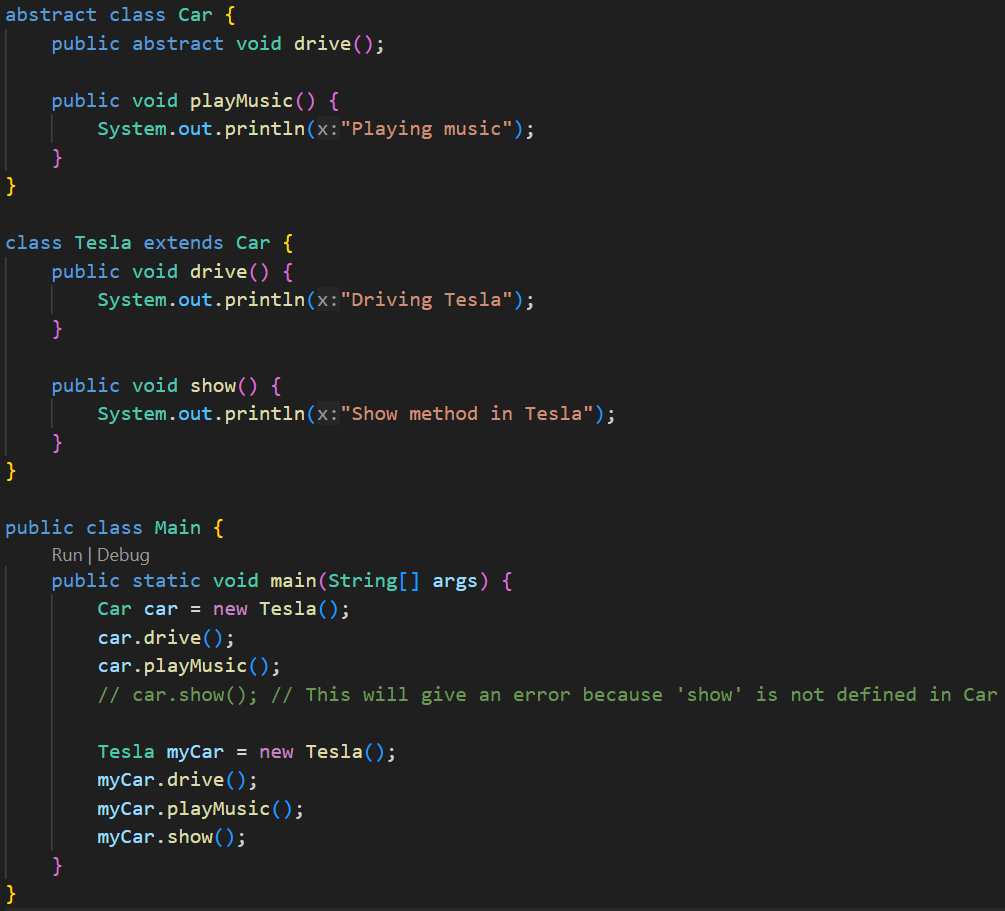
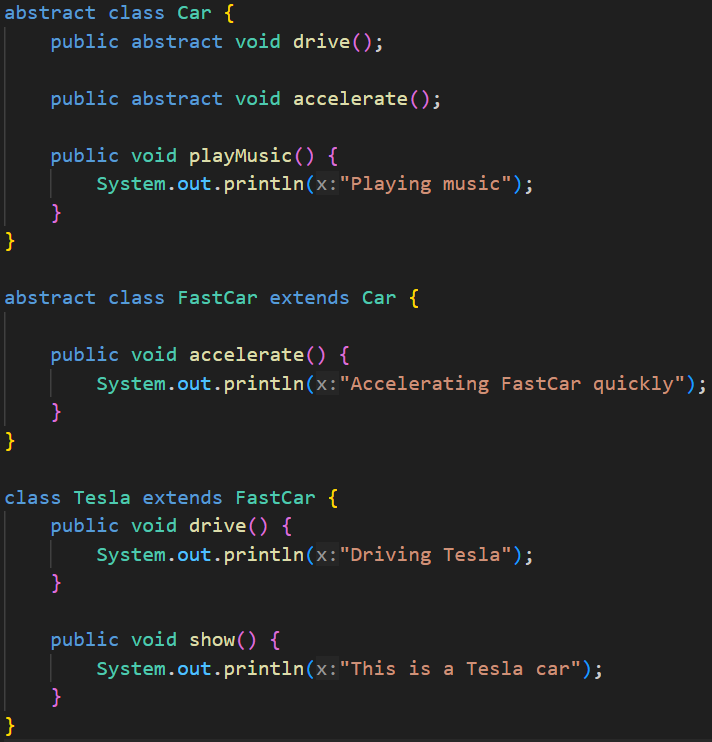
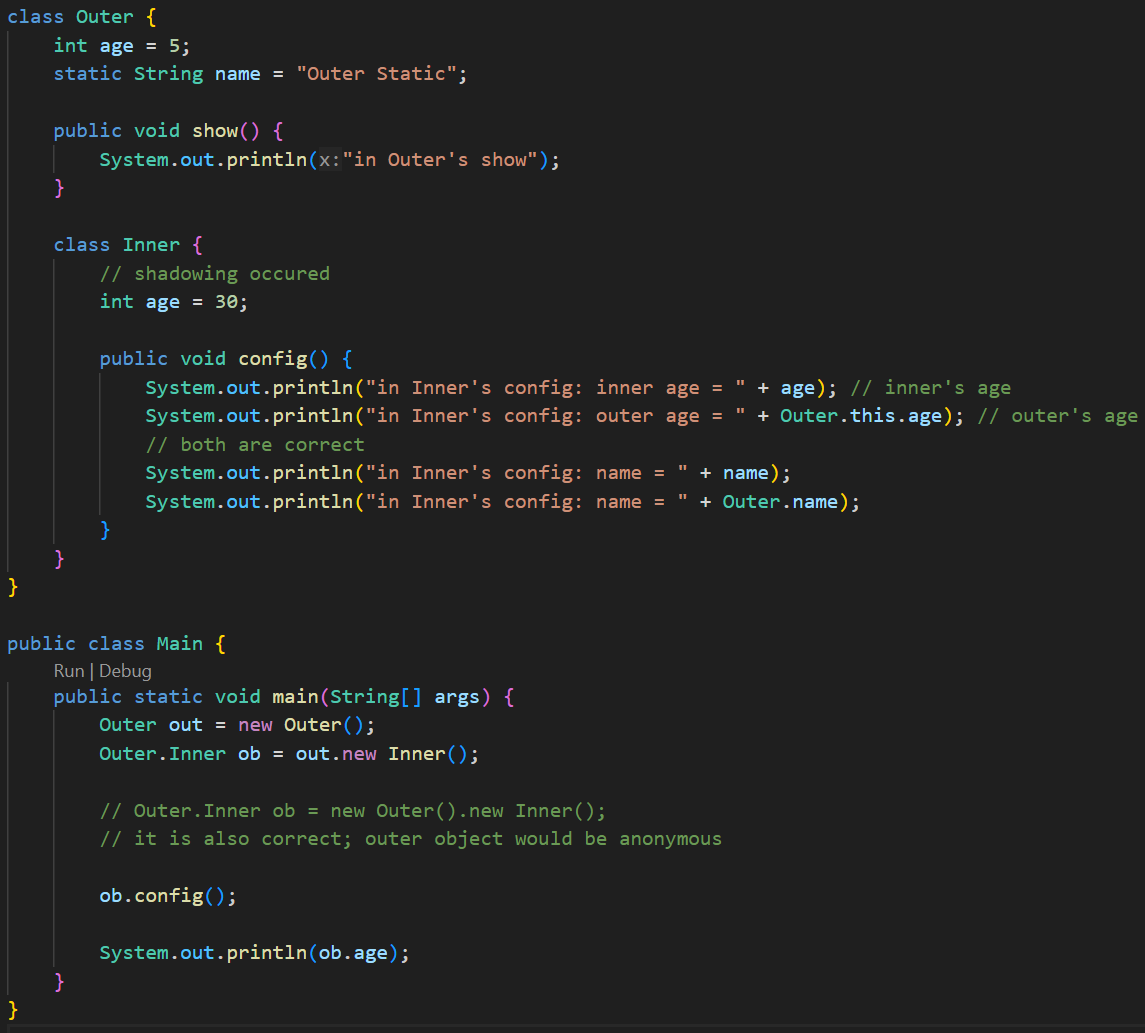
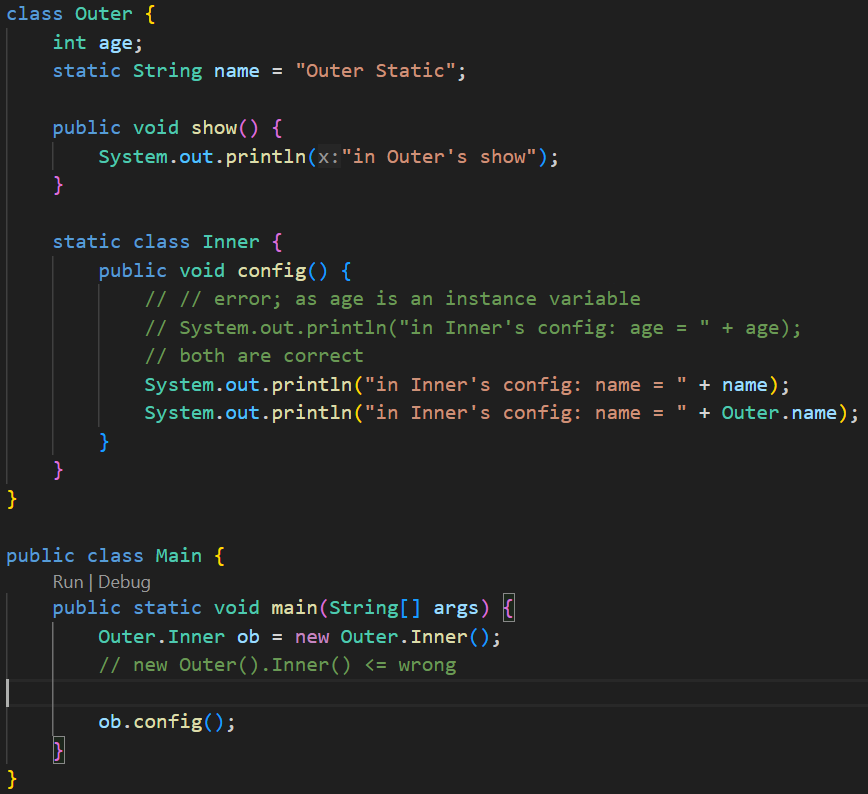
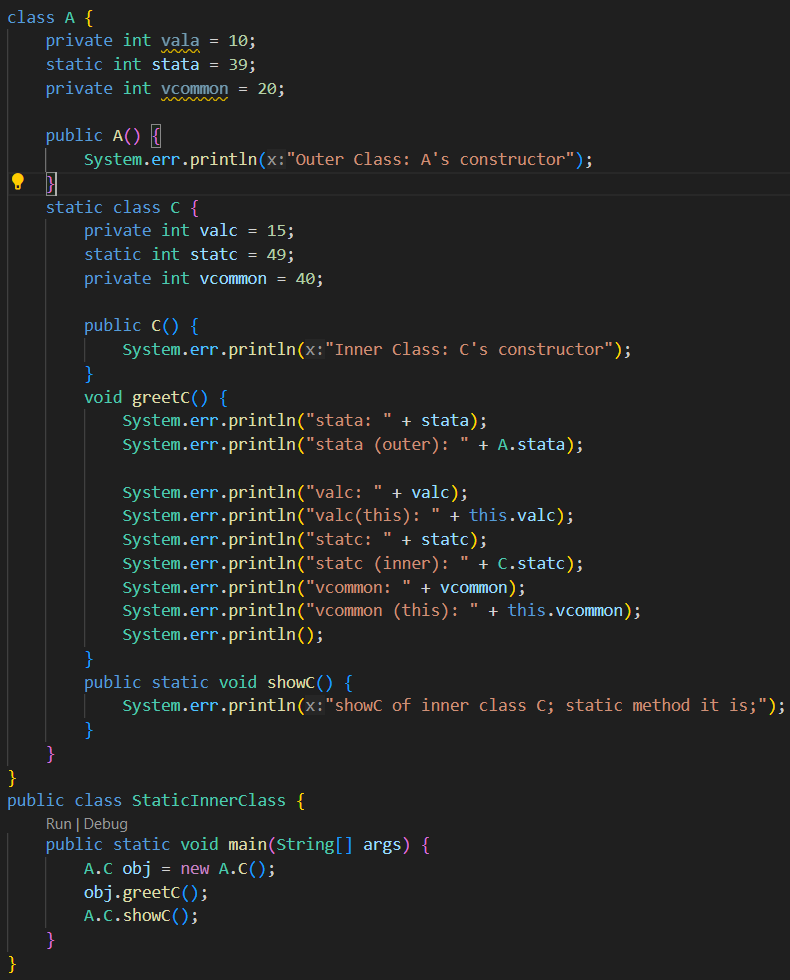
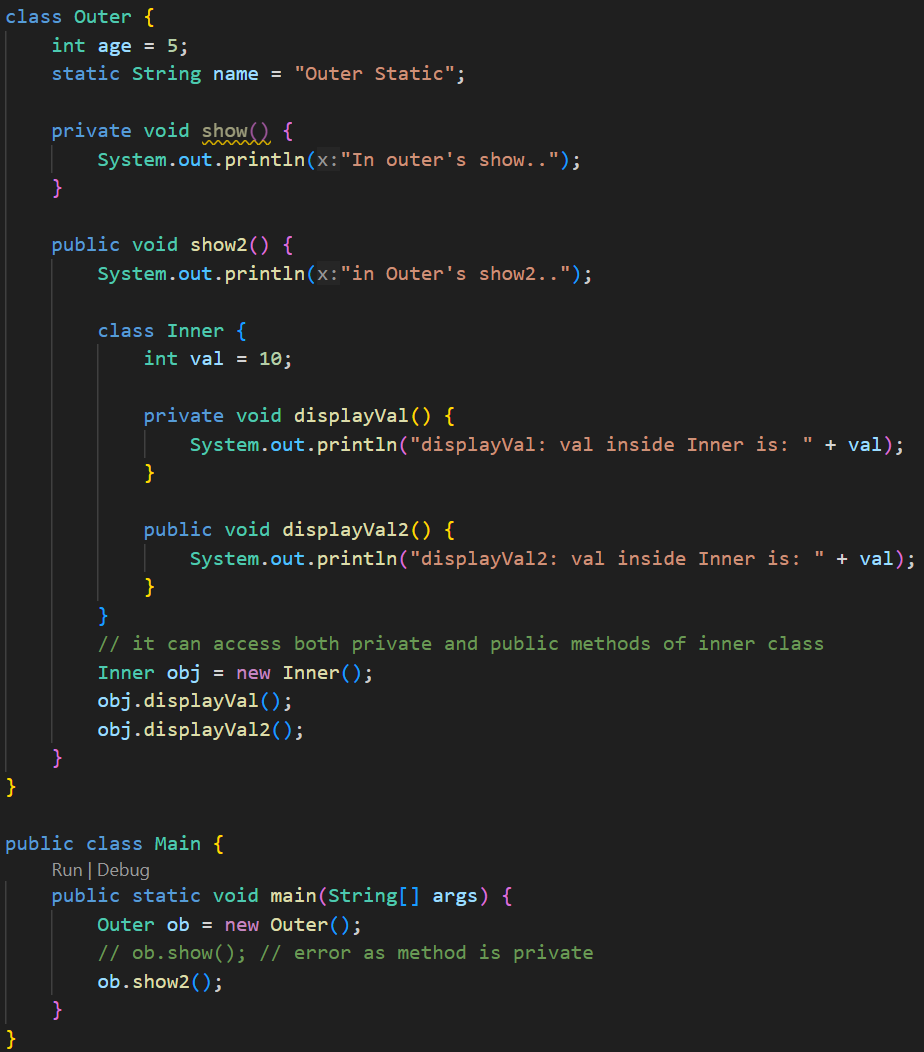
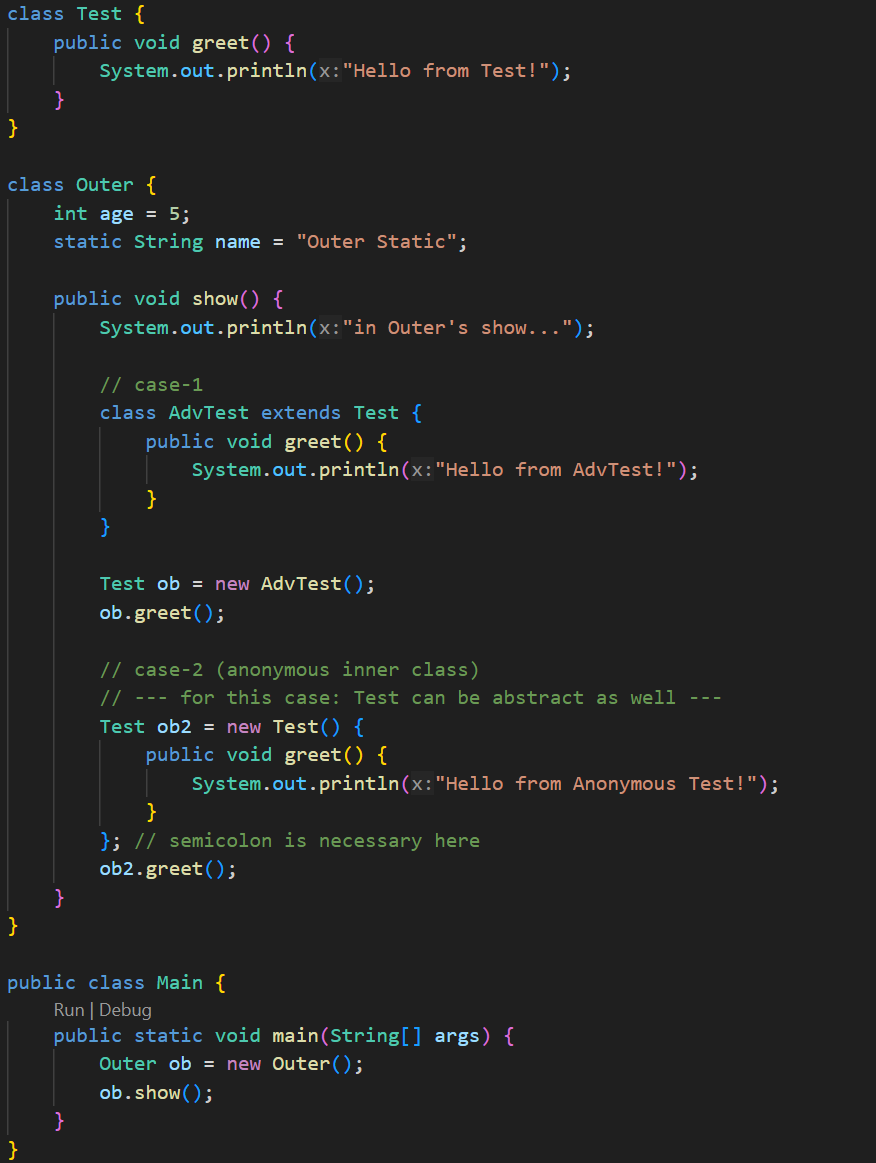
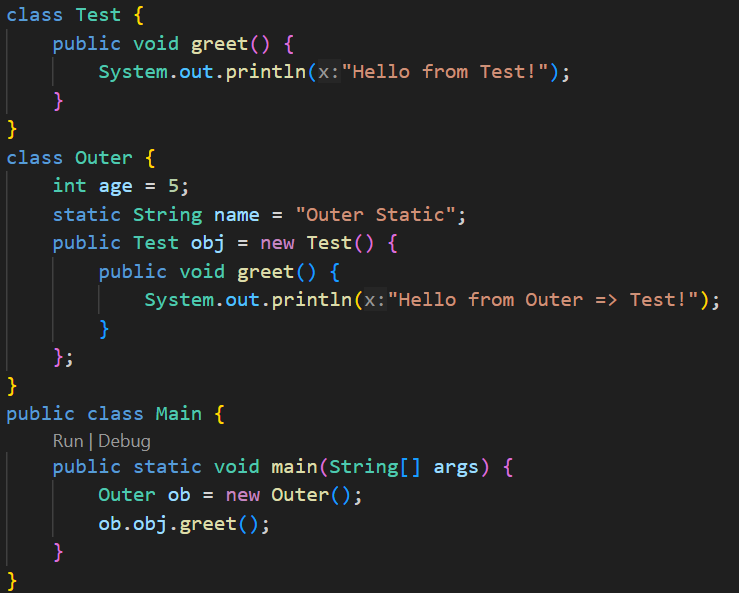
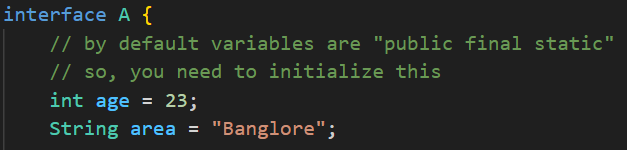
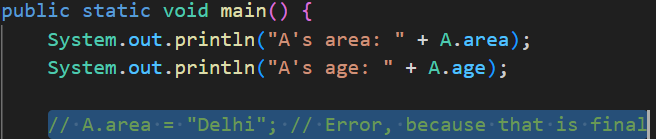
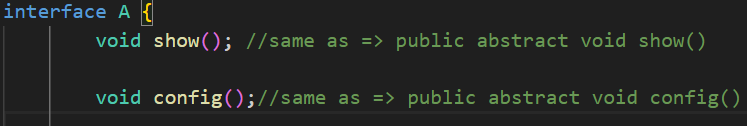
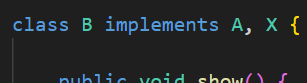
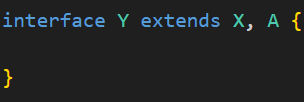
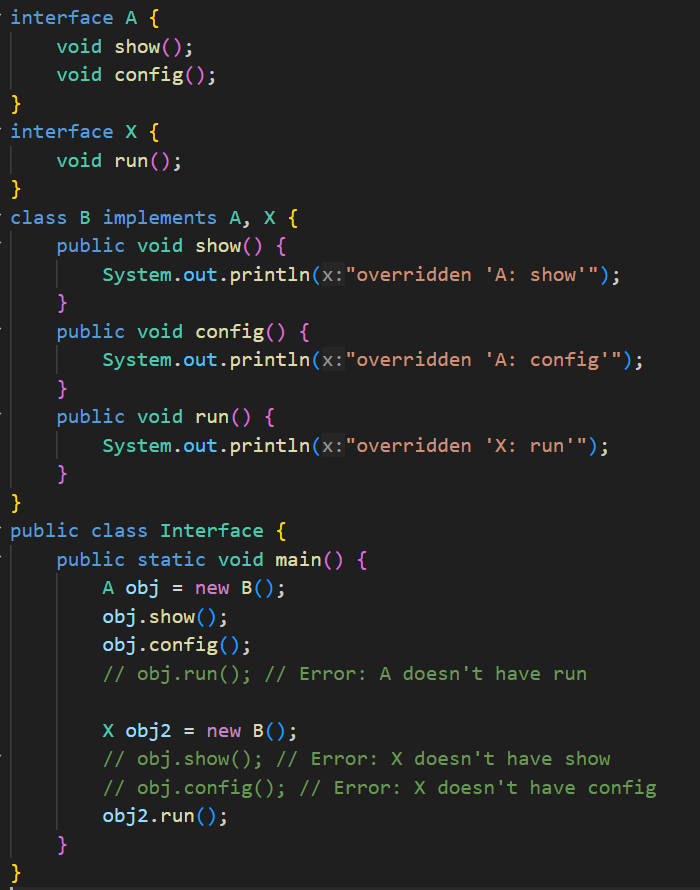
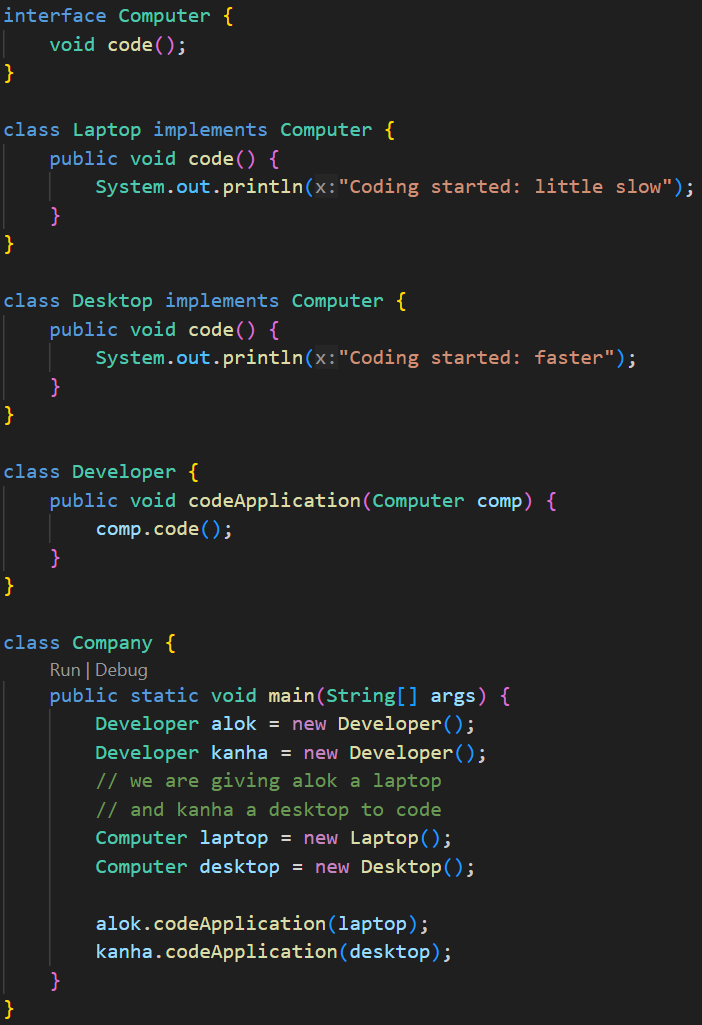
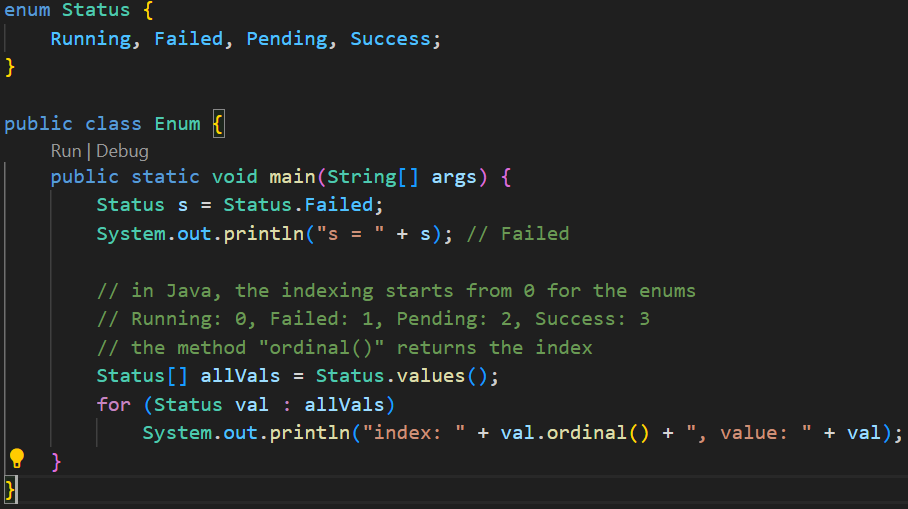
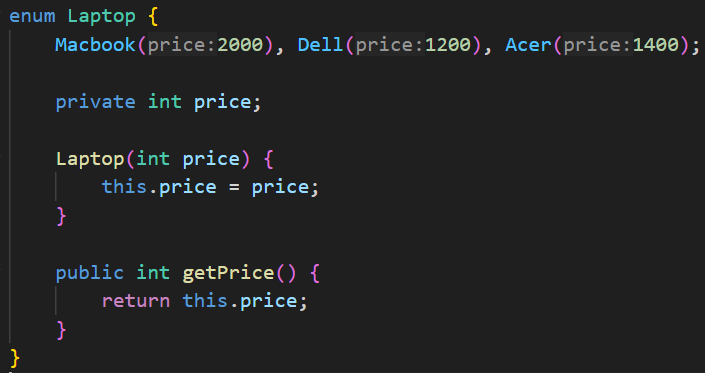
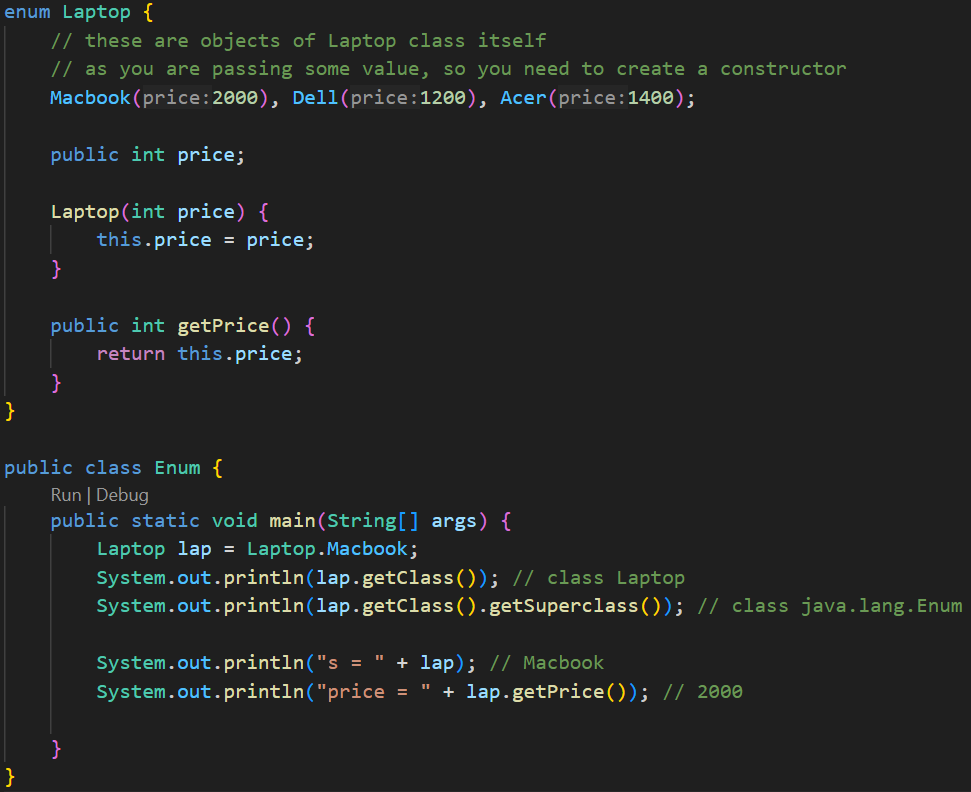
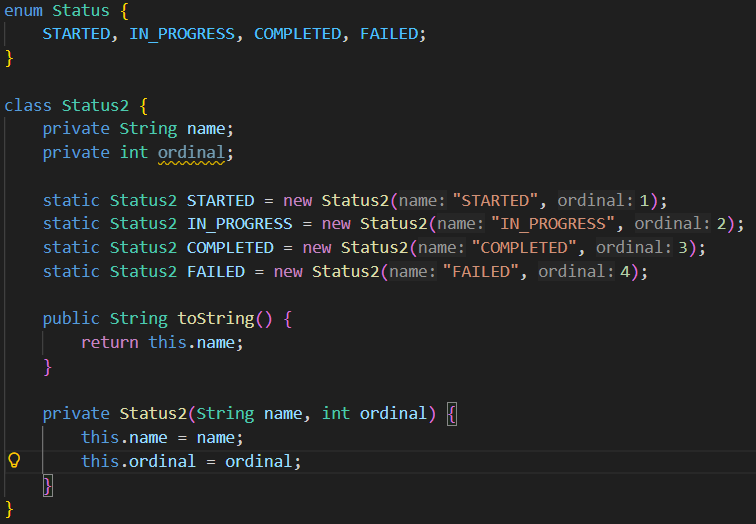
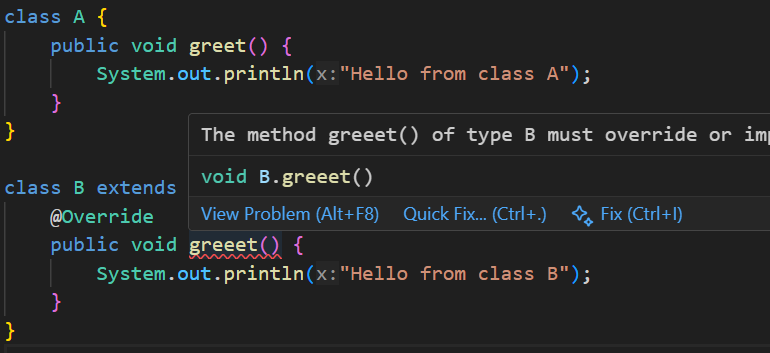
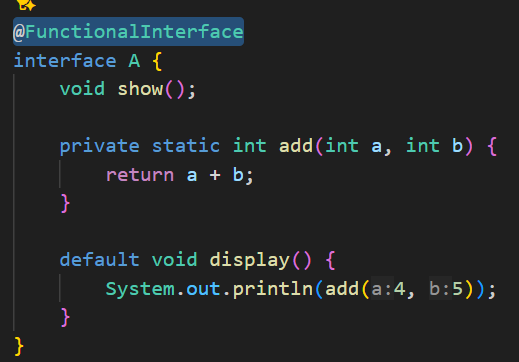
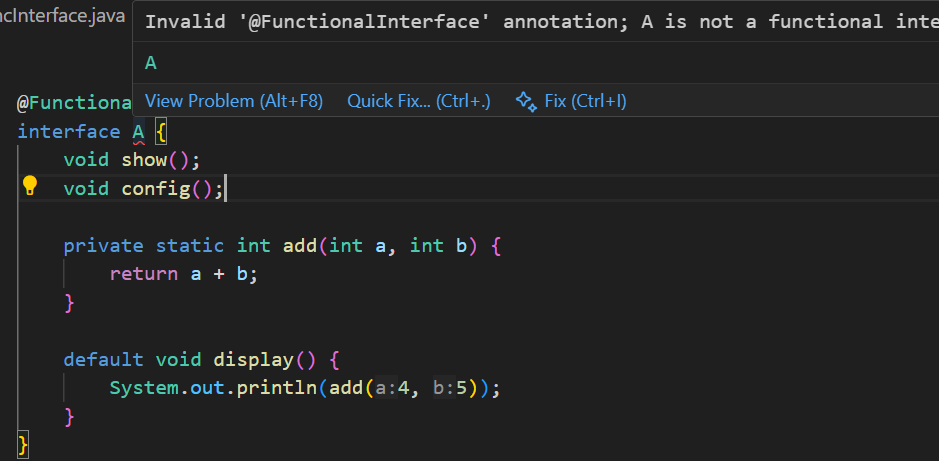
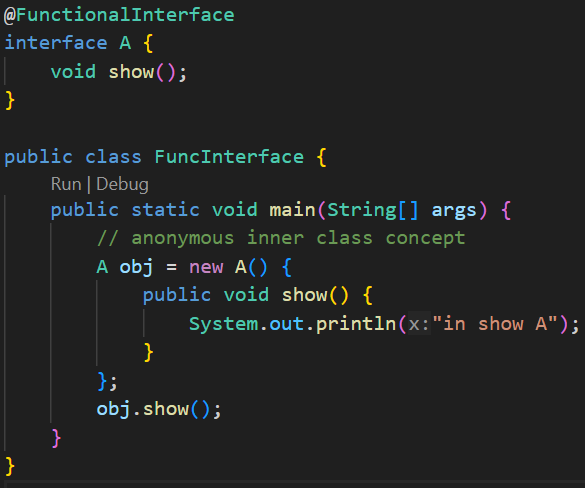
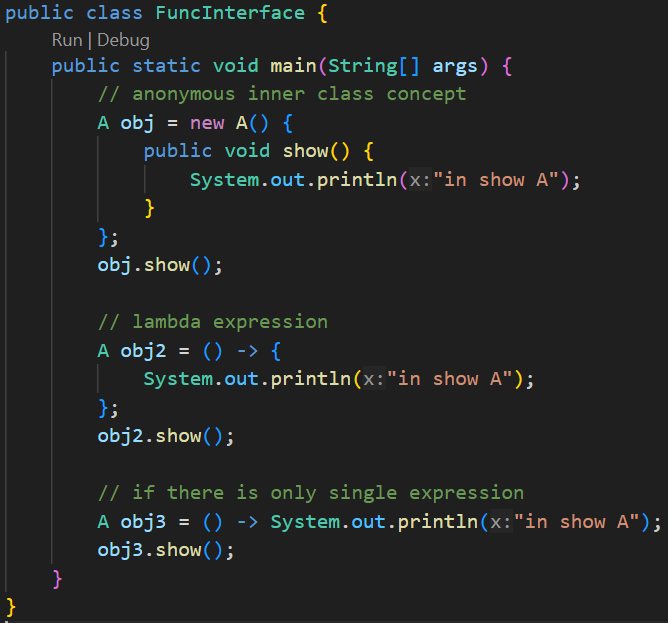
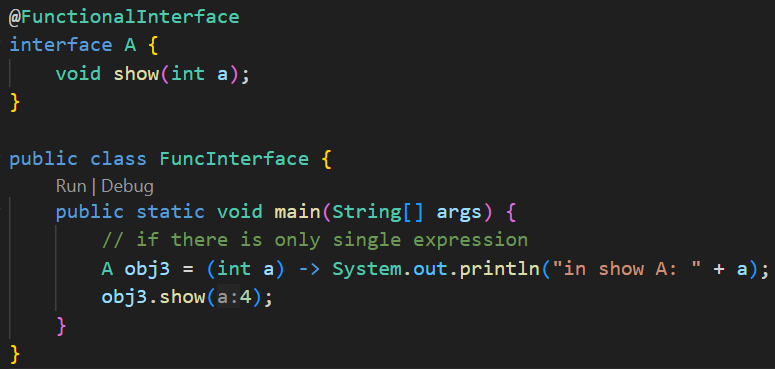
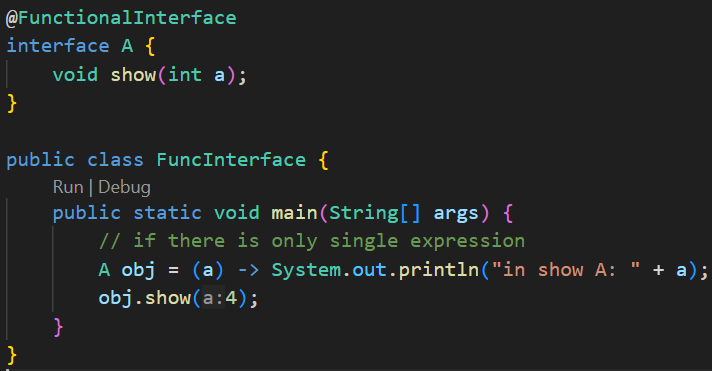
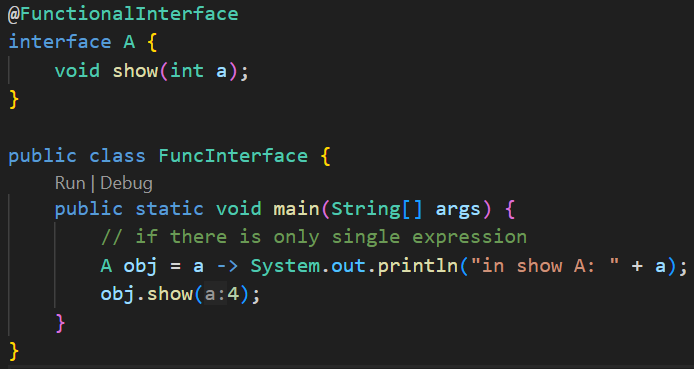
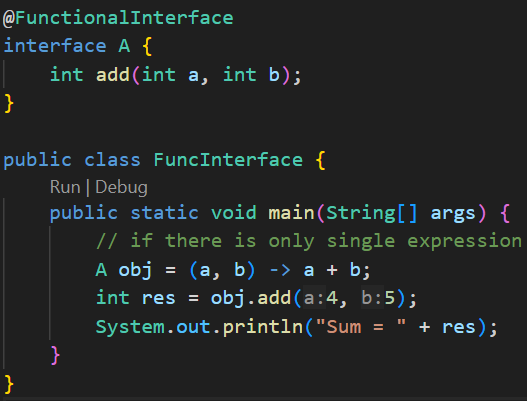
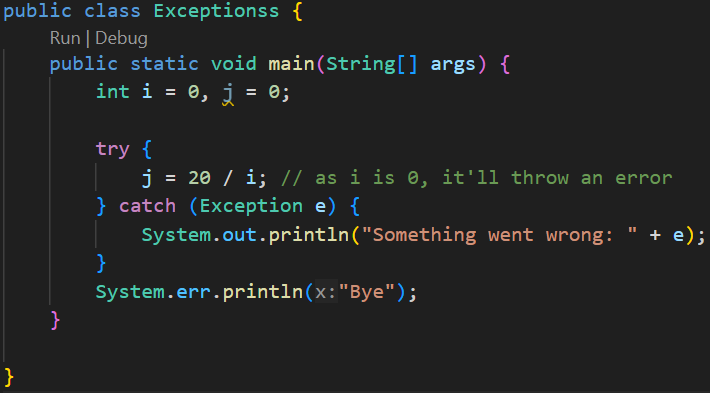
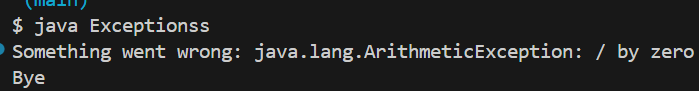
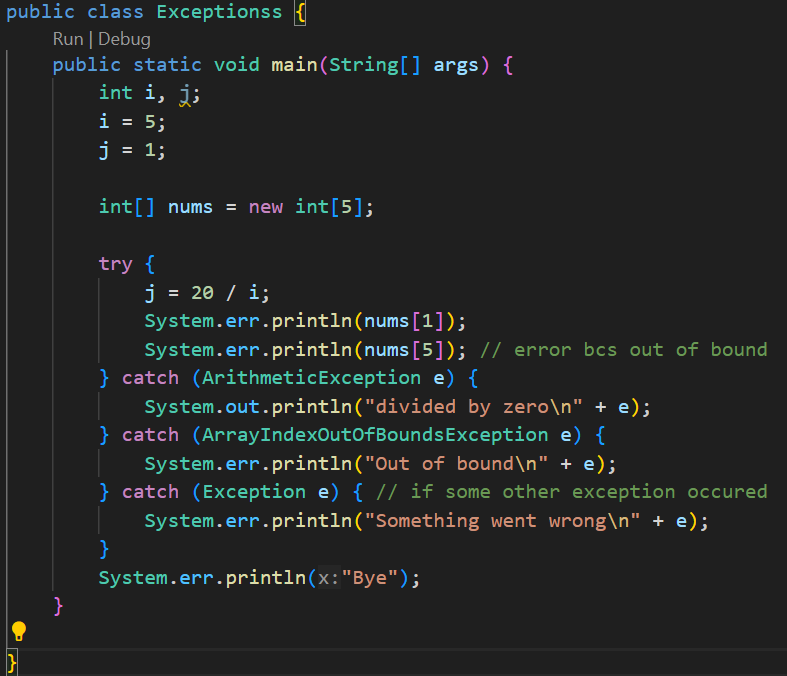
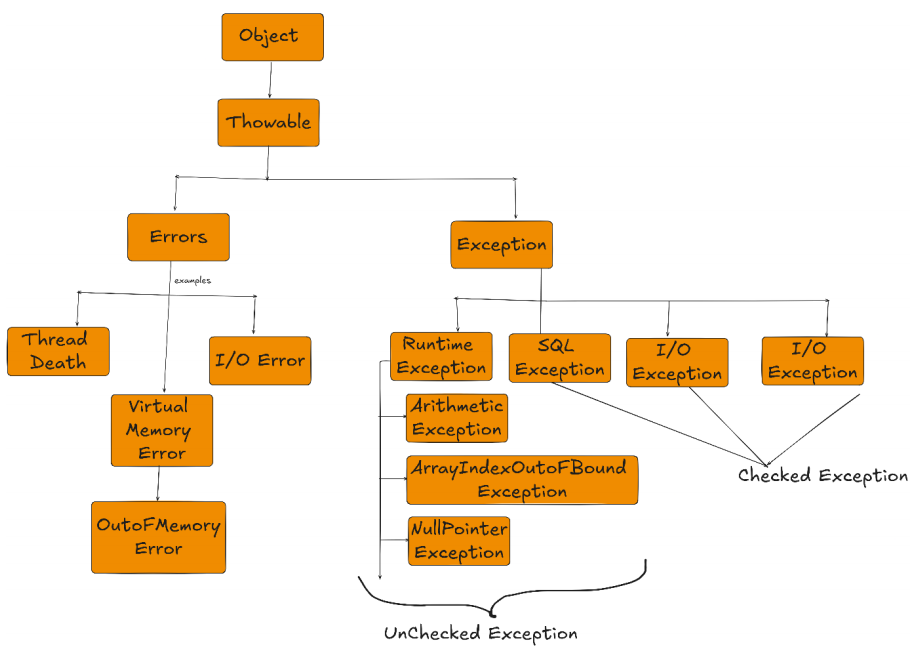
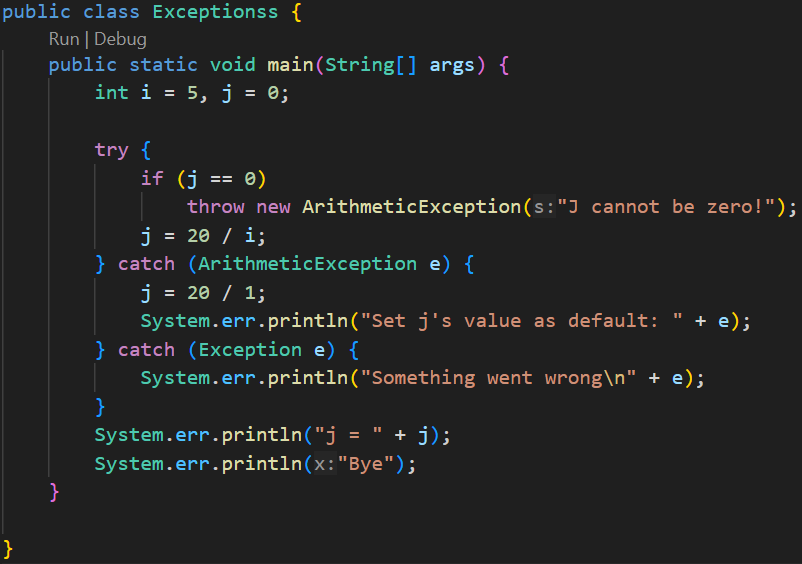
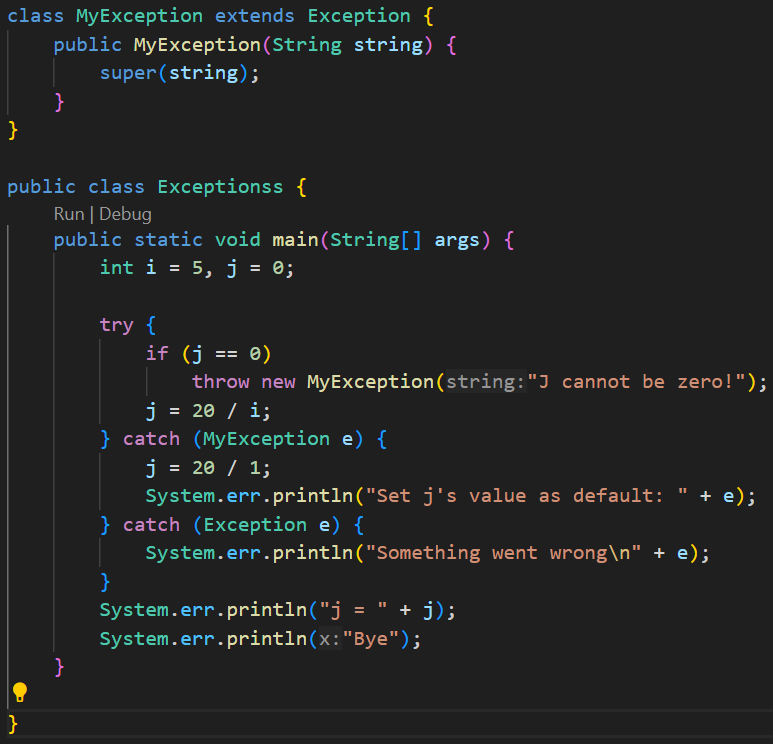
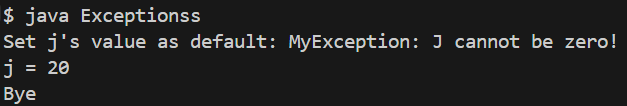
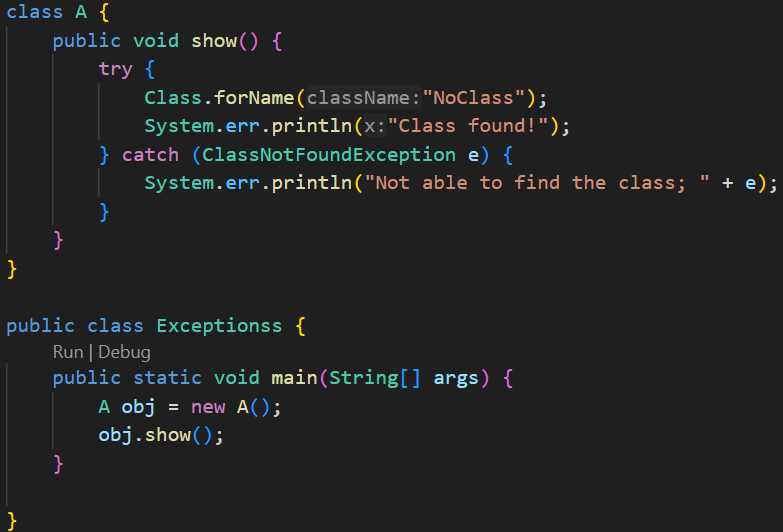
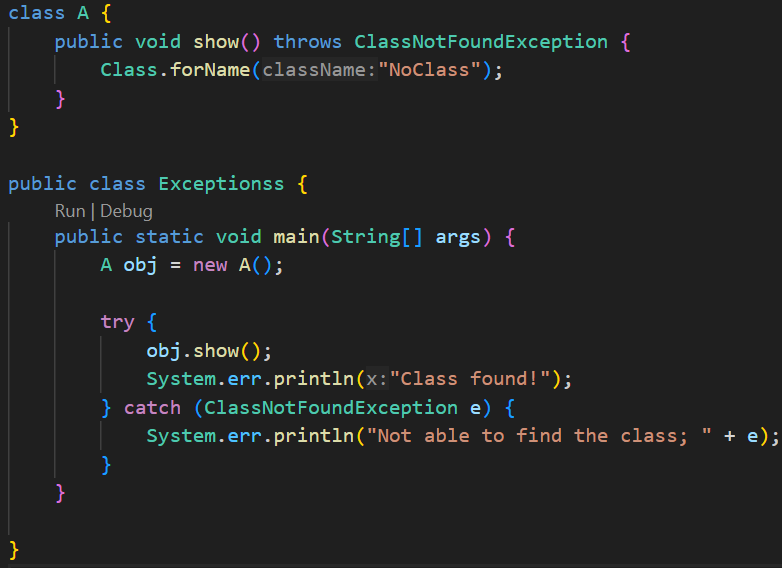
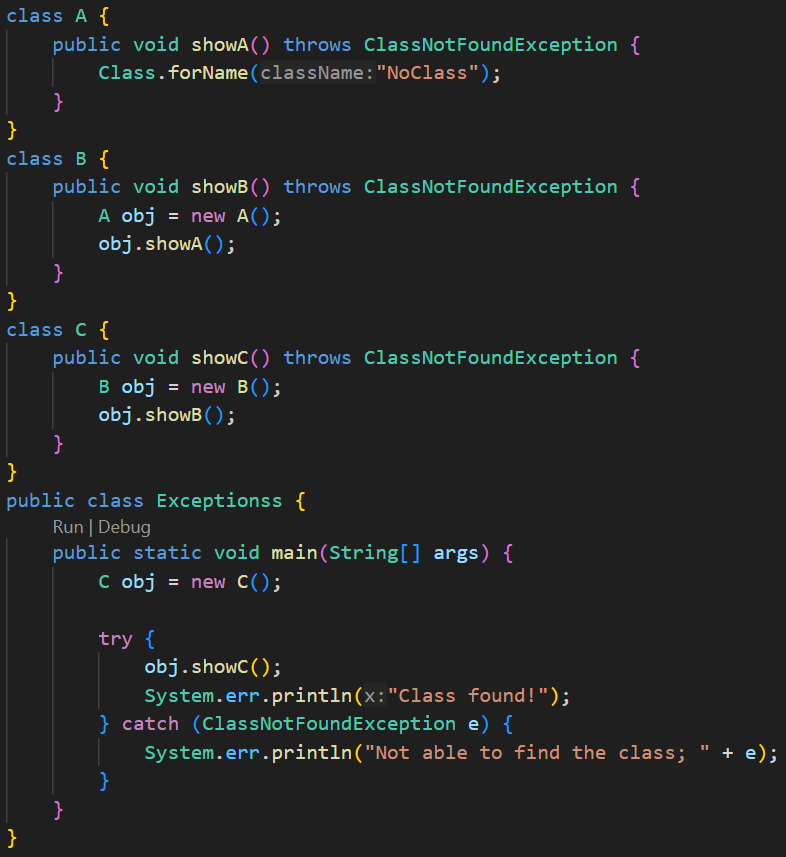
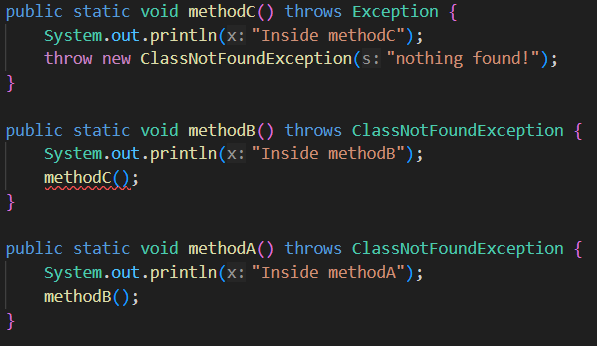
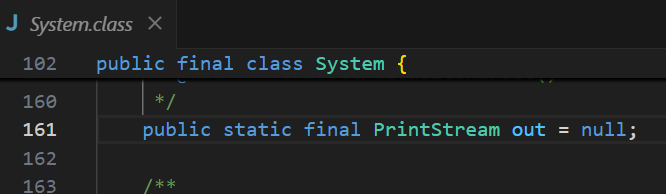
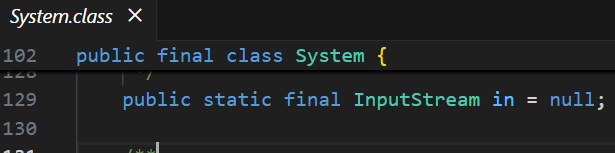
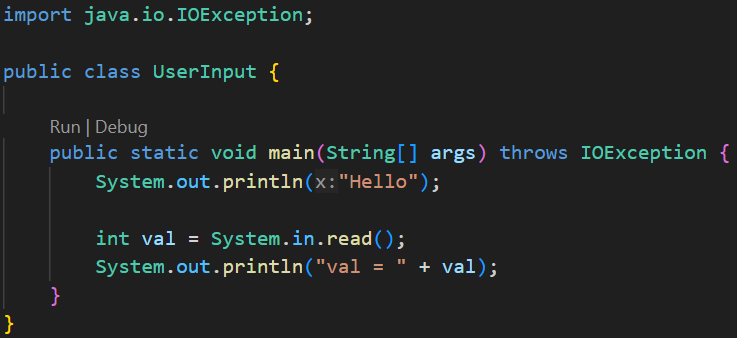
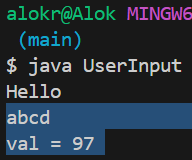
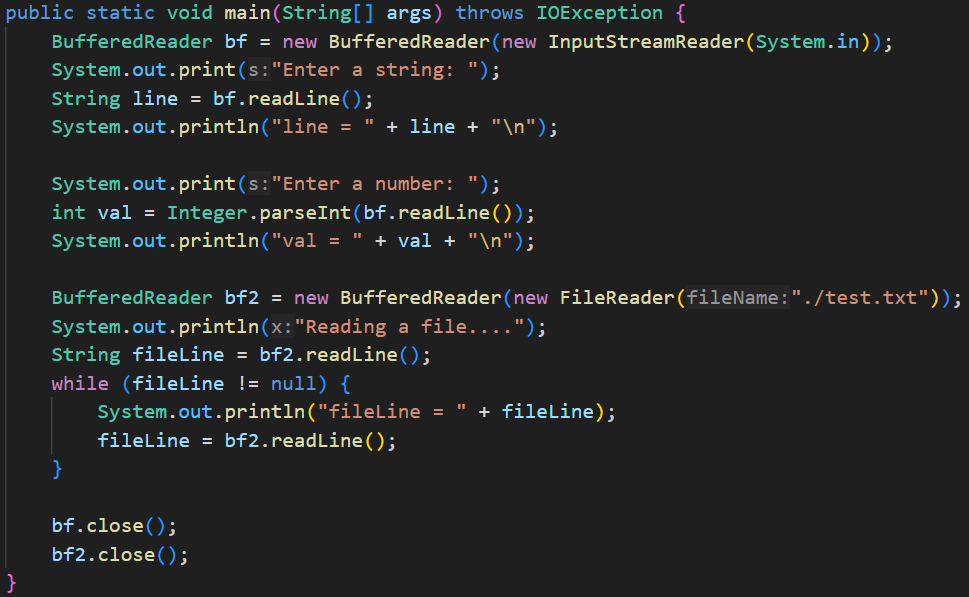
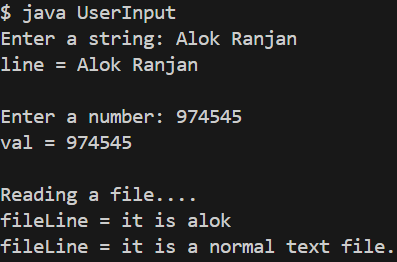
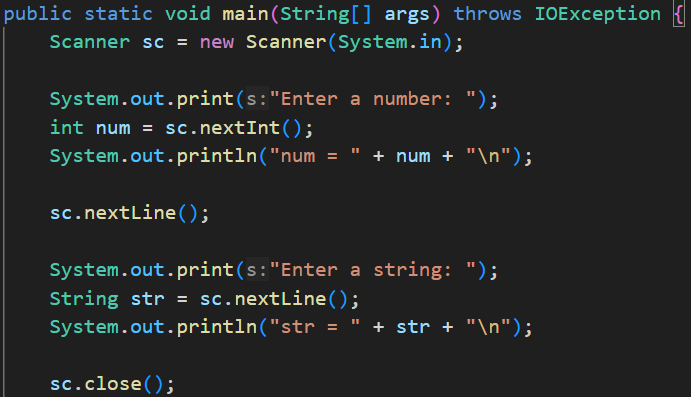
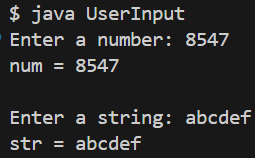
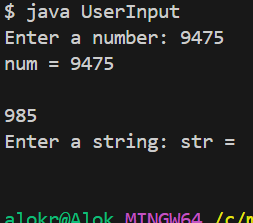
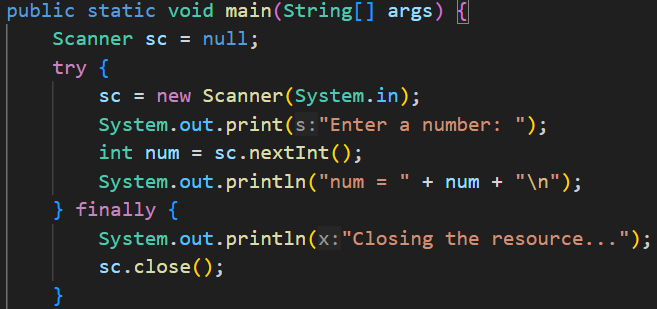
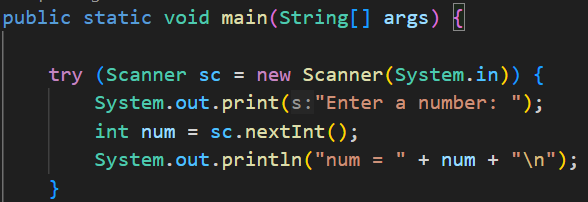
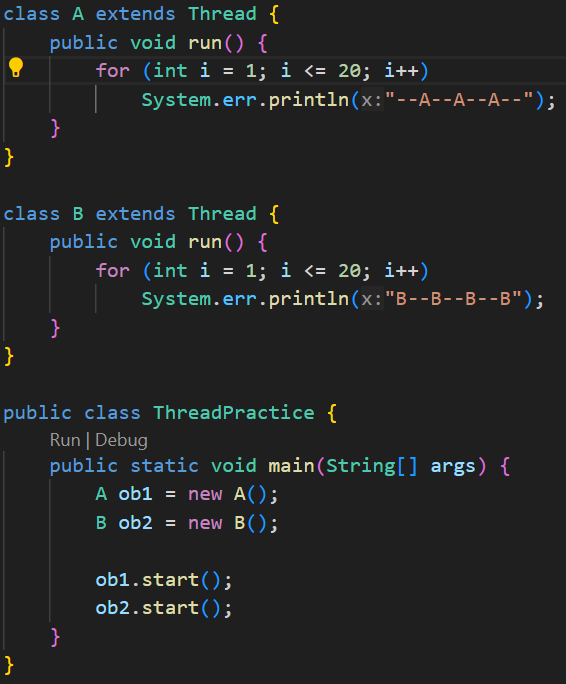
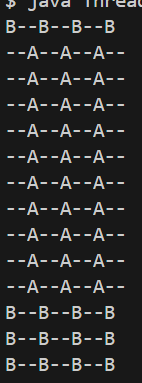
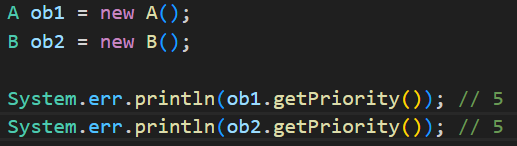
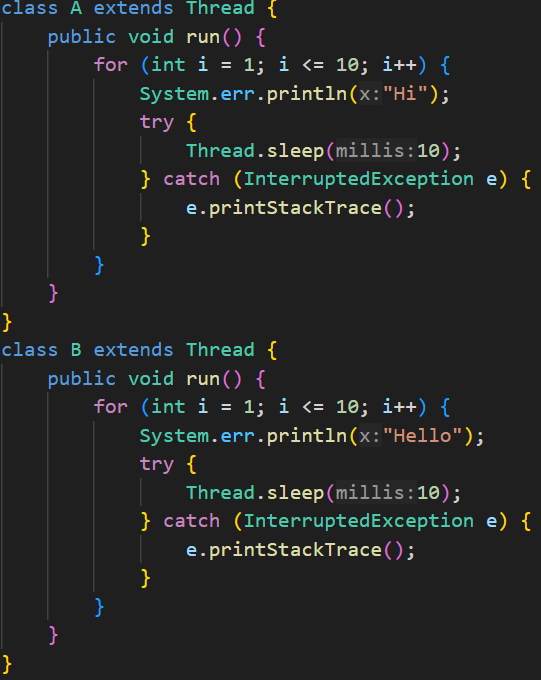
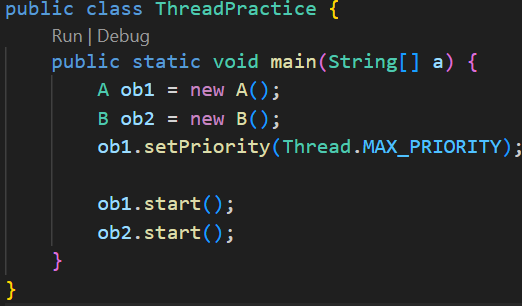
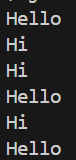
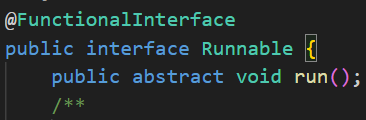
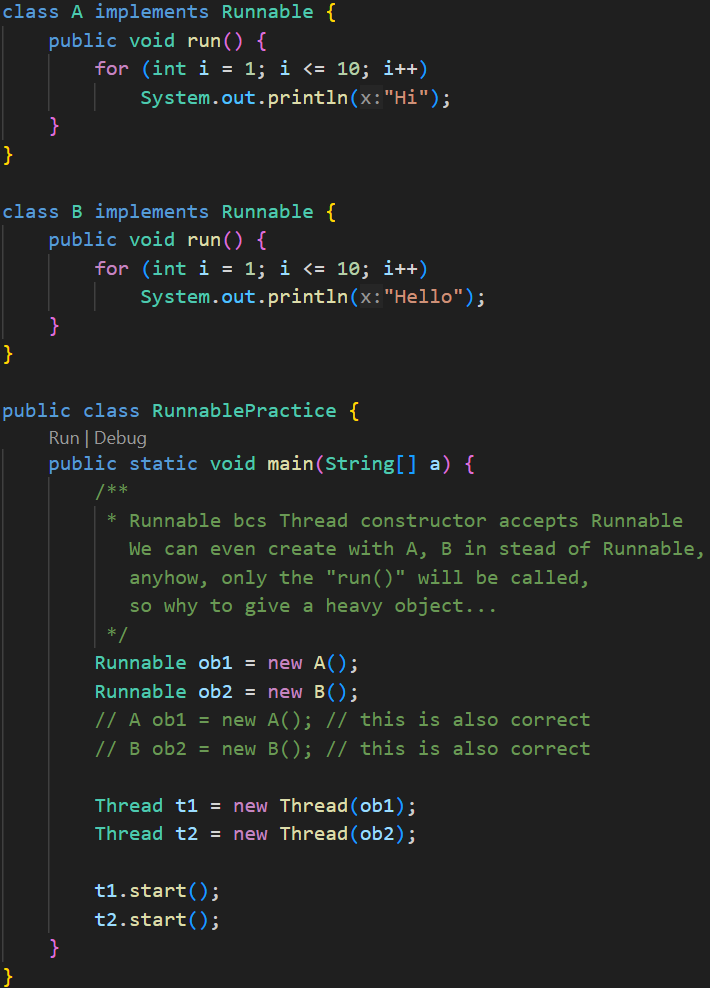
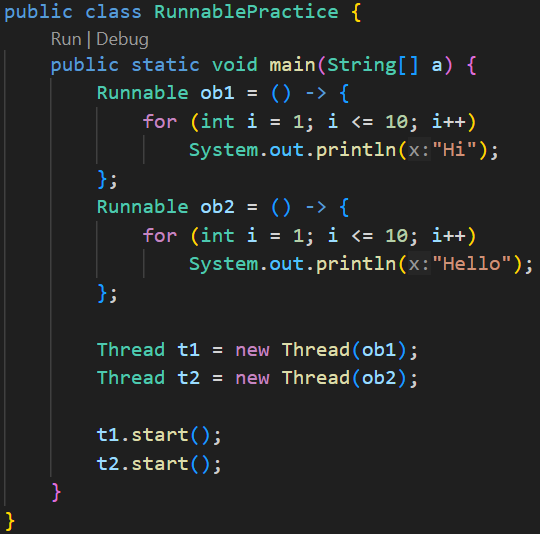
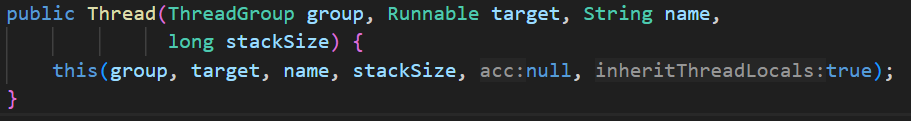
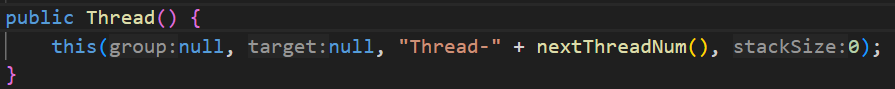
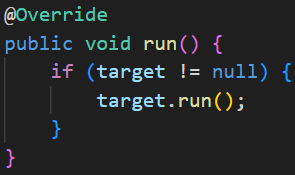
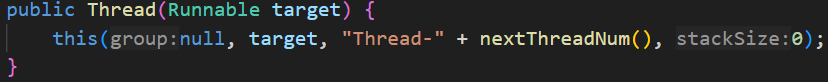
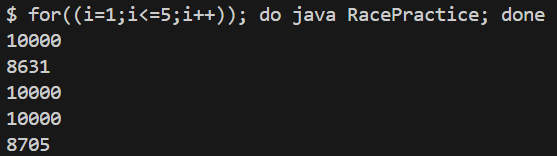
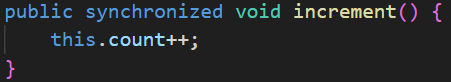
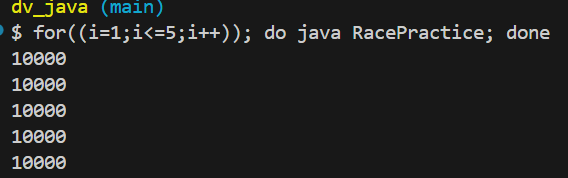
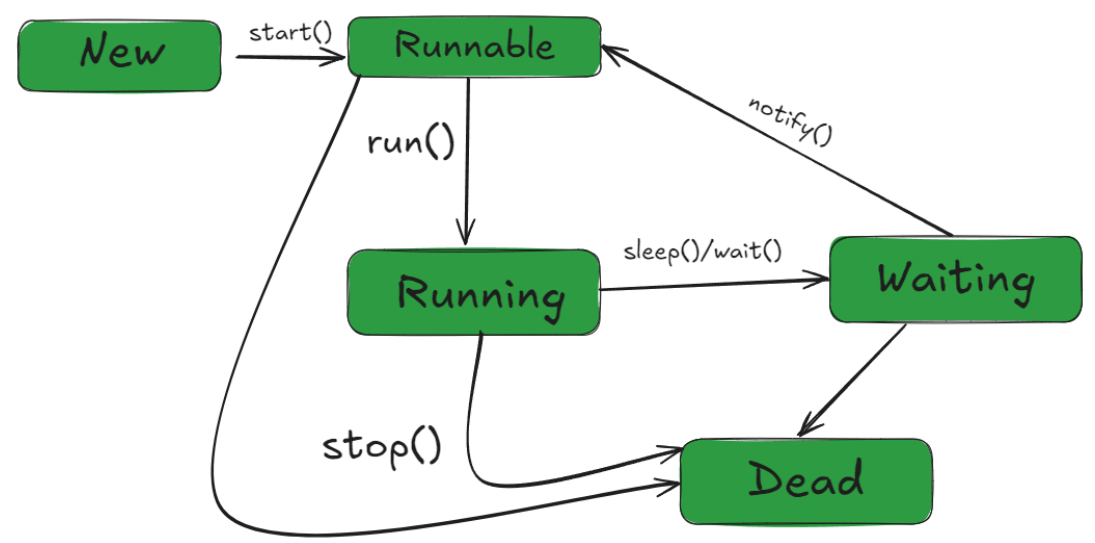
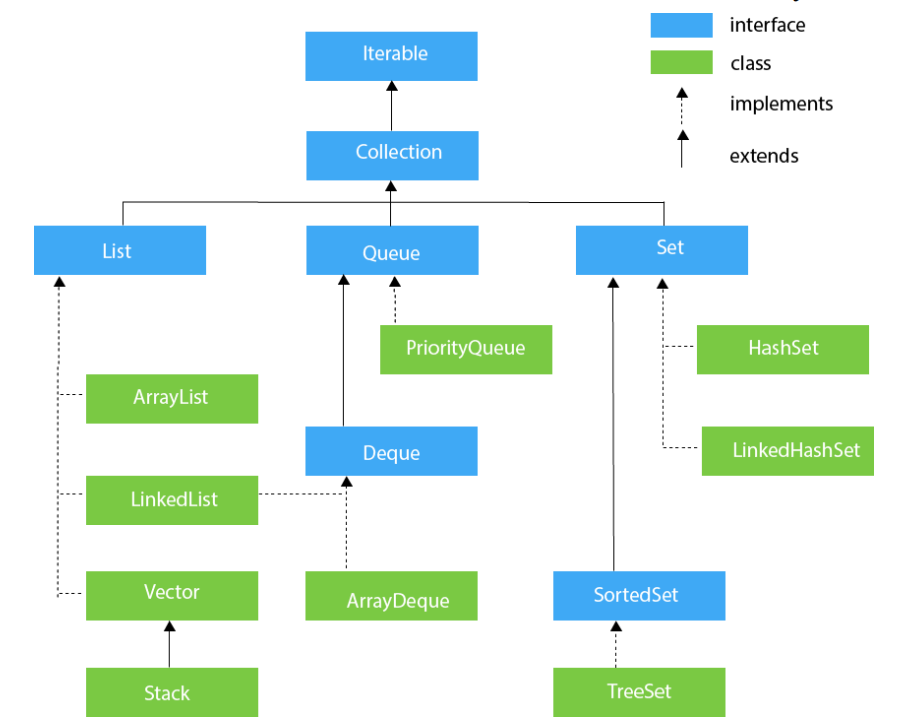
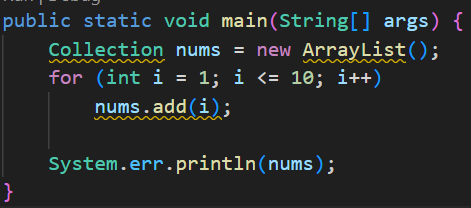
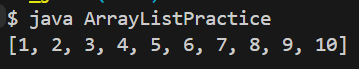
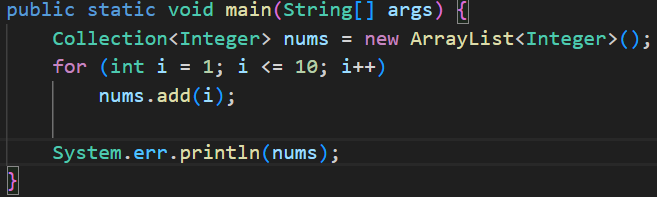
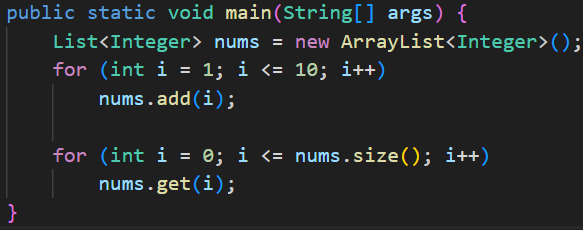
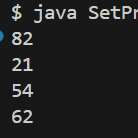
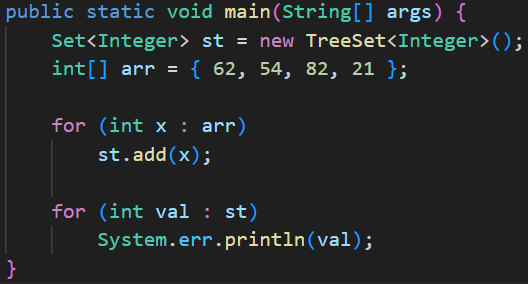
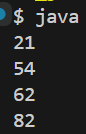
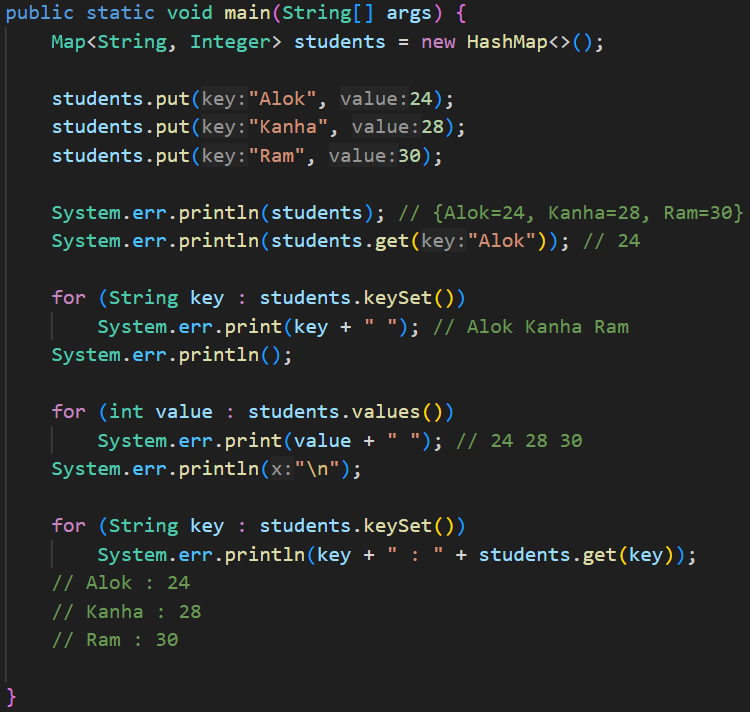
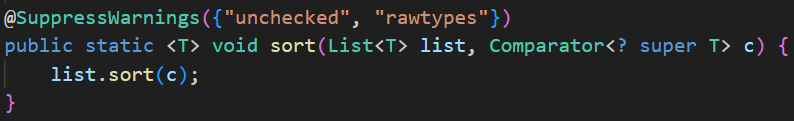
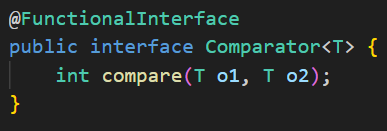
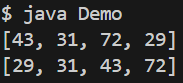
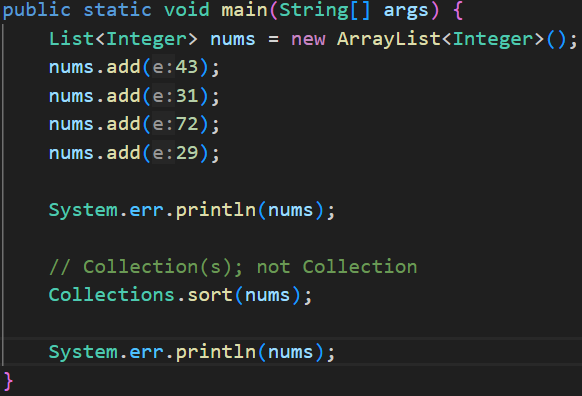
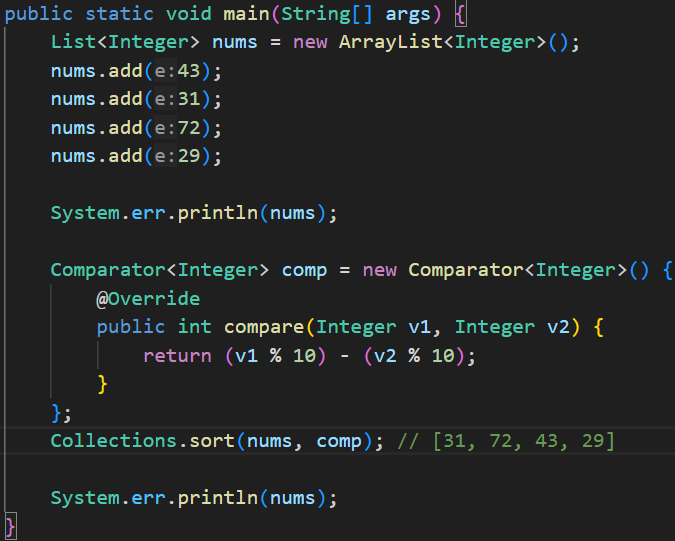
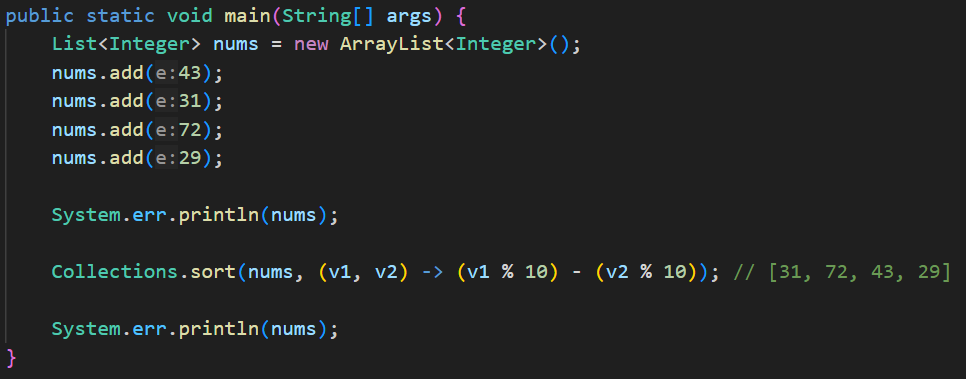
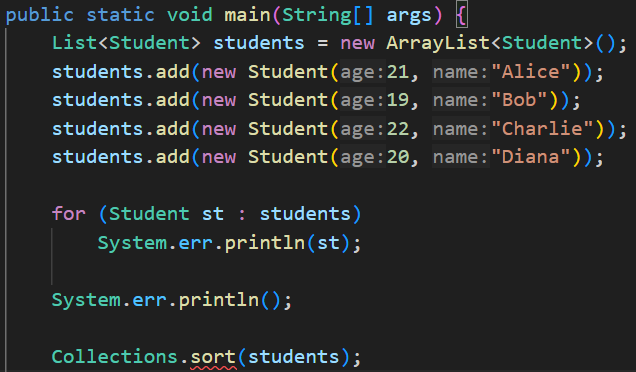
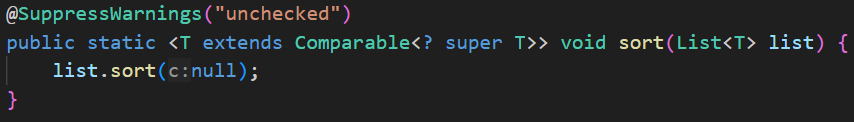
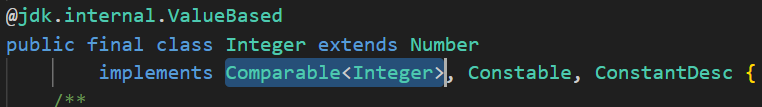
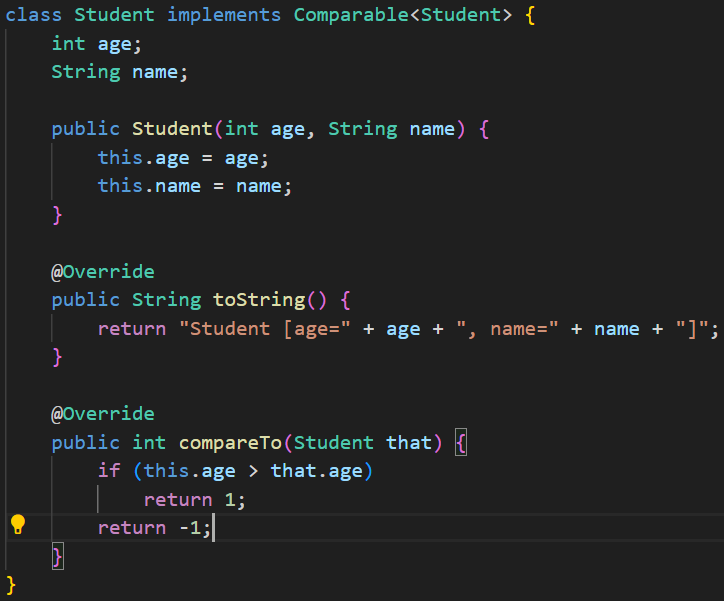
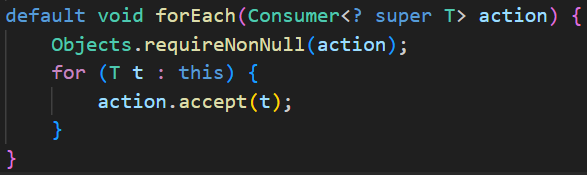
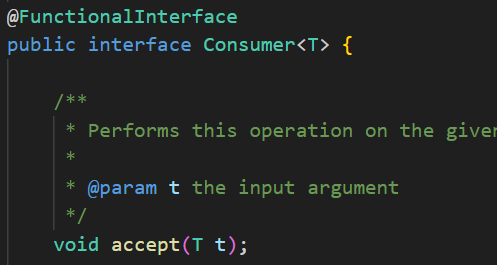
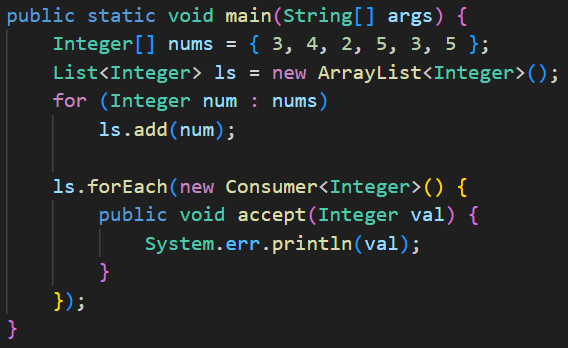
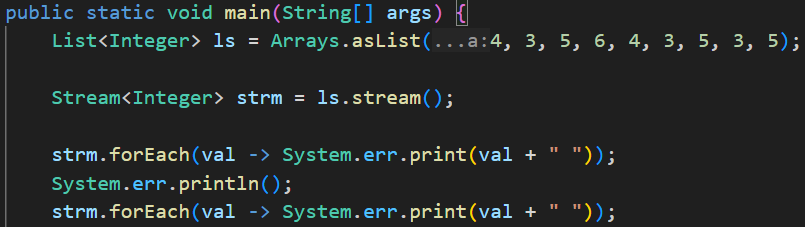
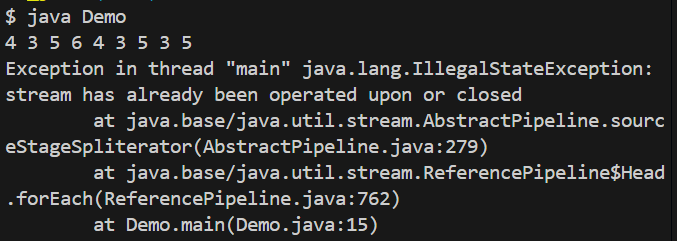
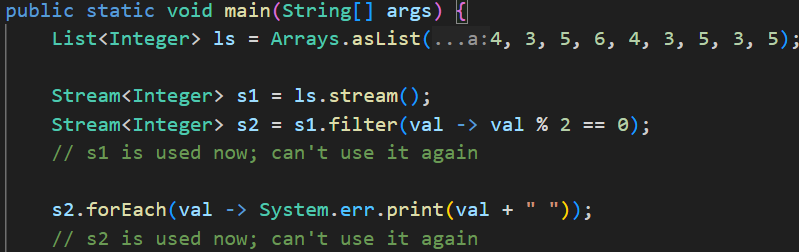
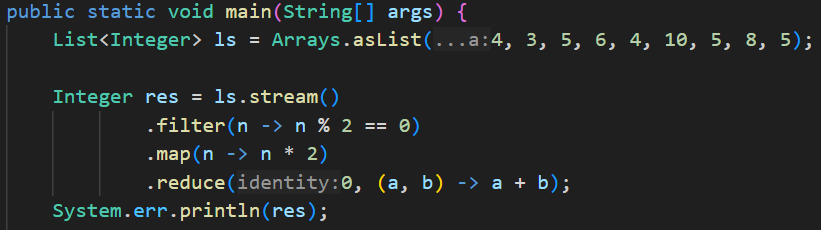
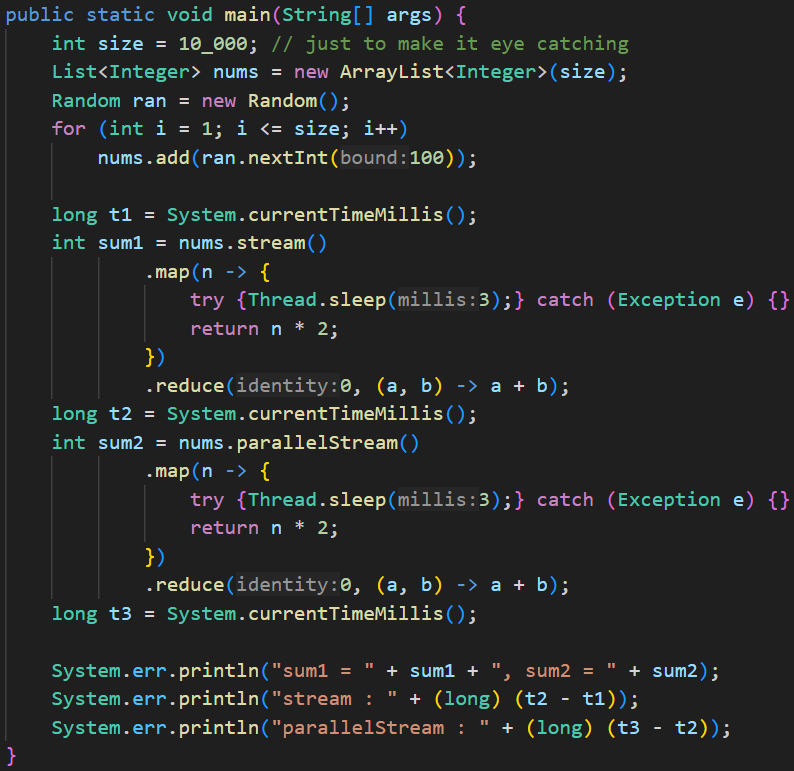
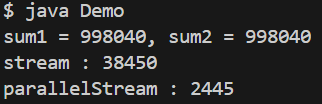
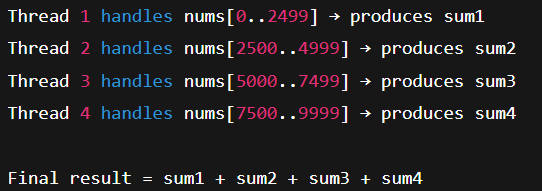
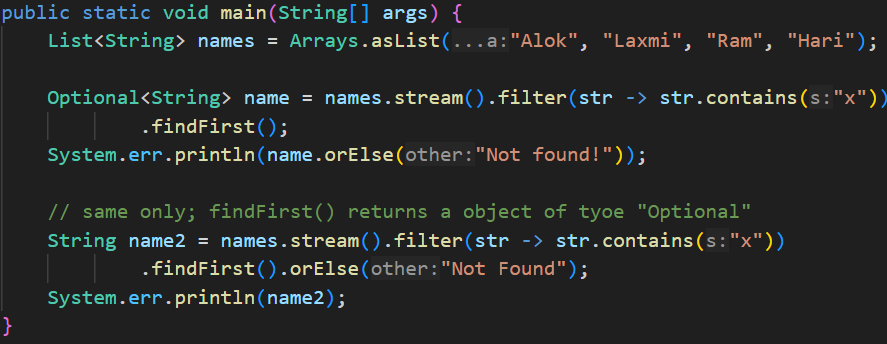
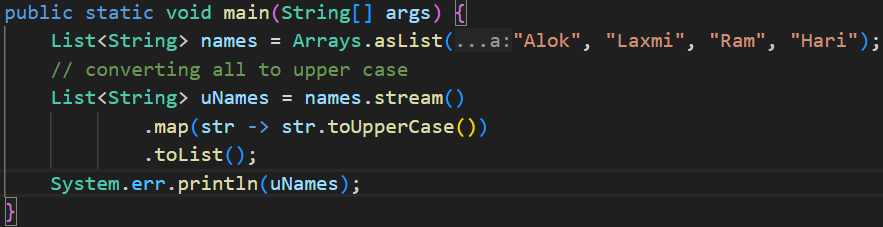
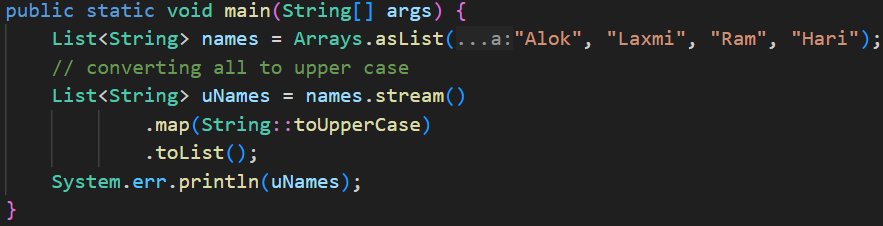
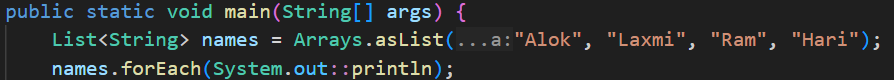
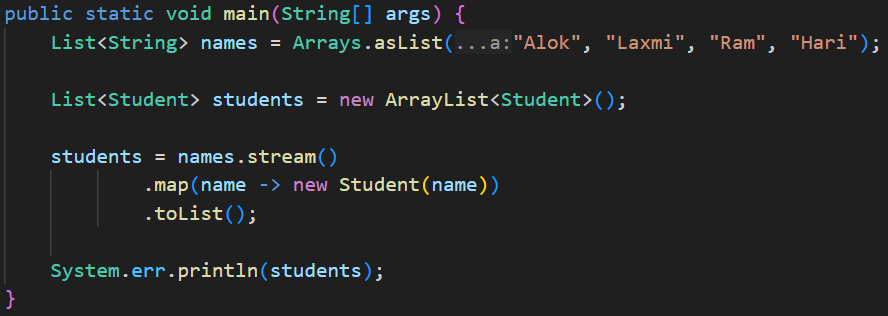
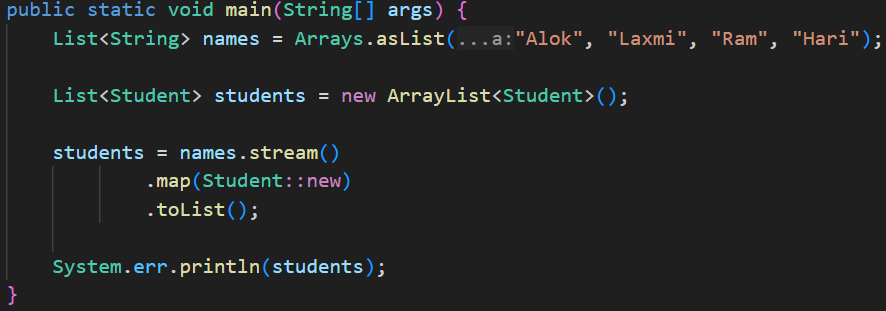
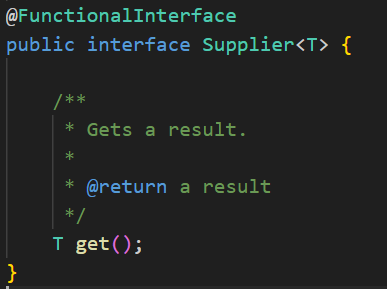
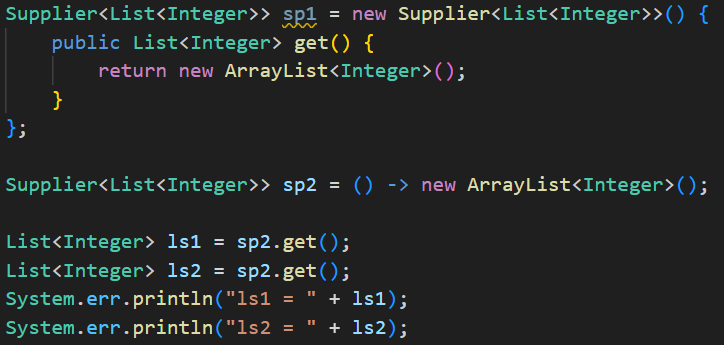
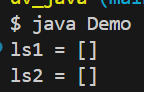
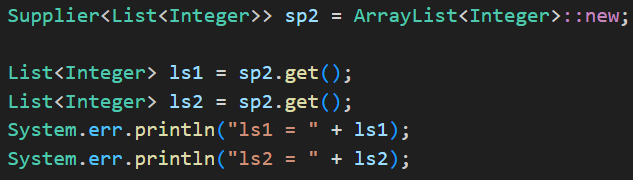
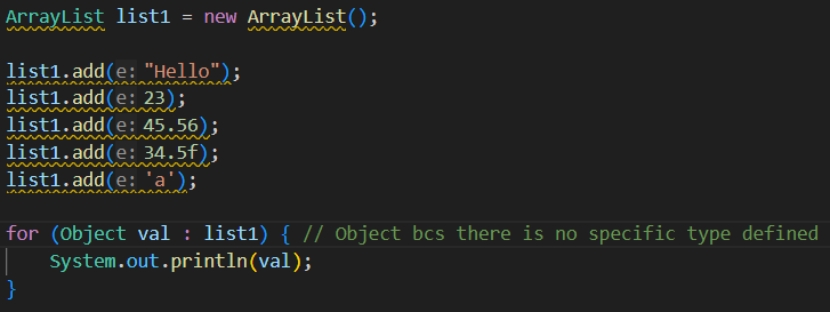
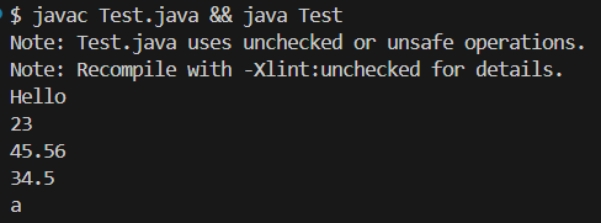
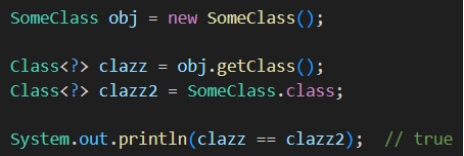
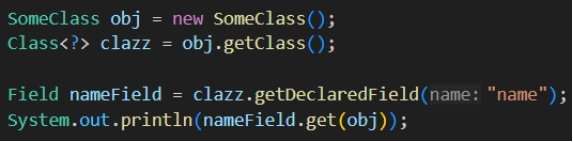
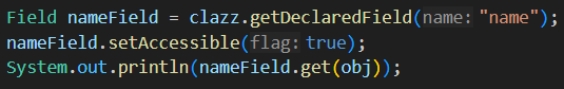
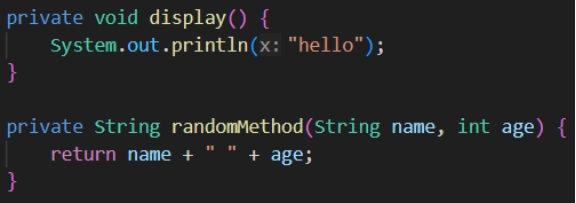
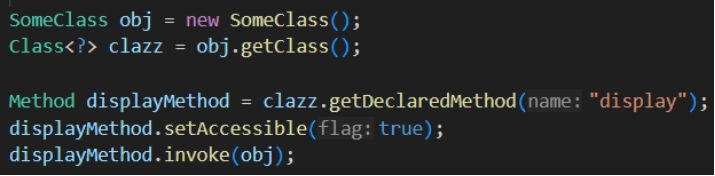
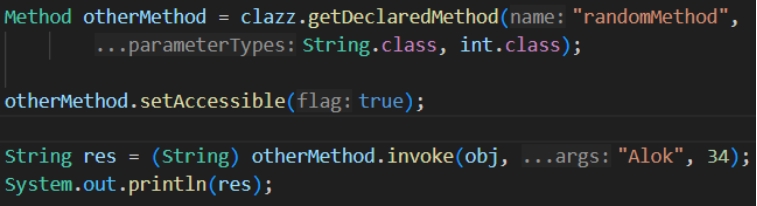
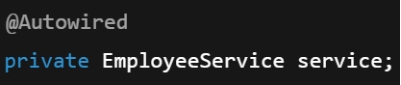
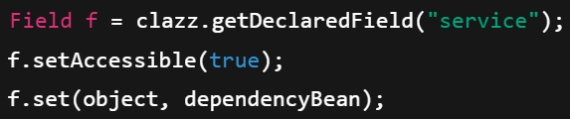
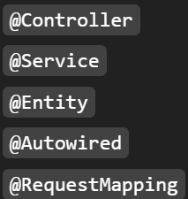
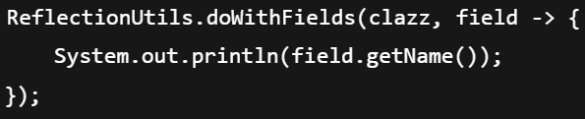
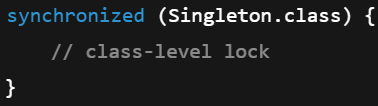
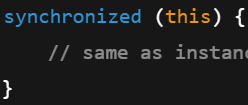
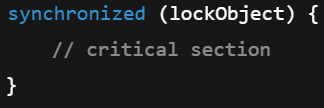
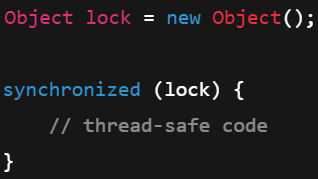
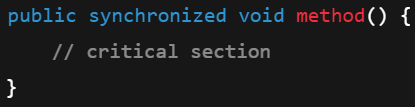
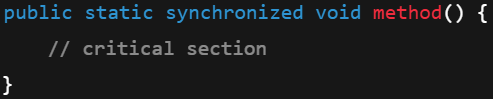
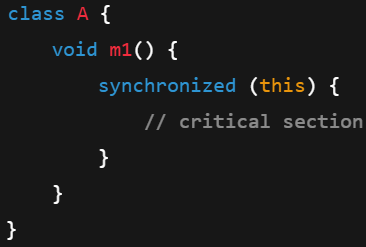
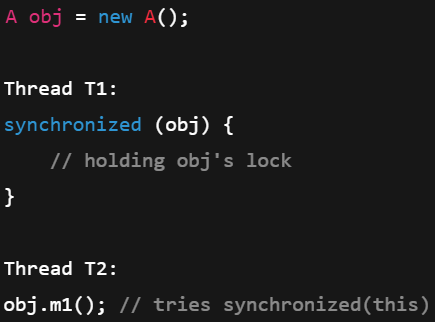
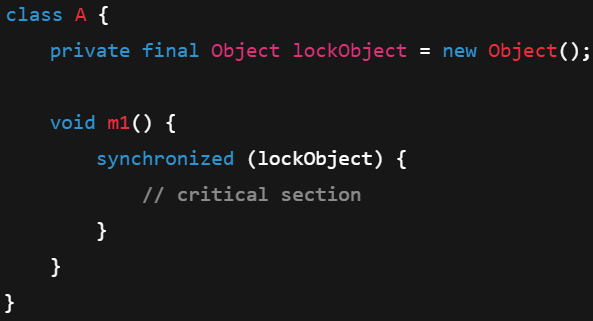
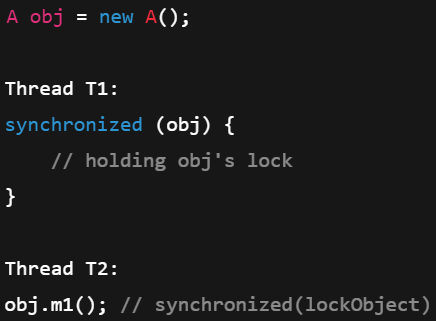
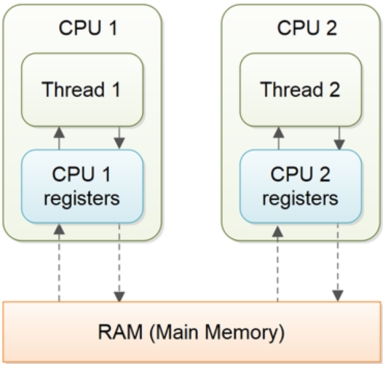
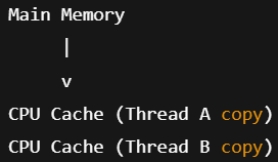
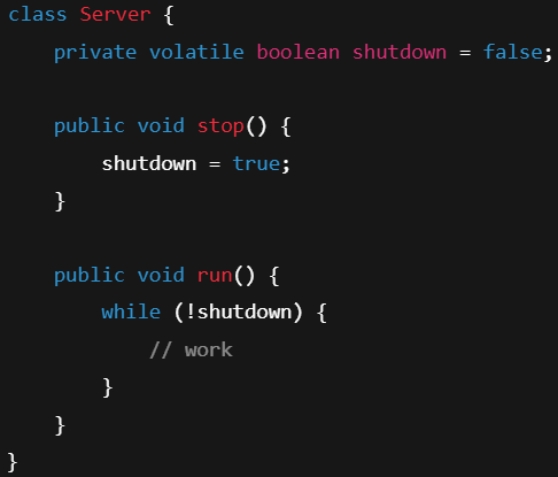
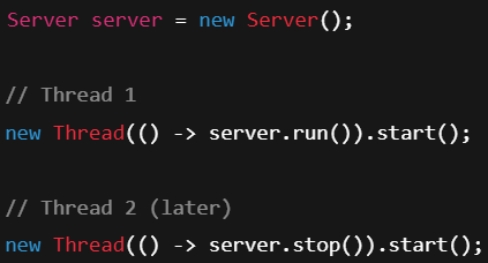
* If you try to access any method of an object, but the object is null; then you’ll get an **Null Pointer Exception**.
* **Remember: in java we initialize an array using {} curly braces; not [] square brackets.**
* Consider the following scenario:
  + I have 3 files:
    - Main.java
      * Contains 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**
      * Here we are able to do **OuterClassName.this.VariableName** because the inner class object is *created by the instance* (object) of the Outer class. So, *Inner class instance can access Outer class instance*.
      * But, if you write **InnerClassName.this.VariableName** from a method present in the Outer class then it’ll not work because the Outer class doesn’t have access to any instance of the Inner class.
    - Why not **new ob1.Inner()** ?
      * **new** keyword is used before the class name i.e. **new MyClass()** like this.
      * But if we write **new ob.Inner()** then it should be like **ob** should be one class but its an object.
      * So, we write like **ob.new Inner()**
      * In case of static inner class, we use **new Outer.Inner()** because here **Outer** is a class not an object.
  + **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())
    - Inner static class can have both instance and static variables and methods , as we can create instance of the inner class. Those instance variables can be accessed using this keyword inside the inner class.
    - 
    - 
  + **Local Inner Class:**
    - When the Inner class is defined inside a method of Outer class, then it is Local Inner Class. **(It has access to outer class’s instance variables)**
    - 
    - 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**
  + **Interface** can’t have **constructor**.
  + By default the variables inside interfaces are: **public static final**.
    - So, **static methods** can also access those variables.
    - Static methods can be called as: **InterfaceName.staticMethodName(…args)**
    - 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.
    - 
  + **enum** only supports **private constructor**.
  + 
    - Manual creation of Enum (not proper; just to understand)
* **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 super-class.”
    - 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** (written in **method** level, not **class** level)
  + **throws** required only for **checked** exception. For **unchecked** exception by default it populate back.
  + 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
  + In the following case it’ll give error.
    - 
    - Here **method** is throwing **Exception** (superclass) but **methodB** only handling **ClassNotFoundException** which is one child class of **Exception**.
* **User Input**
  + When we write **System.out.println(“…”)** it prints something in the CLI.
  + Here, **System** is a class where there is a *static variable* which is **out**.
    - 
  + And, this **out** variable is of type **PrintStream**.
    - Inside the class **PrintStream**, there is a method which is **println**,
    - This is how, **System.out.println** works.
  + Just like that **out** variable, another variable is there inside the **System** class which is **in**
    - 
    - It is of type **InputStream**
  + Inside the class **InputStream**, so many methods are there like *read, readAllBytes* etc etc.
  + 
    - As we can see, it is saying that **read** method might throw **IOException** (it is a checked exception; so it’ll give error during compilation)
  + Just to handle this temporarily, I am appending the **throws** keyword in the main method (It is not at all preferable; because if the **main** method throws the exception, it’ll go to JVM directly and the application will stop).
    - 
    - Now the compilation error gone.
    - 
      * It’ll just return the ASCII value of the first character (here ‘a’)
  + 
    -  (Output)
    - It is how we can take input from user using **BufferedReader**.
    - **BufferedReader** constructor takes an Reader type argument.
      * First case, I took **InputStreamReader**, it’ll be used to take user’s input from terminal.
      * Second case, I took **FileReader**, it’ll be used to read a file.
    - Here **bf** and **bf2** (BufferedReader instance) are resources. So whenever you create these, you have to close it as well.
      * It’ll not give any error, but it is a good idea to close the resources.
  + 
    -  (Output)
    - Here you must be thinking why we have written **sc.nextLine()** in between.
      * When you give input and hit *Enter*, the next **sc.nextLine()** will take that as its input.
      * So, you can’t take the input for the string here because it’ll take that ***\n*** *(Enter)* as its input.
      *  (it would have occurred without that middle **sc.nextLine()**)
* **Try with Resources**
  + There is a keyword **finally**; this block executes even if the Exception occurred (catch) or not (try).
  + Even you can just run **try** and **finally** without **catch**.
  + The finally block is mostly used to *close the resources*.
  + Without this **finally** block, we would have to close the resources on both **try** and **catch** blocks.
  + 
    - This is how we can use the finally block to close the resources.
  + 
    - It is a short syntax.
    - Here, after the try block is completed, the resource will be closed automatically.
    - 
    - 
    - You can see, the Scanner class’s ancestor is the AutoClosable interface, so it’ll be automatically closed.
* **Threads**
  + There is a class **Thread** in java, which has a method called **start()**.
    - This **start()** method call a method whose name is **run()**.
    - So, if you want to run a method in a thread, then you need to give **run** as the method name.
    - Below is the example of threads:
      *  
        + Output is not continuous like --A--, --B--, --A--, --B-- like this
    - If your CPU has **n** cores, then **n** threads can be run at a same time.
      * In modern systems, *1* core may be able to run *2 or more*threads at a same time.
    - 
      * The priority range is from **0 to 10**.
        + **0** is least priority and **10** is highest priority.
      * To set the priority, we can use **setPriority** method.
    -  
      * (System.out, not System.err (minor mistake :))
      * Here, I gave some sleep to make the output alternate Hi, Hello, Hi, Hello.. like this.
      *  (you can just optimize it; how it’ll work can’t control)
      * In above case, might be both the run came to the scheduler to get executed after their respective sleep of 10 milliseconds (mentioned in code), then scheduler might have given someone.
* **Runnable vs Thread**
  + It is not a good idea to inherit the **Thread** class to make a thread.
  + Because, if the class has to inherit some other class, then it can’t be done in this case.
  + 
    - The **Thread** class implements an *functional* interface **Runnable**, and the **run** method is present inside the **Runnable** interface only.
      * 
  + 
    - It is how **Runnable** and **Thread** work.
  + (using lambda expression)
* **Some subtle links between Thread and Runnable**
  + 
    - This is the main constructor of the Thread class.
    - **target** is of **Runnable** type.
  + When we extend the Thread class by any custom class, by default the default constructor of the Thread class (non-parameterized constructor) gets called.
    - 
    - It is the default constructor of Thread class.
    - Here we can see, the target is null.
  + So, when we extends Thread class from our class, the **target** is **null**.
  + Also there is a **run()** method inside the Thread class which overrides the **run()** method of the interface **Runnable**.
    - 
    - In case of extending Thread class, we override this **run()** method, so that our **run()** method (present in our class) will get executed.
  + One more constructor inside Thread is there which accepts **target** .
    - 
    - So, if we are not extending the class, we need to pass a **Runnable** type object inside the Thread constructor while initializing.
    - Our custom class can implement the **Runnable** interface and that object can be passed inside the **Thread** class’s constructor.
  + In simple words:
    - **start()** method will trigger the **run()** method of **Thread** class.
    - In case of inheriting, the **run()** method of **Thread** class is overridden by our own class. So, our **run()** method gets executed.
      * Runnable’s run() --- Thread’s run() --- CustomClass’s run()
      * This is the **overriding** hierarchy.
    - In case of implementing runnable, the **run()** method of **Thread** class doesn’t get overridden, so the **start()** method will call **run()** method of **Thread** class as it is. As this **run()** method is running **target.run()**, i.e. its running the **run()** method of the **Runnable**, (and our own class has overridden the **run()** method of runnable), so our **run()** method gets executed.
      * Runnable’s run() --- CustomClass’s run()
        + These are same as **Thread’s target.run**
* **Race Condition**
  + When 2 threads are running, they should not modify one variable at the same time.
  + Like imagine transacting to 2 different persons from the same bank account at the same time, it’ll cause issues.
  + When you start the threads in side a method, the method doesn’t stop there and execute the remaining code after starting the thread.
    - If you want to execute the statements after the threads are complete, then use **join()** method.
  + 
    - This code should give the output *10000*, but the output will not be consistent.
    - 
    - I ran for 5 times, the results are inconsistent.
  + It is happening because, at sometimes, both the threads are executing the **increment()** method at same time;
    - Lets value of count was *100* at a time, both executed *increment()* method at that time.
    - So, now, instead of 102, the value of count became 101.
    - This is the cause of the inconsistent result.
  + There is a keyword called **synchronized**, it doesn’t allow the method to be called 2 times at once.
    - 
    - 
    - Now, the result is consistent.
* **Thread States**
  + 
* **Collection API**
  + Collection API : concept
  + Collection : interface
  + Collections : class
  + 
* **ArrayList**
  + 
    - Here, you can directly print the object.
    - 
    - Here you can’t access the values like **nums[2]**
    - You can use **nums.get(2)** to get the value at index-2.
    - In the above code, we have taken **nums** as **Collection** type. In that *Collection* interface, there is no method called **get()**.
    - The type of nums should be **List** or **ArrayList** or something like that.
  + 
    - The warning that was coming before, was due to not specifying the type.
    - Note: here you can use Wrapper type (Integer, Double etc.) not primitive type (int, double etc).
  + 
    - Now we used **List** in place of **Collection**, so we can use the **get()** method now.
* **Set**
  + Whenever you add some values in a set, sometimes it looks like sorted; but it is **concidence**.
    - Internally, it stores elements in a hash table based on the element’s hash code.
    - The iteration order you see depends on how the hash table buckets are organized.
  +  
    - **Set** is interface; **HashSet** is class
  + If you want a sorted set, then go for **TreeSet** instead of **HashSet**
    -  
* **Map**
  + 
    - The **HashMap** is not **synchronized**.
    - To use **synchronized** version, use **HashTable**.
* **Comparator vs Comparable**
  + The **Collections** class contains **static methods** that operate or returns collections.
  + 
    - This is the **sort()** method.
    - Here **Comparator** is there; Comparator is nothing but a *Functional Interface* containing a **compare()** method.
    - 
  + In the below code snippet, the list nums will be sorted according to its values.
    - 
    - 
  + I want to sort the values by their right most digit:
    - 
    - 
      * Using lambda expression.
  + 
    - Here, in case of a list of objects, **sort()** method is not working without passing the **comparator**.
    - 
      * It is the **sort()** method, it requires that the Type (i.e. Integer, Double etc etc) should be of **Comparable**.
      * 
        + You can see, Integer is implementing **Comparable**; so it can be used in the **sort()** method without passing the *comparator*.
    - 
      * Now, we implemented the **Comparable** interface in the **Student** class, and overridden the **compareTo** method inside it.
      * Now, when we sort without using **comparator**, it’ll not give any error.
* **forEach method**
  + 
    - It accepts a Consumer typed object.
    - **Consumer** is a **functional interface**.
    - **<? super T>** means, type should be T or any of its super class.
  + 
    - This the abstract method that is to be overridden.
  + 
    - It can also be written like this.
* **Stream API**
  + You’ll get a method in the **List** objects called **stream()**.
  + It’ll create a **Stream** object containing all the methods that **List** has.
  + Even if you change something inside that **Stream** object, nothing will affect the original list.
  + But the condition is: the Stream object can be used **once**. If you use it more than once, it’ll give **Run time exception**.
  + 
    - 
    - Run-Time exception. Only once the **Stream** object can be used.
  + Using of stream provides so many methods like **map**, **filter**, **flatMap**, **sorted**, ..etc. These are part of **Stream API**, not part of the List or something like that.
  + 
  + 
    - As the return types of these **stream api’s methods** are Stream only, so we can chain these kind of methods.
* **parallelStream in Java**
  + 
    - 
    - parallelStream ran faster (as multiple threads)
    - (it works like this)
* **Optional class**
  + It’s a class just to avoid the **Null Pointer Exception**.
  + 
* **Method Reference**
  + 
    - Here, in the **lambda** expression passed inside the **map()** method, we are just calling a **method** i.e. **toUpperCase()** present inside that object i.e. **str**.
    - So, when you are just calling a method of an object, no need to write the full lambda expression. Just pass the reference of that method.
    - 
    - **::** is used to reference a instance method of a class.
    - 
  + Now the confusion is: in both the cases, it is acting differently.
    - In first case, its calling that method using the str i.e. str.toUpperCase()
    - But in second case, its calling that method using the object (System.out) provided and passing the str inside it.
    - **NOTE**
    - If the reference is in the form **ClassName::methodName**, the method is called on the object provided by the function (e.g., str.toUpperCase()).
    - If the reference is in the form **objectName::methodName**, the method is called on that object, with the function-provided value passed as an argument (e.g., System.out.println(str)).
* **Constructor Reference**
  + 
    - Here, I am creating a **ArrayList** of type **Student**, from a **List** of type **String**, using **map** function.
  + 
    - It’ll do the work.
* **Supplier** **interface**
  + It is a *functional interface* that provides a **get** method to create something.
  + 
  + 
    - Here, I used **sp2** to create 2 **ArrayList<Integer>**
    - 
  + 
    - It can also be written like this using the **Constructor reference**.
* **NOTE:**
  + If you don’t specify the type in array list, it’ll take that as **raw** type. Means different data type values can be added in the **ArrayList**.
    - 
    - 

**Reflection in Java**

* Lets say we have created one object of a class. When we call the method **getClass()** it’ll return the **class object** of the class out of which the object is created.
  + For example: lets say the class name is **Car** and we created one object **carObj**.
  + **carObj.getClass()** will give the object which is **Car.class**
  + **Car.class** is the class object of **Car** class.
* The class object is used for reflection.
* Reflection means **inspecting/modifying** classes at **runtime**.
  + Reading fields
  + Modifying fields
  + Invoking methods
  + Listing constructors
  + Creating objects dynamically
  + Reading annotations
* Sometimes you’ll get some object at runtime, but you don’t know which class does that belong to.
  + In that case you can use **?** type generic
  + IMG_256
  + 
* There are 2 fields: **getField()** and **getDeclaredField()**
  + **getField()**
    - It cannot access *private* or *protected* fields.
    - It *searches for the field in super classes* as well.
  + **getDeclaredField()**
    - It can access all type of fields.
    - It **doesn’t** search in super classes.
  + IMG_258
  + In the class, **name** is a private field of type *String*.
  + **getField()** will not be able to get this. But **getDeclaredField()** can access this.
* IMG_259
  + Now, I extracted the field **name** from the class using **getDeclaredField()**
  + Then I set it accessible using **setAccessible(**true**)** method.
  + NOTE: **setAccessible(**true**)** doesn’t mean you are making the field public. It just allows you to read/write after getting it.
  + 
    - This will give error because the **name** field is private.
  + 
    - Now it can be accessible because we made it accessible using the method **setAccessible**.
* I have written 2 private methods in the class named as **display** and **randomMethod**
  + 
    - Both are private
  + 
    - Here I am getting the **display** method using the function **getDeclaredMethod()** and making it accessible
    - Then **invoke** means it’ll run the function. You need to pass the **instance of the class** as it’s a non static method.
  + 
    - As the other method is returning something and also it has some arguments, so we need to pass the **class object** of all the *arguments*.
      * Here *String.class*, *int.class*
    - And the **invoke** method returns a Object type value so the *res* is of type Object.
    - We can also **downcast** here.
      * 
* Reflection in Spring
* Spring uses reflection to inspect classes, discover annotations, create objects, inject dependencies, and call methods — all at runtime.
* Dependency Injection
  +  (we write like this)
  +  (internal of spring)
* Reading Annotations
  +  (we write these)
  +  (spring does this)
* **ReflectionUtils** is a Spring utility class that makes using Java Reflection easier, safer, and cleaner.
  + Java Reflection = raw, low-level, verbose
  + Spring ReflectionUtils = wrapper that simplifies reflection
* Comparision
  + IMG_270
    - Spring ReflectionUtils
  + IMG_271
    - Java Reflection
* Some advantages of Spring **ReflectionUtils** over Java Reflections
  + It handles exceptions by itself
  + It marks **setAccessible(true)** by default
  + Handles super class traversal as well.
  + Provides iteration helpers
    - 

* F

**synchronized**

* It is used to lock **class, object, instance method, static method** to make those thread safe.
* Synchronized block with **Class object**.
  + 
  + It is **class level**, not object level
* Synchronized block with **objects**
  + Synchronized block with **this** (lock itself)
    - 
    - It is **object level**
  + Synchronized block with **3rd party object** (lock another object)
    - 
    - here **lockObject** is another object
      * 
* Synchronized **instance method**
  + Method can be called once at a time.
  + 
* Synchronized **static method**
  + Static method can be called once at a time
  + 
* **this** vs **3rd party object** **synchronized** 
  + **this** 
    - 
      * Here it is using **this** means this object (object of ***A***) itself will be locked.
      * So, if a thread has called **obA.m1()** then no other method can use **obA** ; different instance of Account can be used.
    -  (pseudocode; not syntax)
      * Here, **T2** will not run, as **obj.m1()** requires the lock on this object **obj** itself, but it is locked by **T1**,
  + **lockObject**
    - 
      * Here, **lockObject** (3rd party object present inside class A) is only locked, but **obA** can be used by other threads as this is not locked.
    - 
      * Here, **T2** will run because **obj.m1()** requires lock on **lockObject**, and **T1** has locked **obj**.
      * **obj** is object of class **A**; **lockObject** is a object present inside class **A**;
* There is a keyword called **volatile** in ***Java***.
  + IMG_256 (Stripe class)
  + **volatile** make sure that if one thread changes the variable, then other thread will immediately sees that.
  + If you think, both the threads contains the same reference of variable, then you are wrong.
    - 
    -   
      this is how execution happens, both thread has their own copy of **cache** where the variable is stored.  
      And when it is made **volatile**, both take reference from the ***main memory*** directly.
  + Consider the below example:
    - 
    - 
    - **thread-1** is calling **run**, **thread-2** is calling **stop**, if the variable was not ***volatile***, it’d not stop the thread-1.
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F