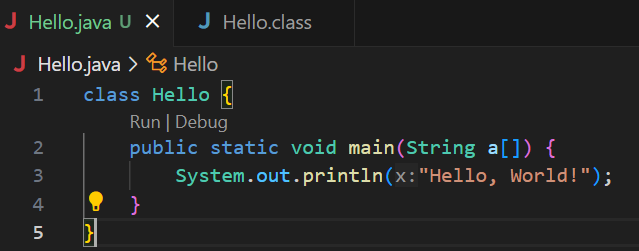
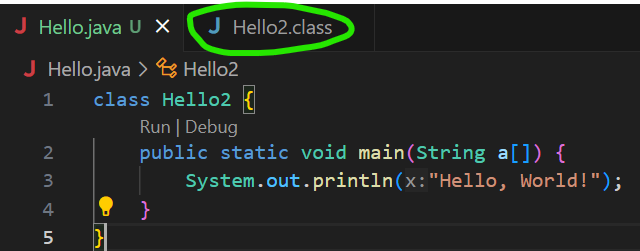
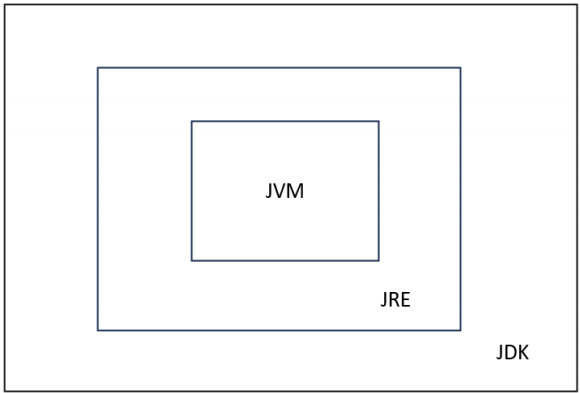
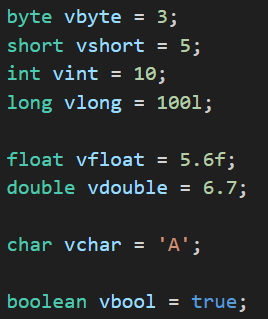
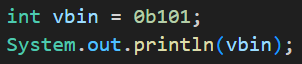
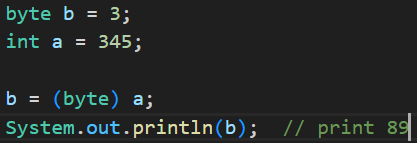
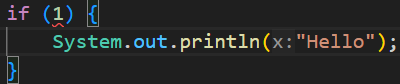
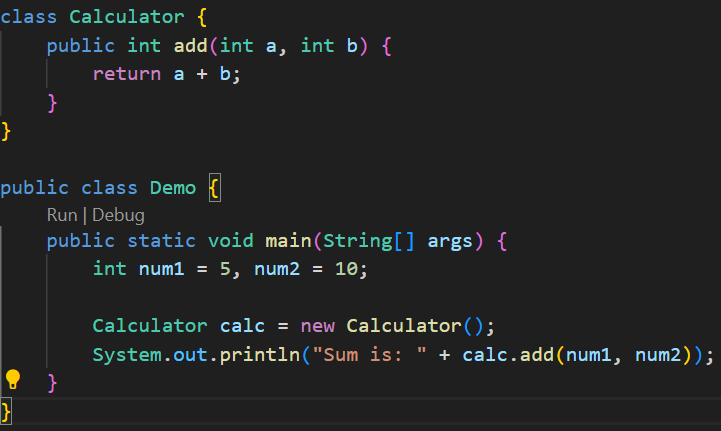
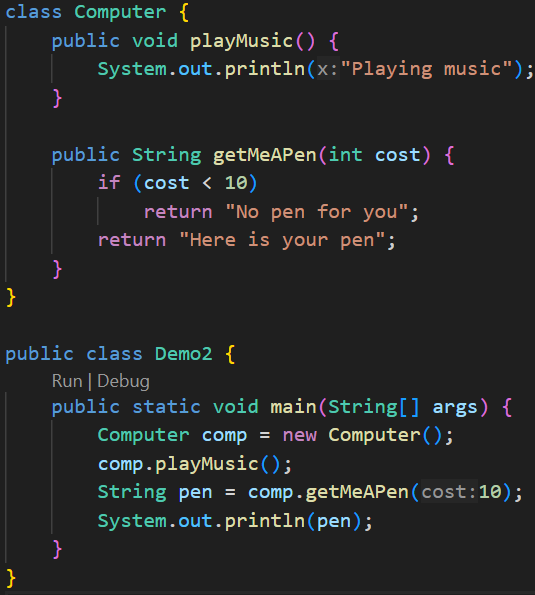
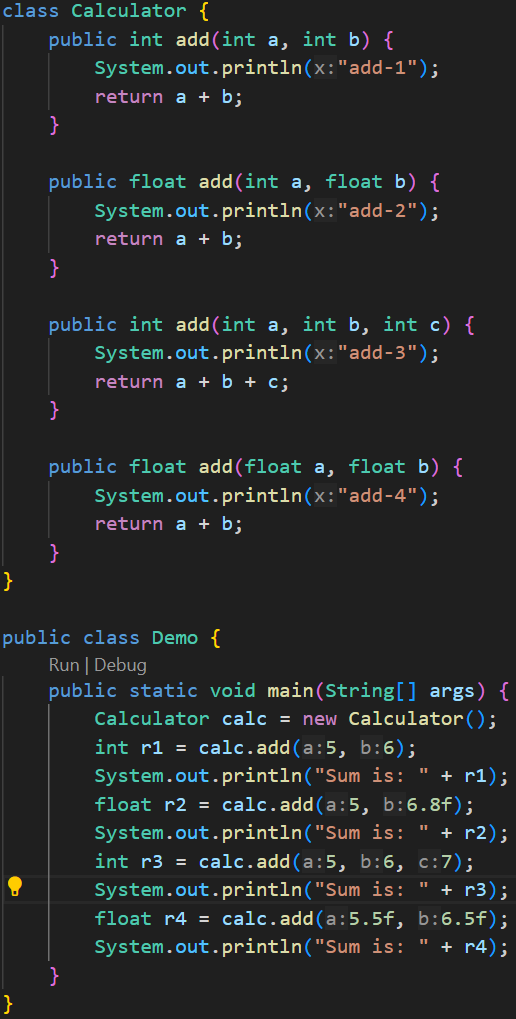
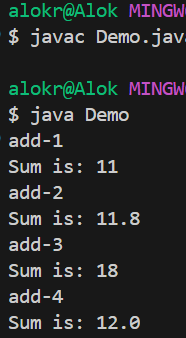
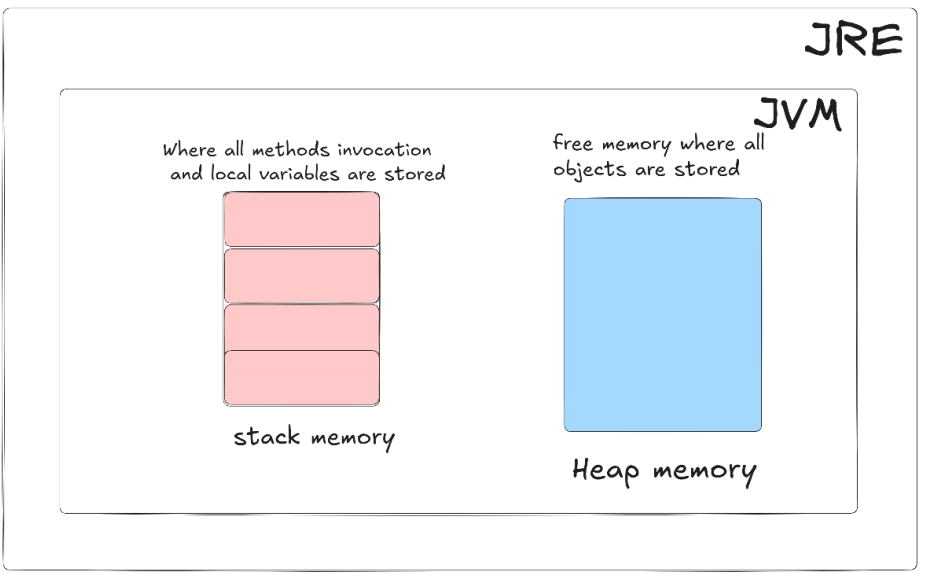
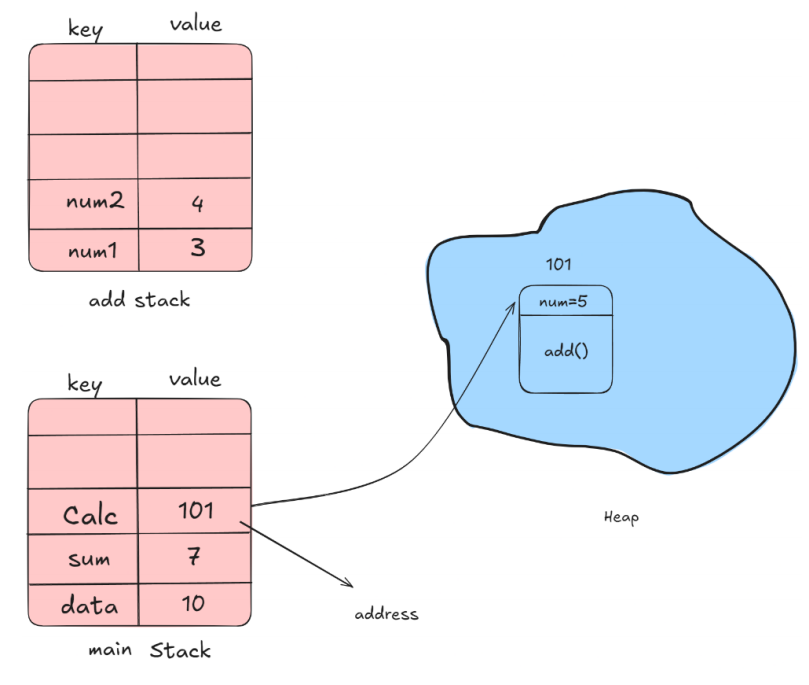
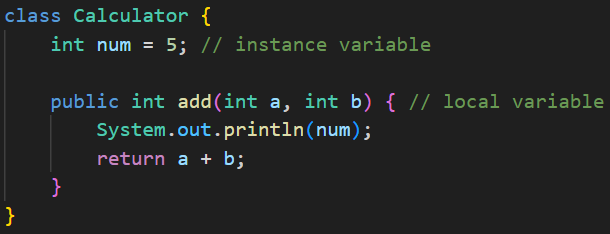
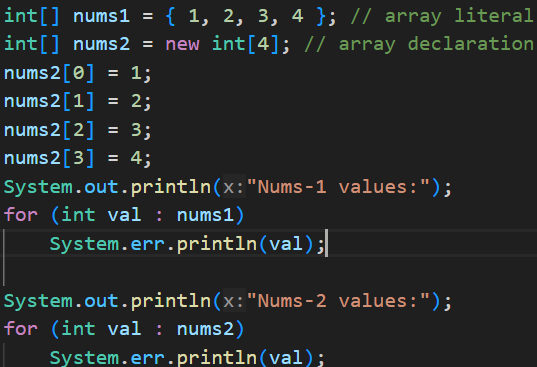
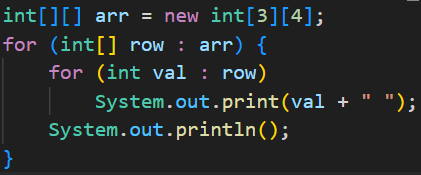
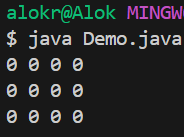
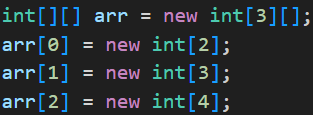
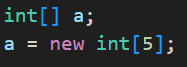
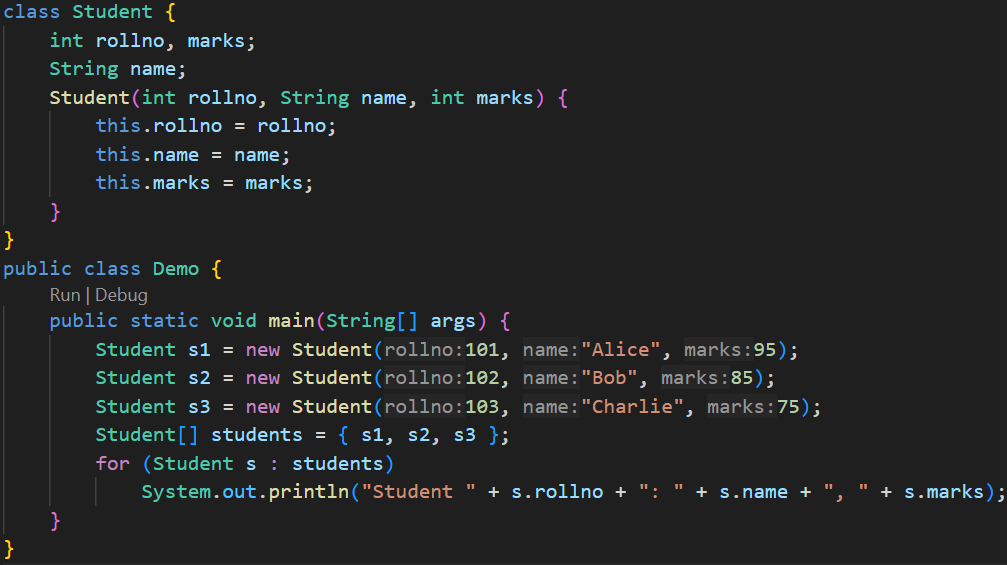
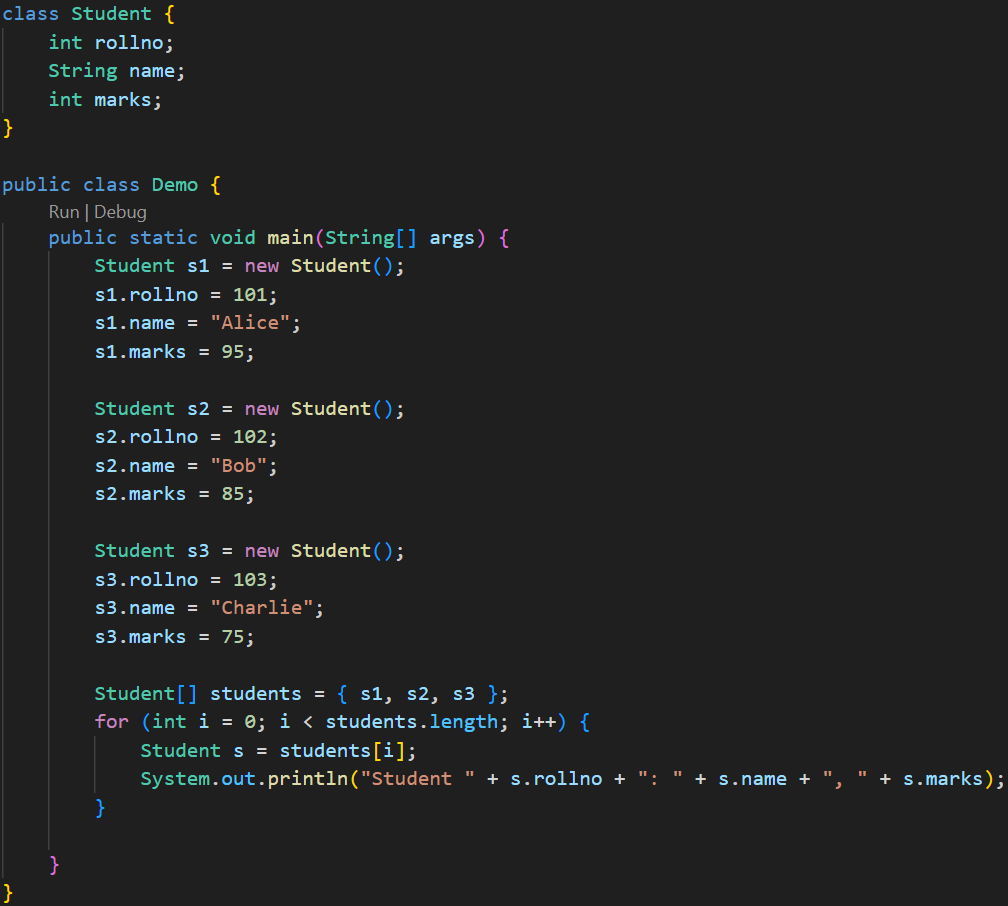
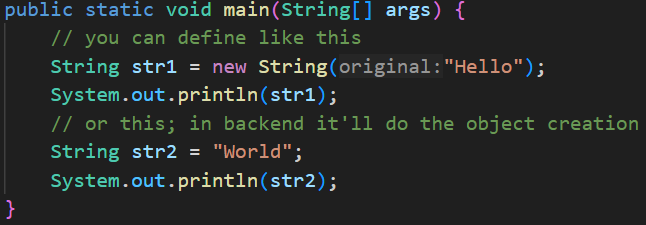
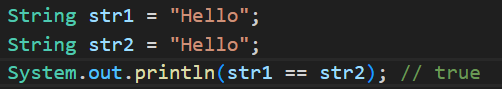
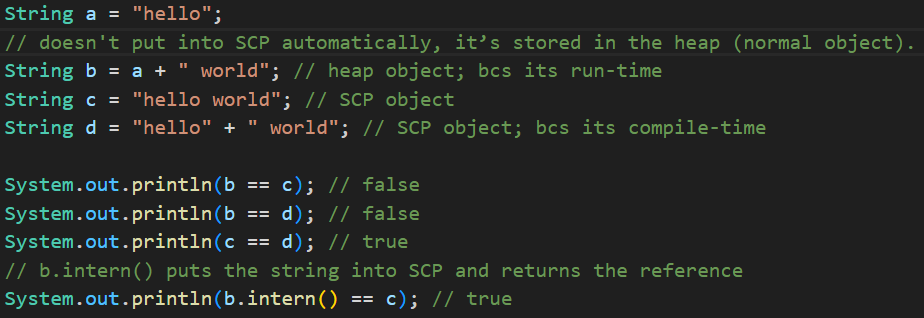
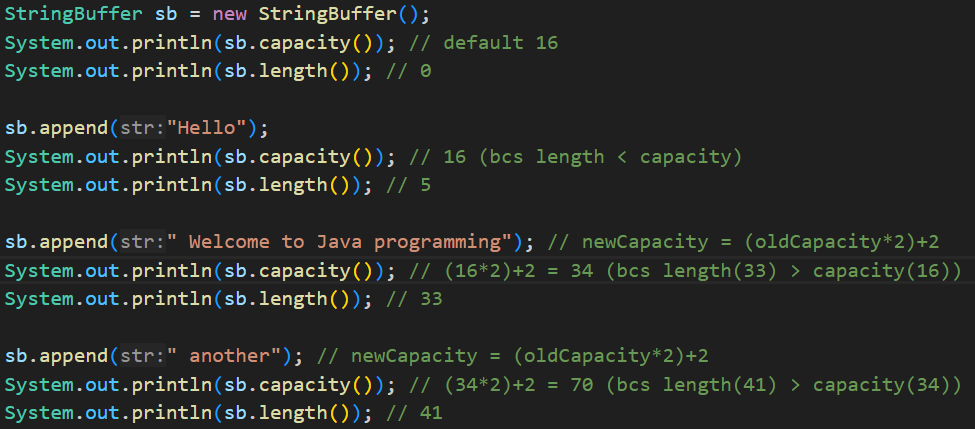
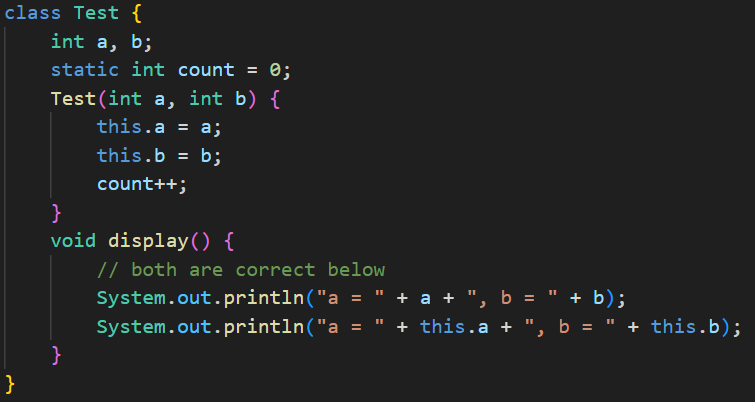
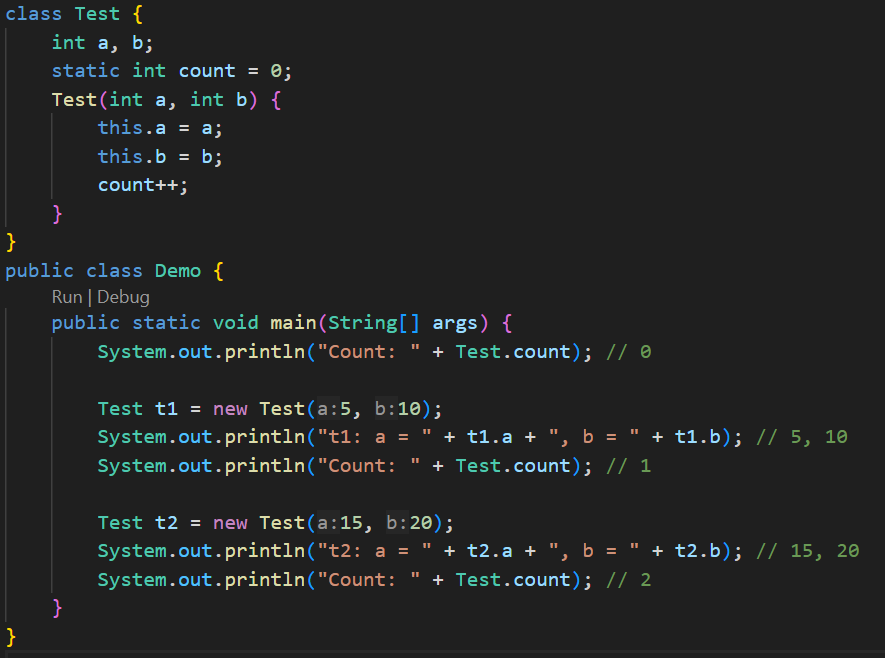
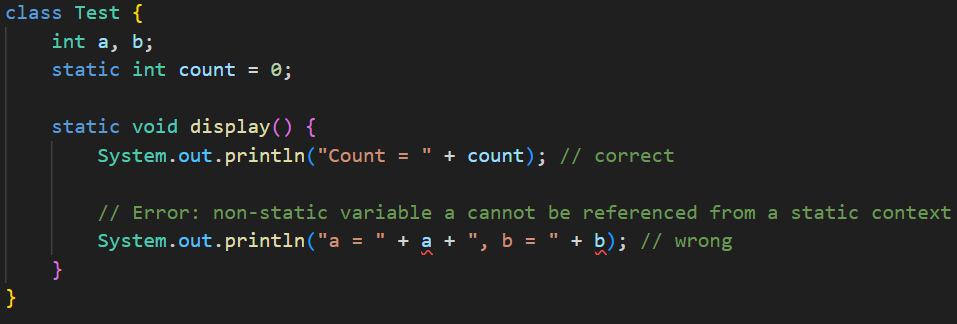
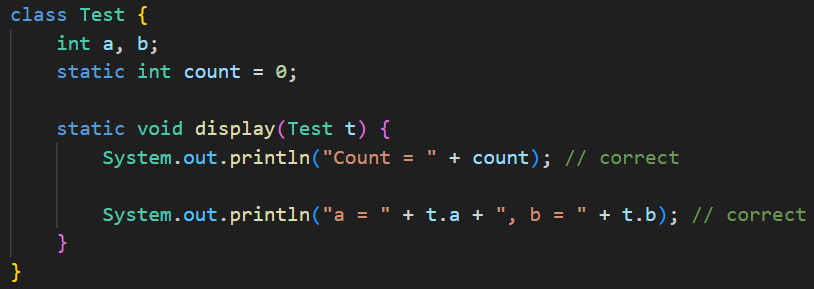
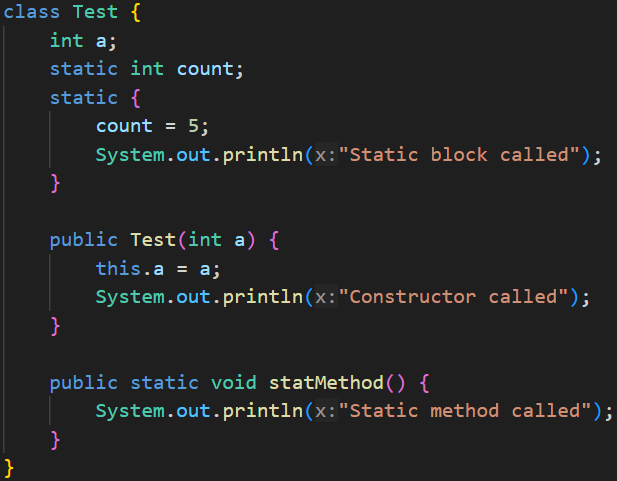
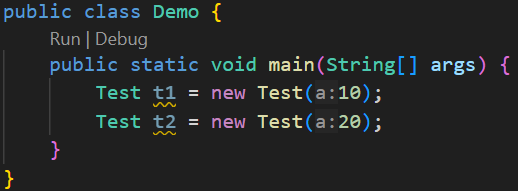
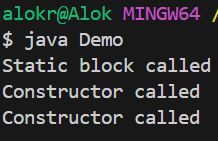
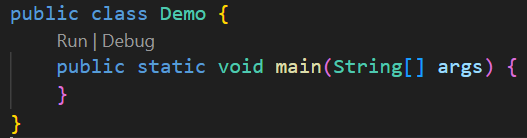
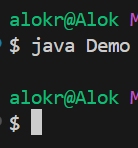
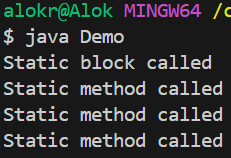
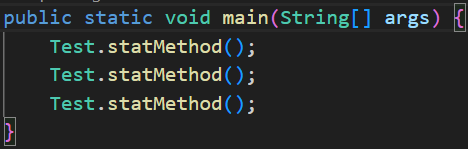
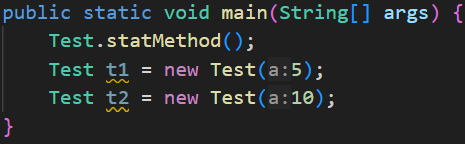
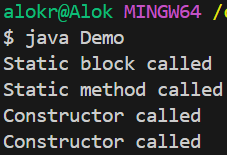
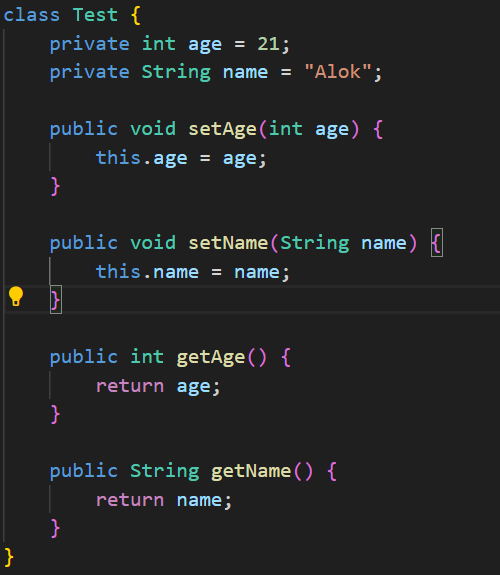
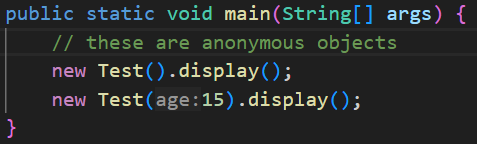
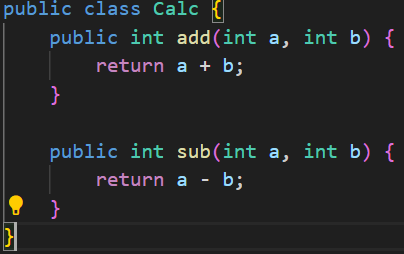
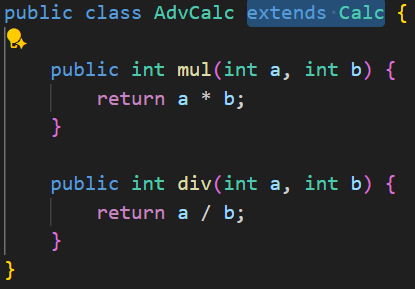
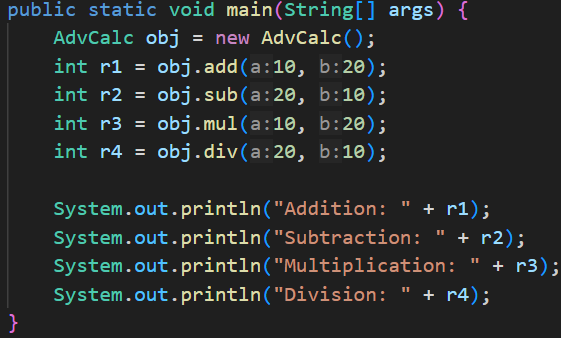
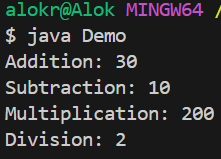
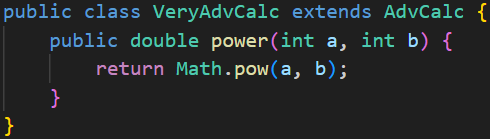
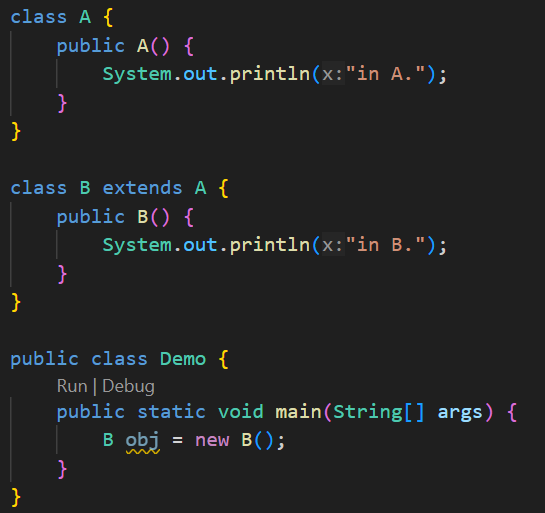
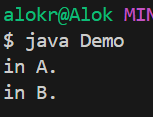
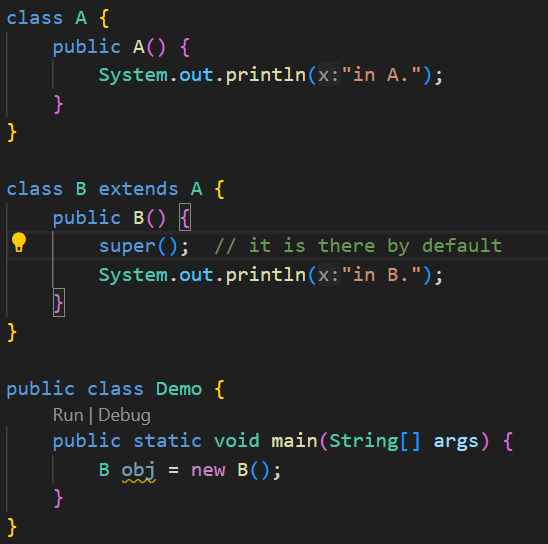
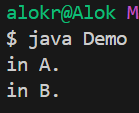
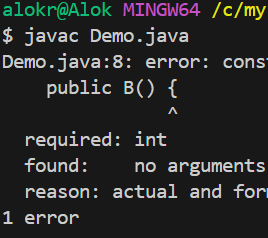
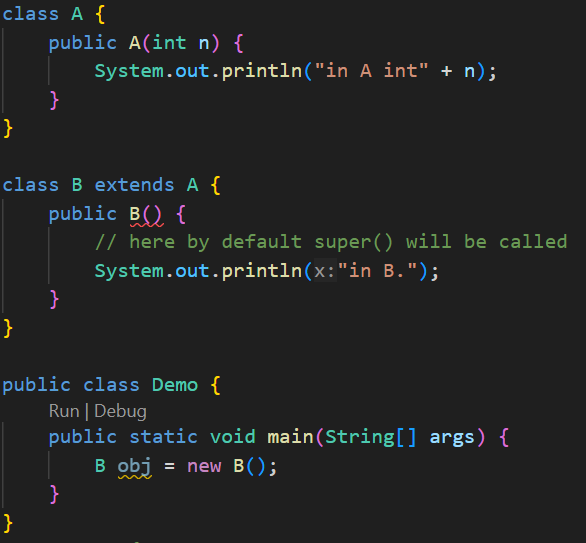
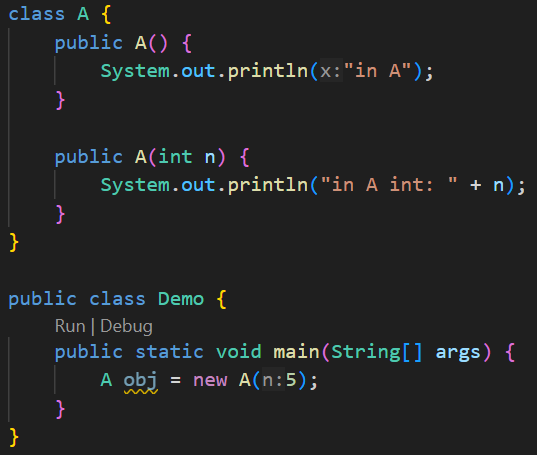
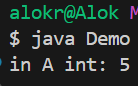
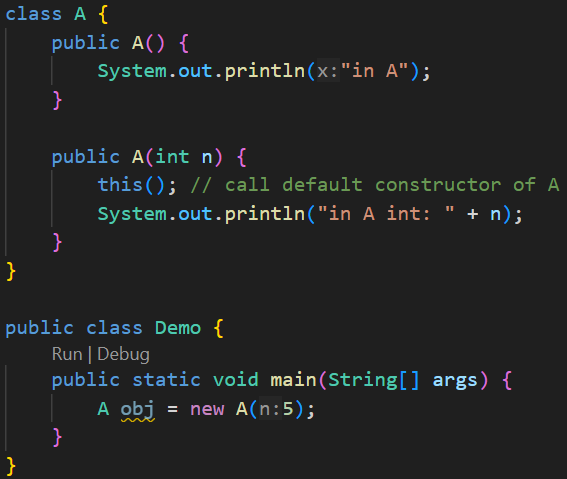
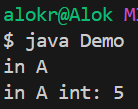
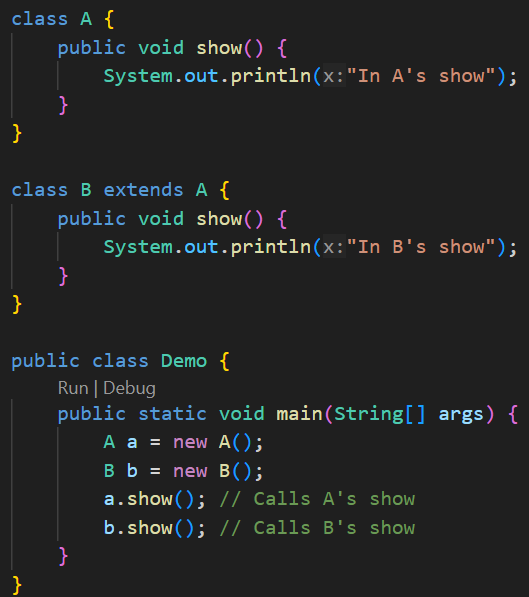
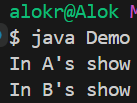
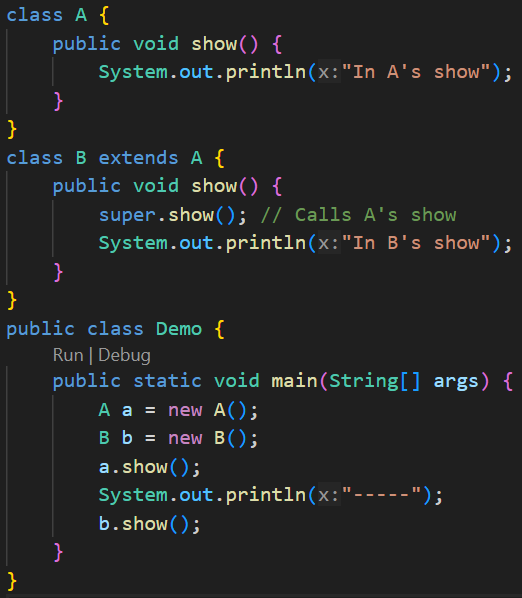
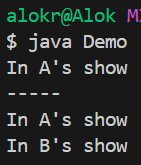
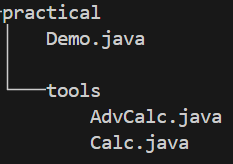
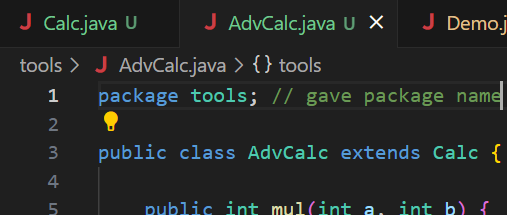
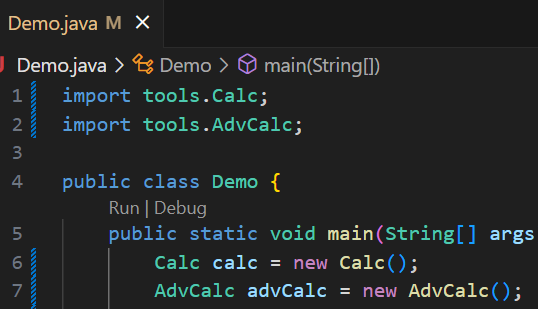
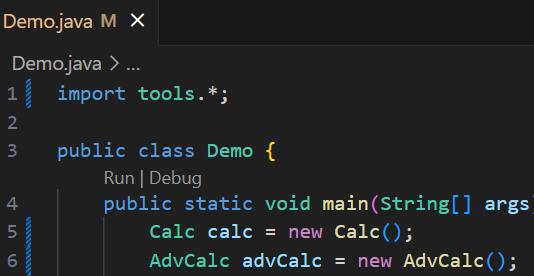
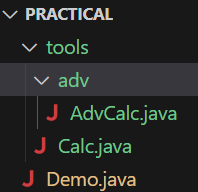
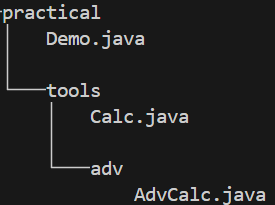
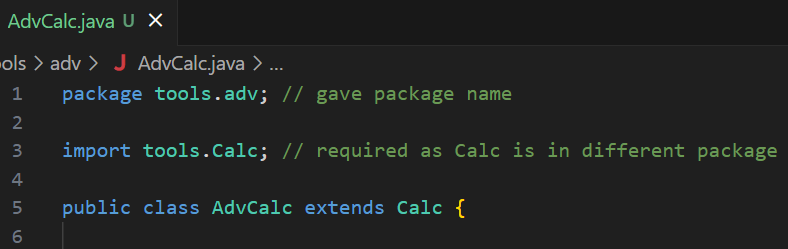
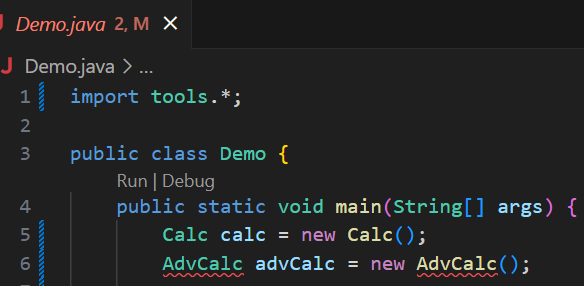
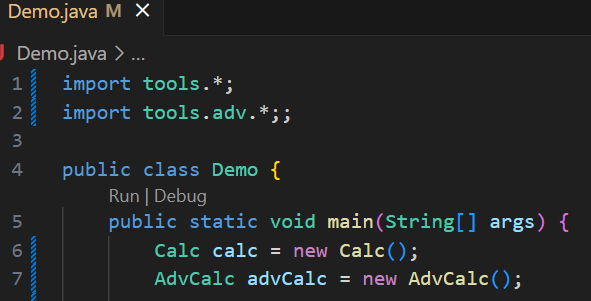
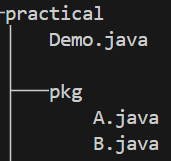
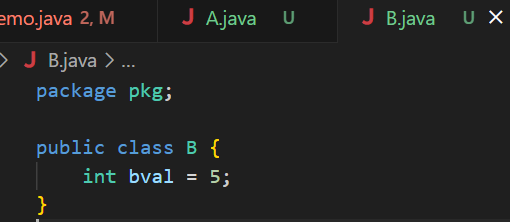
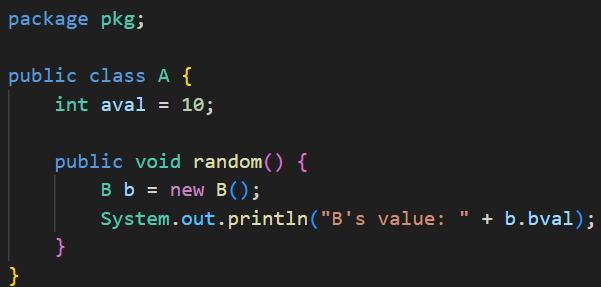
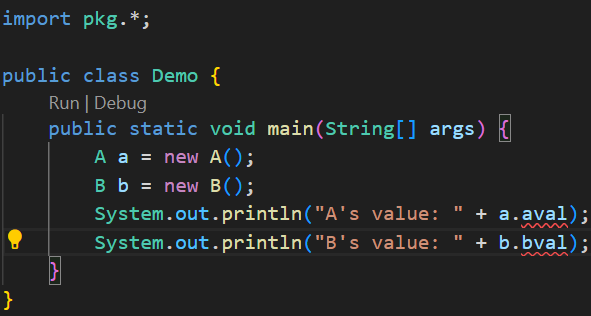
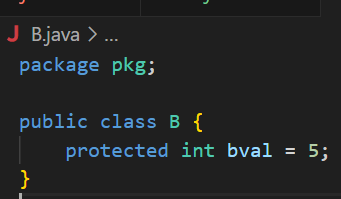
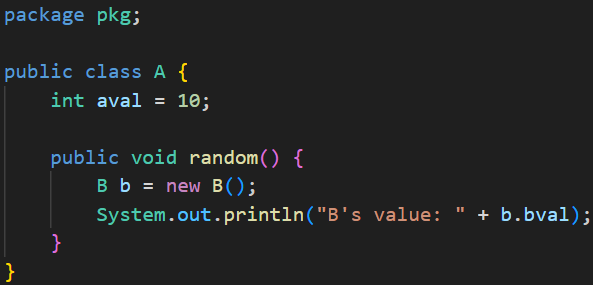
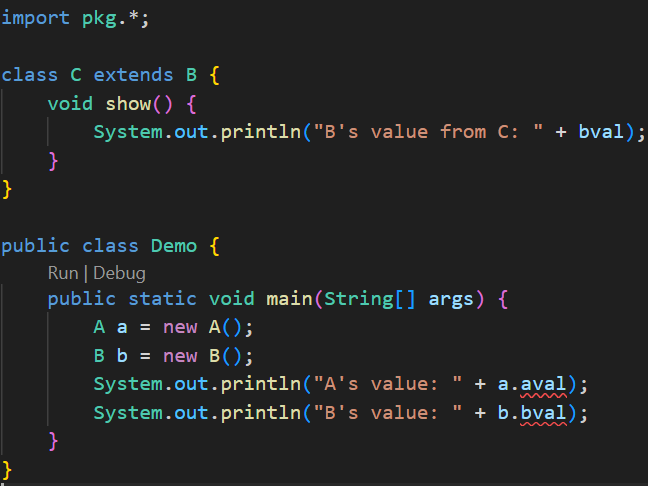
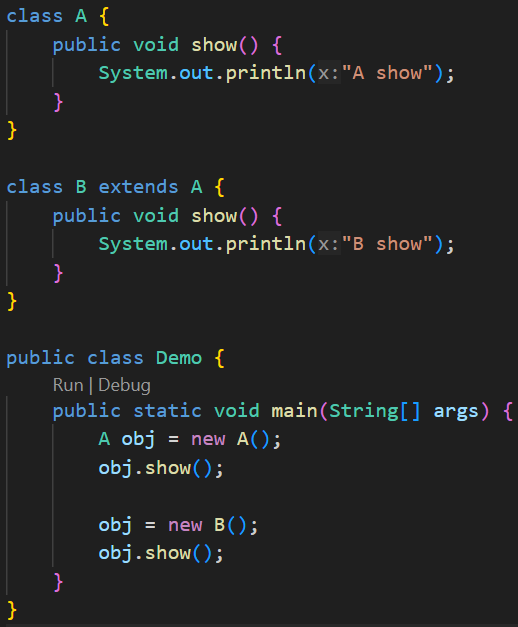
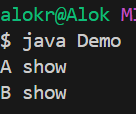
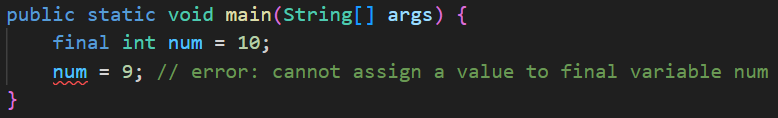
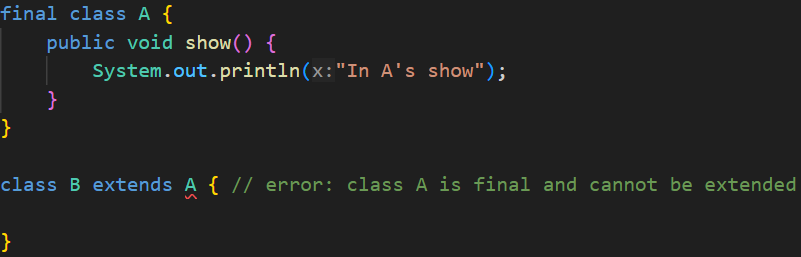
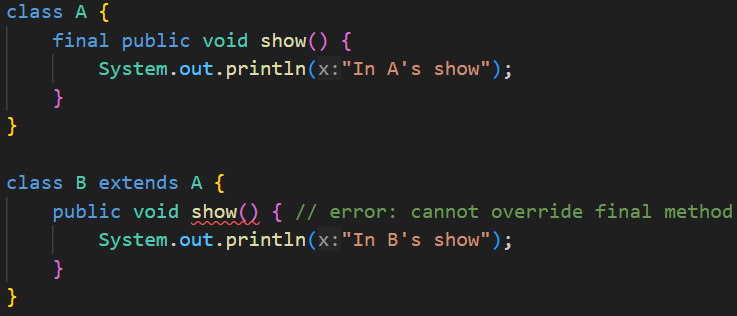
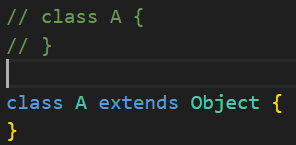
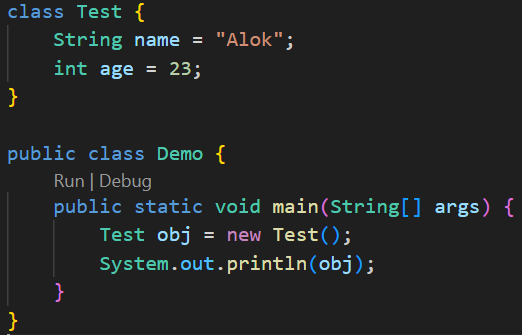
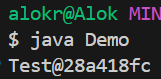
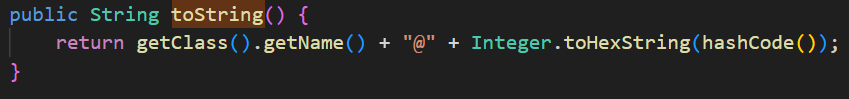
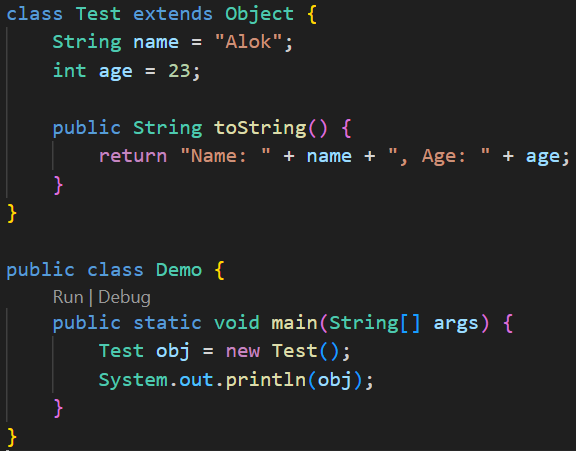
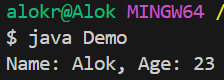
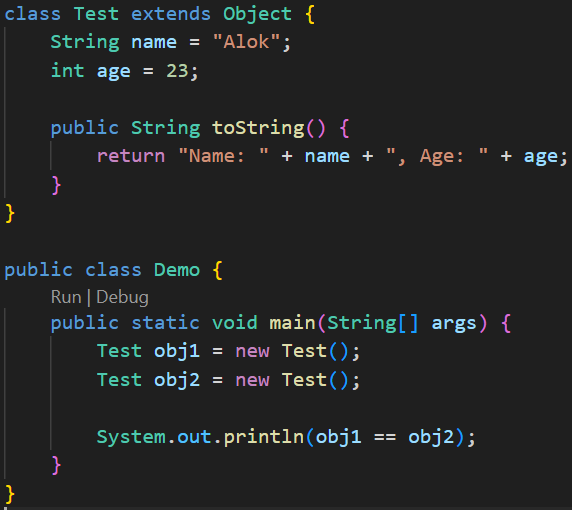
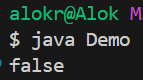
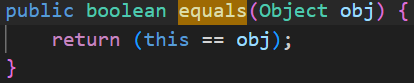
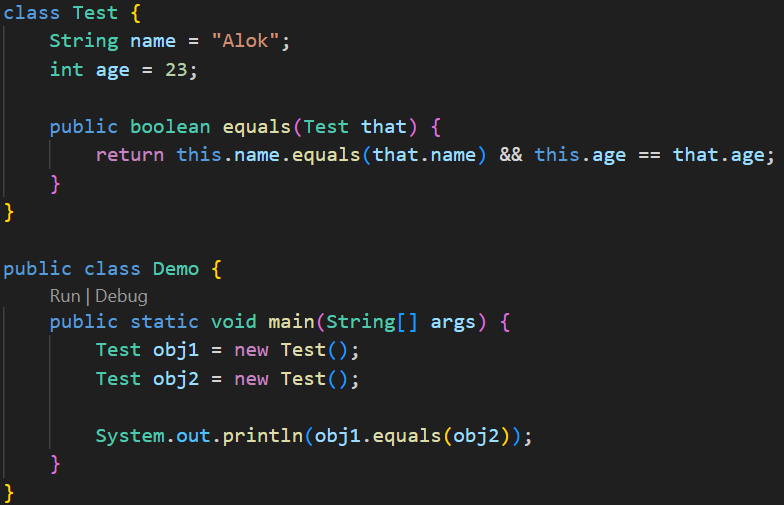
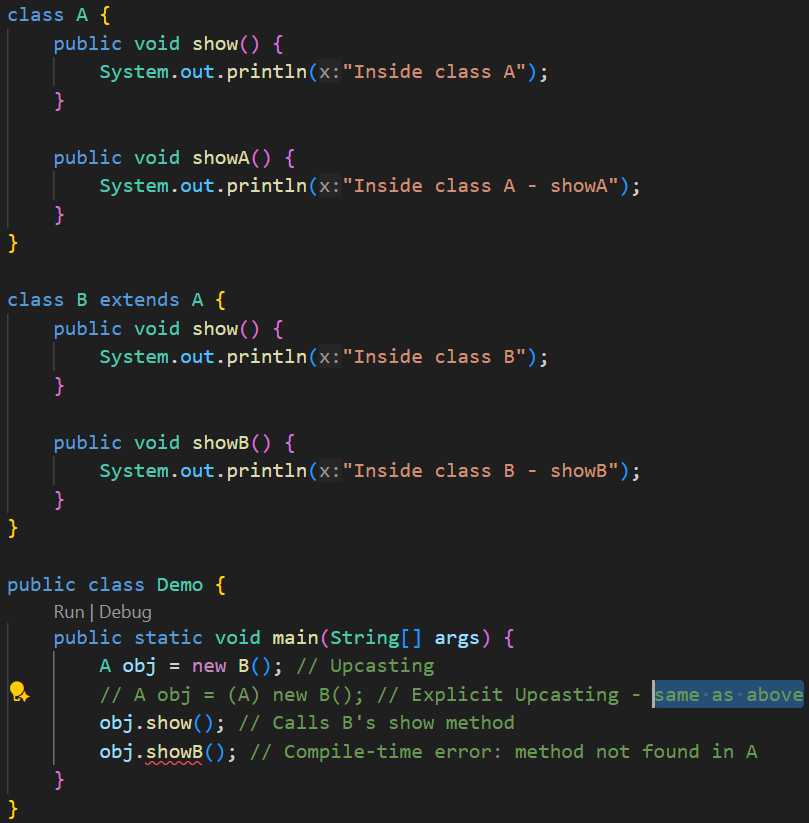
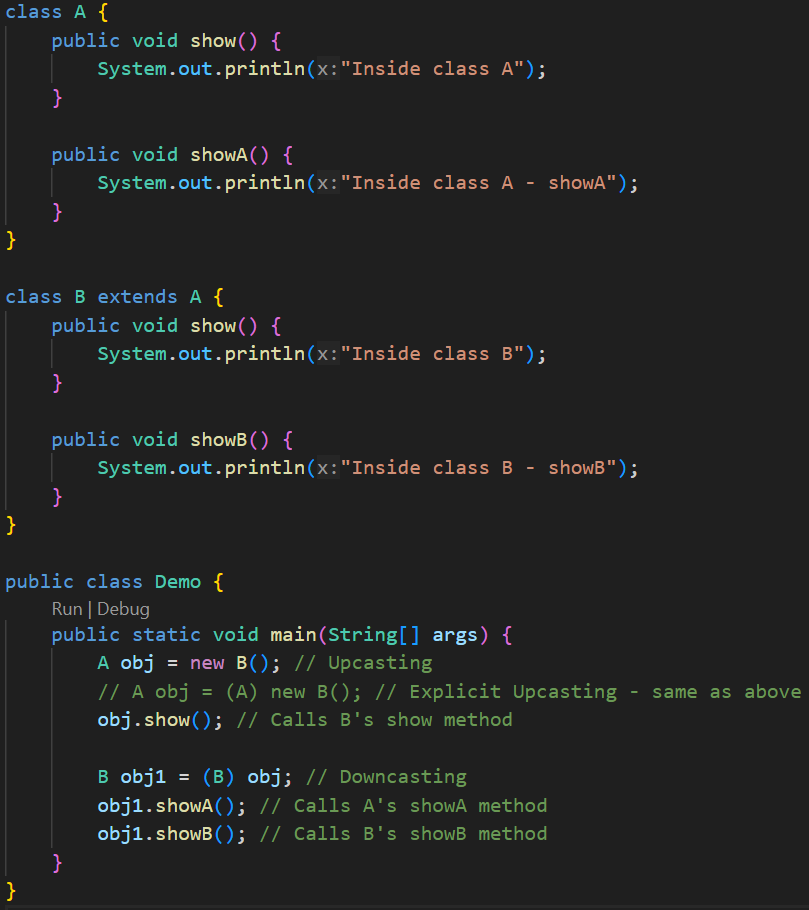
* **setx JAVA\_HOME "C:\Program Files\Java\jdk-21"**
  + It is used to set the environment variable permanently.
* How the Java code works?
  + Java Code (**.java**) ------<compiler (javac)>------ Byte Code (**.class**)
  + Byte Code goes into JVM (JVM accepts only byte code)
  + JVM only run only one file.
    - It says, even if you have 1000 files, you need to tell me which is the first file that I’ll run.
    - That file needs to have **main** *method*.
  + Whenever you run a **.java** file, one **.class** file will get created. It is the Byte Code file.
* To run a java code (file name is let: Hello.java)
  + If you run this using **javac Hello.java**, one file will be created depending upon the *classname* used inside the **Hello.java**.
  + 
  + Here I gave the *classname* same as the *filename*.
  + But, it doesn’t matter what is the filename; you can give the *classname* different.
  + But, the **.class** file will be created with the *classname* that is mentioned inside the **.java** file.
  + 
    - You can see here, I gave the class name *Hello2*, and the file got created is *Hello2.class*.
    - But, if you are creating **public** class, then the **filename** must be **same** as the **classname**.
  + First you need to run the command **javac Hello.java**
    - It’ll create the **.class** file.
  + Then, run the **.class** file using **java** **Hello**
    - Here, don’t write *java Hello.class*
* JVM(Java Virtual Machine) is present inside JRE(Java Runtime Environment).
  + When you need to run something, it might requires some libraries.
  + JRE provides that.
  + JVM is just a part of JRE.
  + Kitchen Analogy:
    - The kitchen provides the environment to cook a dish.
    - Utensils and Ingredients are like the libraries and resources JRE provides.
* **JVM < JRE < JDK** (inner to outer layer)
  + JVM present inside JRE and JRE present inside JDK.
  + JDK is used by developers to develop the code and all.
  + If you want to run your application in any other system, that doesn’t require JDK. Only JRE should be there to run the application.
  + 
* Is JVM like a virtual machine on top of OS?
  + Yes and No:
  + ✅ Yes: It behaves like a virtual computer for Java bytecode. That’s why Java is “Write Once, Run Anywhere.”
  + ❌ No: It doesn’t emulate hardware or run another OS. It sits on top of the real OS, using system calls, memory, CPU instructions, etc.
  + So, JVM is an abstraction layer, not a full-blown VM like VirtualBox.
* **Variables**
  + Primitive
    - Integer (byte(1), short(2), int(4), long(8)) (for long: you need to write **l** as suffix)
    - Float (double(8), float(4)) (**2.3** => double, **2.3f** => float) (default is *double*)
    - Character (2 bytes)
    - Boolean (doesn’t work like 0 and 1; only **true** and **false** works)
    - 
    - NOTE: you can’t double quotes in **char** variables. That is only for **string**.
    - 
      * You can also write in binary format, when you execute this, output will be **5**.
      * As 101 = 5
      * For hexadecimal, use **0x** as prefix.
    - Unlike C++, in Java also if you print **(int)(ch)** where ch is char variable, it’ll print the ASCII value of that character.
    - Higher sized variables can’t be assigned to smaller sized variables; but vice-versa is possible.
      * Ex: int can’t be assigned to short; but short can be assigned to int.
      * If you are assigning larger to smaller, then you need to do type casting.
        +  (345 % 256 = 89)
    - Type promotion:
      * lets say you have 2 byte variables 10, 30.
  + When you multiply these and store the result in a variable, it’ll automatically become int (as 300 is not in the scope of byte variable).
* **Some points to be remembered**
  + Just like C++, here also, when you divide 2 inegers, it’ll return a integer value only. Not float number.
  + Operators (arithmetic, logical, ternary and all), Conditional statement (if, else if, else) are same as C++.
  + **switch case** statement is also same as C++.
  + string + int + int => for example: “abcdef “ + 5 + 6 => “abcdef 56”
    - It’ll concatenate.
  + 
    - This is wrong. It’ll give error that *can’t convert int into boolean*.
* **Classes and Objects**
  + - 
      * You can’t run this directly using **java filename.java**, you need to compile and run separately.
      * Bcs, when you compile it, 2 .class files will be created i.e. Calculator.class and Demo.class.
* **JDK JRE JVM**
  + JDK: Java Development Kit
  + JVM: Java Virtual Machine
  + JRE: Java Runtime Environment
  + Compilation happens in JDK, Running happens in JVM.
  + Most of the time, you’ll be using some built-in libraries; in this case JRE comes into play.
    - One extra layer outside JVM, which is JRE, stays there to provide the libraries during the run.
* **Methods**
  + While creating a method, you should provide a proper access modifier.
  + 
  + **Method Overloading**
    -  
      * Number of arguments, type of arguments, type of return type: depending upon these, method overloading can be done.
* **Stack and Heap**
  + Inside JVM, there are 2 types of memory.
    - Stack (Last-In-First-Out)
    - Heap (open space)
  + 
  + 
    - Each method takes its own stack like main method, add method (consider previous example)
    - The stack is having 2 partitions: left side is for key and right side is for value.
    - Whenever you create an object out of a class, it is created inside the Heap.
      * In the heap, inside the memory block that the object acquires, is having 2 parts.
        + One is for **properties** (instance variables)
        + One is for **method definitions**
    - When you call the method using the object, the method gets loaded inside the Stack, create its own local variables and gets executed.
    - Note: the instance variables inside a class will be staying inside the heap only.
    - 
    - Instance variables are specific to objects, not class. Means, each objects will have independent instance variables.
      * Because, each objects will be having different memory blocks inside the heap.
    - The reference of the object inside heap, is stored inside the stack.
* In Java, everything, which is an object, gets created inside Heap.
  + Ex: Array,
* **Array**
  + **int arr[]** and **int[] arr**
    - These both are exactly same
    - But, **int[] arr** is preferable; as it shows **arr** is of **int[]** type.
    - First one i.e. **int arr[]** is derived from C/C++ style; which is 100% correct in Java.
  + 
    - These are 2 types of Array declaration.
    - Both are **fixed sized** only.
    - An non-initialized array’s values will be **0**.
  + 2D Array:
    -  
    - Arrays are not having dynamic sized (for dynamic size, ArrayList is used)
    - But in case of 2D array, we can create an Array having different sized rows.
    - 
      * Just don’t mention the column size while creating a 2D array and after that initialize the rows.
      * It is called  **jagged** array.
      * 
        + It’s just like this.
      * We are creating an array of sized **3** *(new int[3][])* where each value is of type **int[]**. After that we are initializing those.
  + **Array of Objects:**
    - 
    - 
  + **Drawbacks of Array:**
    - You can’t change the size of the array.
    - O(n) for searching.
* **Strings**
  + 
  + When you create a string variable, one object of type **String** will be created in heap and your variable will store the reference of that object.
  + One part is there inside Heap called as **String Constant Pool**,
    - Whenever you assign one *string value* to a variable, one *constant string literal* will get created inside that String Constant Pool and reference of that will be stored in a variable.
  + **Immutable Strings:**
    - Lets you create **String** variables having same value, then both the variable will be storing the reference of same *string literal* as in side the String Constant Pool, all the strings stored are unique.
    - 
    - Whenever you assign one *string literal* to a variable, it first checks inside the String Constant Pool;
      * if that particular literal is not present then it creates one and store its reference inside the variable.
      * If present, if just store the reference of existing string literal in the variable.
      * For concatenation: there are the following cases:
      * 
  + **Mutable Strings:**
    - **StringBuffer** is used to create mutable strings.
    - 
      * By default, the capacity is 16.
      * If the string size exceeds the capacity, then the capacity will be increased with the formula: **New Capacity = (Old Capacity) \* 2 + 2**
* **NOTE** Inside a non-static method, use of **this** keyword to access the **instance variables** is optional. (**this** is required in case of naming conflict; constructor is example here)
  + 
* **static** keyword:
  + Static variables is shared among all the objects.
  + Static variables are stored in the Method Area (a part of JVM memory), not on the Heap.
  + You can call access the static variables using objects as well; but it is not preferable.
  + You should access the static variables using class only
    - **ClassName.StaticVariableName**
  + 
    - Example of Static Variable.
  + From a **static method**, you can’t access the **instance variables**. Because instance variables are specific to the objects but **static method** is specific to class; not objects.
    - If you want to access the instance variables, then you can pass the object as an argument to the **static method**.
    - 
    - 
      * Now it is correct.
    - If you create any variable inside a static method, then after the method is executed then the variable will be gone.
* **Static Block**
  + Whenever you instantiate a object with a class, then first the class gets loaded then the object will be created.
    - If you are instantiating more than one object with a single class, then **loading of class will happen only once**.
      * When the first object will be created, class will be loaded;
      * after that when 2nd object will be created, it sees the class is already loaded; so now only object creation will happen.
    - There is a **static block**, where you can assign values to the **static variables**. This block gets called when the class is loaded.
    - It means, even if you are creating **n** number of objects **(or)** you call any static method of that class **n** times, **static block** will be executed only once.
      * If there is no object getting created (or) not any static method call, then static block will not be executed.
    - 
      * This is the class having static block.
    -  
      * Because, class was loaded only once.
    -  
      * Because, as no object was created; so class loading didn’t happen.
    - 
      * Because, only during the first static method call, class was loaded.
    -  
      * When static method got called, class was loaded; so while instantiating objects, it didn’t require to load the class.
    - If you want to load the class even if no **static method** got called or **no object** got instantiated; then you can use **Class.forName**
      * It load the class to the memory using class loader.
* **Encapsulation:** (hiding variables)
  + 
    - You can set the access parameters of the variables and add getter and setter methods for those.
* If you don’t specify any access modifier, then by default it’ll be **package-private**. (neither private nor public not protected) (its valid for class, method, variables inside class).
* **Constructor**
  + 2 types of constructors are there:
    - Default constructor
    - Paremeterized constructor
  + You can define more than one constructors;
  + constructor name will be same as the name of class; it’ll not have any return type.
  + All the constructor you define will come under the method overloading concept.
  + 
* **Naming Conventions**
  + Class, Interfaces : Pascal case (MyClass)
  + Variables, Methods : Camel case (myVar)
  + Constants : All capital (MY\_CONST)
* **Anonymous Object**
  + It is just creating a object but not assigning it to any variable.
  + 
* **Inheritance**
  + **Single Level Inheritance**:
    - 
    - 
    -  
  + **Multi Level Inheritance**
    - 
    - Now: VeryAdvCalc >> AdvCalc >> Calc
  + **Multiple Inheritance**
    - Java doesn’t support multiple inheritance.
* **super method**
  + **super** is nothing but the alias of parent class’s constructor.
    - When you write **super()**, it’ll call the parent class’s default (non-parameterized) constructor.
  + When a class inherits another class, even if you don’t write **super()** inside the constructor of the child class, java executes **super()** by default.
  +  
  +  
    - Both the above cases are same.
    - Even if you don’t call **super()** it’ll be called by default.
    - **super()** constructor call should be in the first line ***default/parameterized***).
  + If there is only a parameterized constructor in the parent class, and if you don’t write **super(arg)**, then it’ll give error.
    - Because, by default Java will call the parent’s constructor as **super()** only i.e. it’ll call only the default constructor (non-parameterized).
    - As there is only one constructor present in parent class, which is parameterized; calling **super()** in the child class will give error.
  + 
    - In this case, you need to call the parent’s parameterized constructor explicitly.
* **this method**
  + **this** refers to the current object instance; when you call **this()** it’ll call the current class’s constructor.
  +  
    - Here, by default the parameterized constructor got called as we passed one argument while instantiating the object.
    - If **this** is getting called inside the constructor, then it should be in the first line inside the constructor just like super.
    - **this()** and **super()** cannot be called inside the same constructor; because both of them needs to be written in the first line inside the constructor method.
  +  
    - Now, as we called **this()** inside the parameterized constructor, so its calling the default constructor inside the class.
* **Method Overriding**
  + When child class implements same method which is present in the parent’s class as well; its called Method Overriding.
  + **Same name, same type of arguments, same number of arguments, same return type; just different definition.** (unlike method overloading: name/type/number of args/return type => one or more of these should be different)
  +  
    - Its simple only; B’s show function is overriding A’s show function.
  +  
    - In here, I tried to implement some extra functionality inside the **show()** method. Not completely overriding A’s show method.
  + Whenever you call a method, it’ll search that method in the current class first; if it doesn’t get that then it’ll go to the parent class.
* **Packages**
  + Some related files should be separated and kept inside a folder. This folder will be treated as a package; but you need to mention the package name inside the files.
  + 
    - It is my current directory structure; I kept *Calculator* related stuffs inside the **tools** directory
  + 
    - Gave the package name as **tools** in both **Calc.java** and **AdvCalc.java**
  + Now, we can’t use the classes (i.e. *Calc* and *AdvCalc*) inside the **main** method inside *Demo* class directly as both are now inside different folders.
    - 
    - Instead of importing the modules one by one; we can import all the modules present inside that package directly using **\***
    - 
  + Now lets see nested package structure:
    -  
      * I transferred **AdvCalc.java** inside a new directory ***adv***.
    - 
      * Now we need to import Calc as well as it is in different package now.
    - 
      * Here you can see, I have imported **tools.\*** still getting error in **AdvCalc**.
      * It is because, **packageName.\*** only imports the files present inside that package; in our case inside **tools** one more package is present which is **adv**.
      * So here, **tools.\*** is only importing the **Calc** file. Its not importing **tools/adv/AdvCalc** file.
    - 
      * Now it’ll work properly.
  + You can’t give different package names to the files which are siblings to each other i.e. present inside the same folder.
* **Access Modifiers**
  + When you don’t give any access modifiers, it’ll be default which is **package-private** i.e. only the files present in same package can access those.
    -  (This is my file structure for now)
    - 
      * (practical/pkg/B.java)
    - 
      * (practical/pkg/A.java)
      * You can see here, there is no error while accessing the variable of class B.
    - 
      * (practical/Demo.java)
      * Here, error is coming as **Demo.java** is not inside the package of **A** and **B**.
  + **Public**
    - It is accessible in everywhere; Within the class, Child class, Classes present inside same package, Classes present in different package etc etc.
  + **Private**
    - It is accessible only within the class. No where else it is accessible.
  + **Protected**  ***(IMPORTANT)***
    - Within the same package (just like default/package-private).
    - From subclasses in other packages (extra power over default).
    - 
      * (practical/pkg/B.java)
    - 
      * (practical/pkg/A.java)
      * No error because A and B are inside the same package.
    - 
      * (practical/Demo.java)
      * **Class C** doesn’t give any error as it is inheriting **Class B**
      * But, inside **Class Demo** it is giving error because neither it is inheriting those classes not it is present inside the same package where **A** and **B** are present.
  + **Protected** and **Package-Default** varies in a single case which is:
    - In case of **Protected**, class X present outside of the package, if inheriting the class Y then it can access that protected variable of class Y.
    - In case of **Package-Private**, even if the class X is inheriting class Y but not present inside the package of Y, then it can’t access the variabled of Y.
  + We can’t have 2 public classes in a same file.
* **Polymorphism**
  + 2 types of polymorphism:
    - Compile time (Method Overloading)
    - Run time (Method Overriding)
* **Dynamic Method Dispatch**
  + Process by which a call to an overridden method is resolved at runtime rather than at compile-time.
  + Assigning a object of **child** class to a variable of type **parent** class **(vice-versa is not true)**.
  +  
    - Here, the variable is of type **A** (parent class).
    - During the compile time, it is not sure that which show method will be called. It’ll be decided during the run time only.
      * The method, that will be called, depends upon the **type of object**, not **type of variable**. (**it is only for overriding; for more check Upcasting *and Downcasting)***
* **Final Keyword**
  + Its used to create a **constant** variable.
  + Can be used to create **variable, method, class**
  + **variable**
    - 
  + **class**
    - When you make your class **final**, means you are stopping further inheritance.
    - 
  + **method**
    - When you make the method **final**, means this method cannot be overridden.
    - 
* By default, every classes inherits the class **Object**.
  + 
    - These both are same only.
* Some methods inside Object Class (toString, equals, hashcode)
  +  
    - When I printed this object of class Test, it gave some output like this.
    - 
    - That output is because of this method present inside the class **Object**
    - We can modify that.
    -  
  +  
    - Now, it is giving **false**; it is because of the method **equals** present inside the class **Object**
    - 
    -  
* **Upcasting and Downcasting**
  + 
    - In here, even if the object is of type **B**, still you cannot call the method **showB** as the variable where it is getting stored is of type **A**.
    - In case of overriding, the method was present on **A** as well, so it just got overridden; but **showB** was not inside A; it is a completely new method for **A**; so it cannot be called.
    - It is **Upcasting**; assigning object of type Child to the variable of type Parent.
  + 
    - It’ll work fine.
    - We assigned the **obj** (which was of type **A**) to a variable of type **B**, but you need to explicitly **Downcast** this otherwise it’ll give error.
    - As **B** is the child class, so it can access both **showA** and **showB**.
* **Wrapper Class**
  + For all **primitive** type, there is a **Object** wrapper present in java.
  + For example: Integer, Double ..etc
  + 