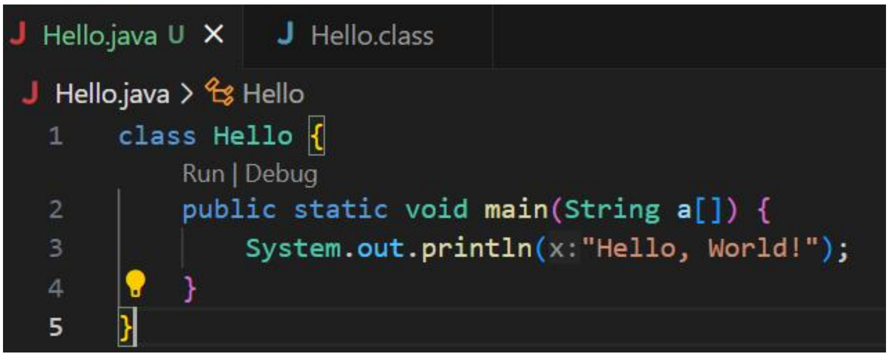
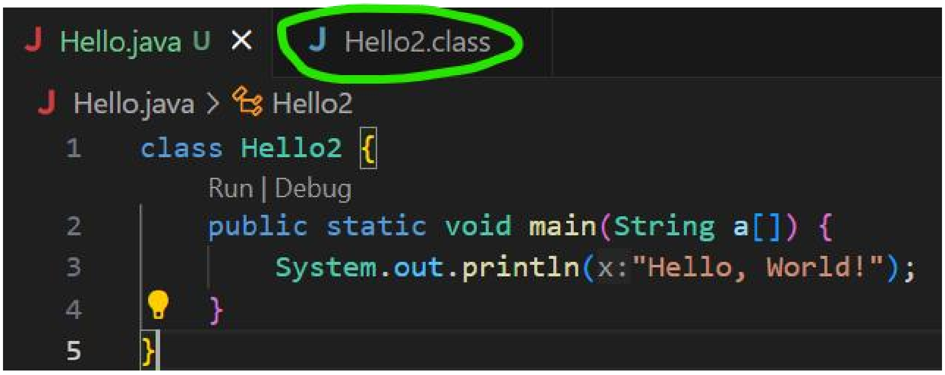
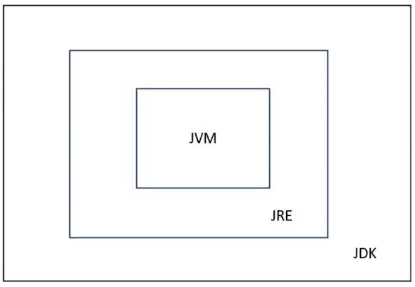
- **setx JAVA_HOME "C:\Program Files\Java\jdk-21"**
    - It is used to set the environment variable permanently.
- How the Java code works?
    - Java Code (**.java**) ------<compiler (javac)>------ Byte Code (**.class**)
    - Byte Code goes into JVM (JVM accepts only byte code)
    - JVM only run 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**.
    - 

```
class Hello {
    public static void main(String a[]) {
        System.out.println(x:"Hello, World!");
    }
}
```

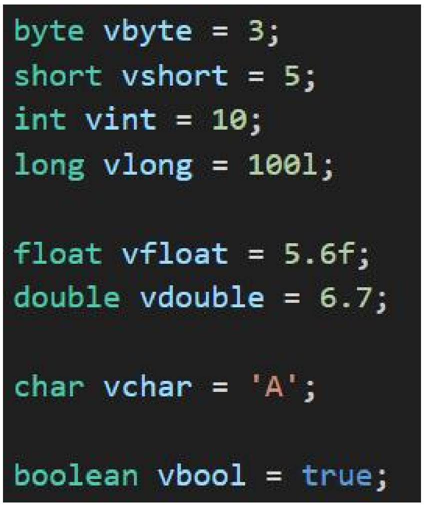
    - Here I gave the *classname* same as the *filename*.
    - But, it doesn't matter what is the filename; you can give the *classname* different.
    - But, the **.class** file will be created with the *classname* that is mentioned inside the **.java** file.
    - 

```
class Hello2 {
    public static void main(String a[]) {
        System.out.println(x:"Hello, World!");
    }
}
```

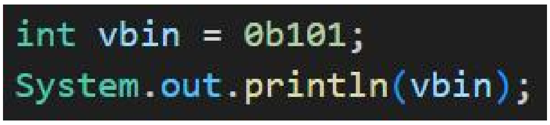
        - You can see here, I gave the class name *Hello2*, and the file got created is *Hello2.class*.

        - 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.
    - 

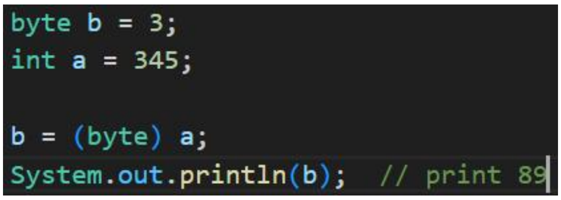
- Is JVM like a virtual machine on top of OS?
    - Yes and No:
    - ✅ Yes: It behaves like a virtual computer for Java bytecode. That's why Java is "Write Once, Run Anywhere."
    - ❌ No: It doesn't emulate hardware or run another OS. It sits on top of the real OS, using system calls, memory, CPU instructions, 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)
- 

```
byte vbyte = 3;
short vshort = 5;
int vint = 10;
long vlong = 100l;

float vfloat = 5.6f;
double vdouble = 6.7;

char vchar = 'A';

boolean vbool = true;
```

- NOTE: you can't double quotes in **char** variables. That is only for **string**.
- 

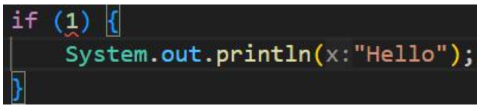
```
int vbin = 0b101;
System.out.println(vbin);
```

  - 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)

```
byte b = 3;
int a = 345;

b = (byte) a;
System.out.println(b);   // print 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).

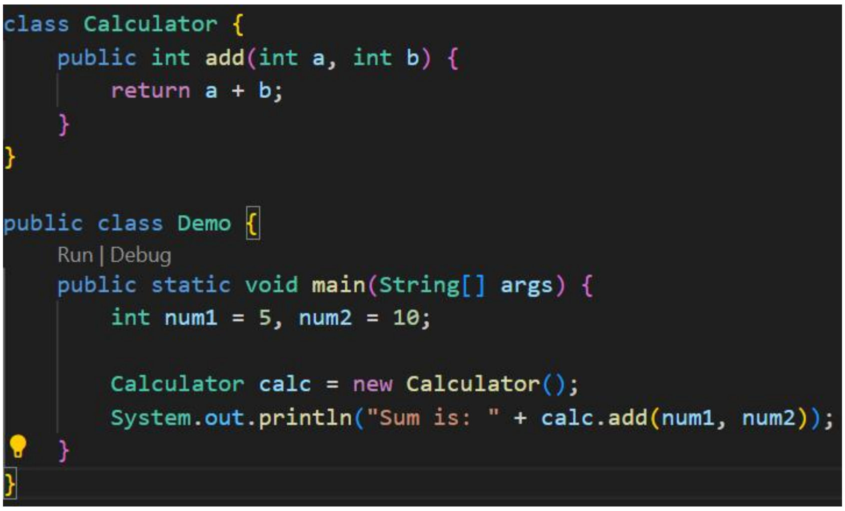
Some points to be remembered

- 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.
- 

```
if (1) {
    System.out.println(x:"Hello");
}
```

    - This is wrong. It'll give error that *can't convert int into boolean*.

Classes and Objects

- 

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}

public class Demo {
    public static void main(String[] args) {
        int num1 = 5, num2 = 10;

        Calculator calc = new Calculator();
        System.out.println("Sum is: " + calc.add(num1, num2));
    }
}
```

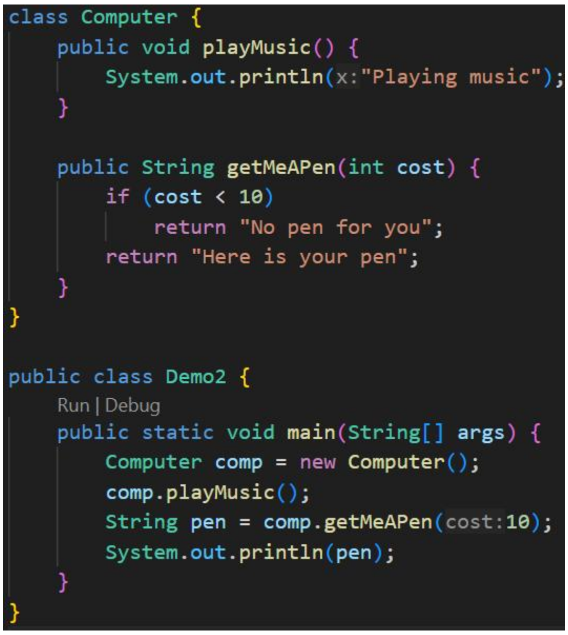
    - You can't run this directly using **java filename.java**, you need to compile and run separately.
    - Bcs, when you compile it, 2 .class files will be created i.e. Calculator.class and Demo.class.

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.

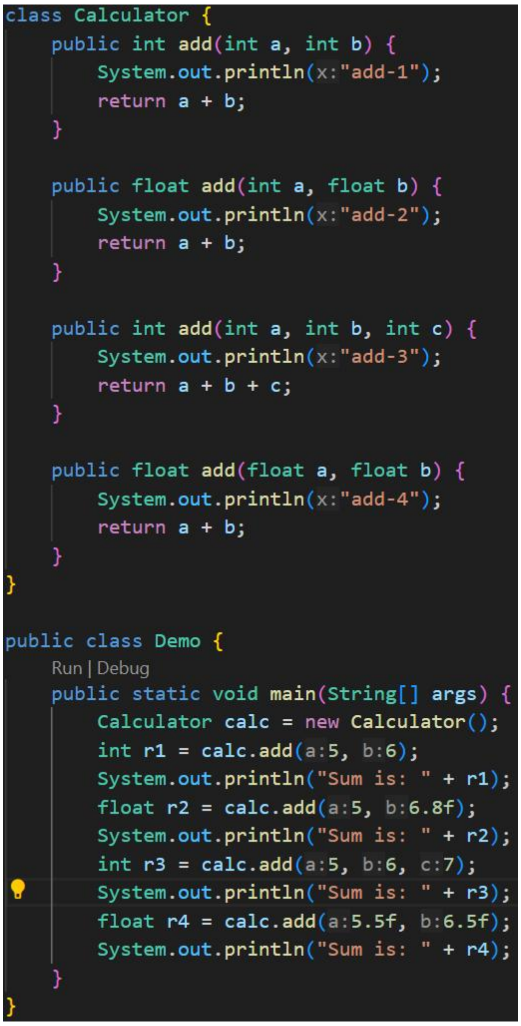
- **Methods**

  - While creating a method, you should provide a proper access modifier.

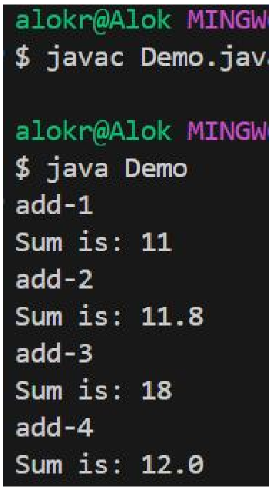


```
class Computer {
    public void playMusic() {
        System.out.println(x:"Playing music");
    }

    public String getMeAPen(int cost) {
        if (cost < 10)
            return "No pen for you";
        return "Here is your pen";
    }
}

public class Demo2 {
    Run | Debug
    public static void main(String[] args) {
        Computer comp = new Computer();
        comp.playMusic();
        String pen = comp.getMeAPen(cost:10);
        System.out.println(pen);
    }
}
```

  - **Method Overloading**



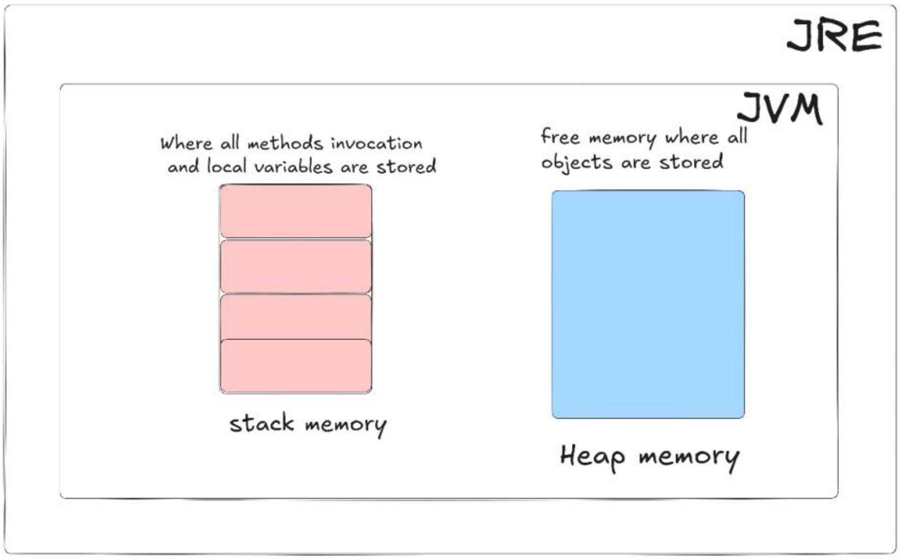
```
class Calculator {
    public int add(int a, int b) {
        System.out.println(x:"add-1");
        return a + b;
    }

    public float add(int a, float b) {
        System.out.println(x:"add-2");
        return a + b;
    }

    public int add(int a, int b, int c) {
        System.out.println(x:"add-3");
        return a + b + c;
    }

    public float add(float a, float b) {
        System.out.println(x:"add-4");
        return a + b;
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int r1 = calc.add(a:5, b:6);
        System.out.println("Sum is: " + r1);
        float r2 = calc.add(a:5, b:6.8f);
        System.out.println("Sum is: " + r2);
        int r3 = calc.add(a:5, b:6, c:7);
        System.out.println("Sum is: " + r3);
        float r4 = calc.add(a:5.5f, b:6.5f);
        System.out.println("Sum is: " + r4);
    }
}
```



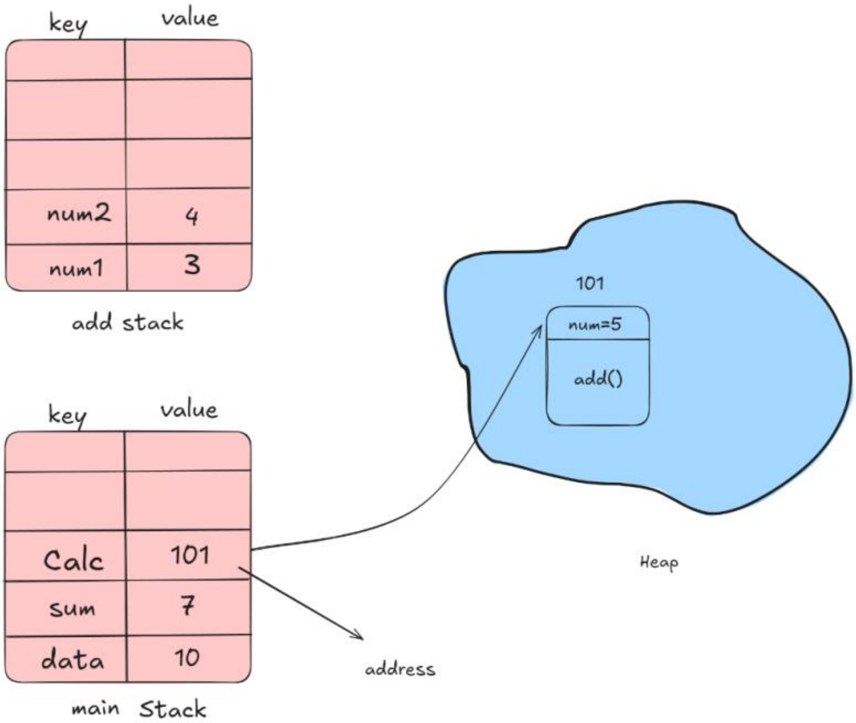
```
alokr@Alok MINGW
$ javac Demo.jav

alokr@Alok MINGW
$ java Demo
add-1
Sum is: 11
add-2
Sum is: 11.8
add-3
Sum is: 18
add-4
Sum is: 12.0
```

        - Number of arguments, type of arguments, type of return type: depending upon these, method overloading can be done.

- **<u>Stack and Heap</u>**
    - Inside JVM, there are 2 types of memory.
        - Stack (Last-In-First-Out)
        - Heap (open space)
    -


JRE
JVM
Where all methods invocation and local variables are stored
stack memory
free memory where all objects are stored
Heap memory

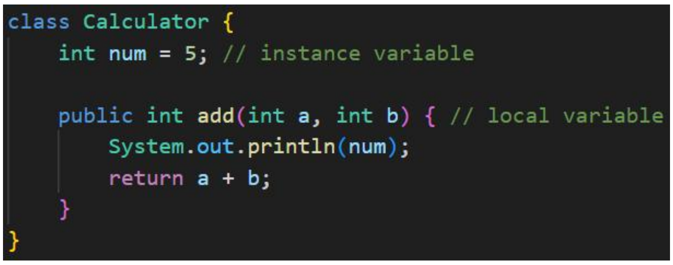
    -



key	value
num2	4
num1	3

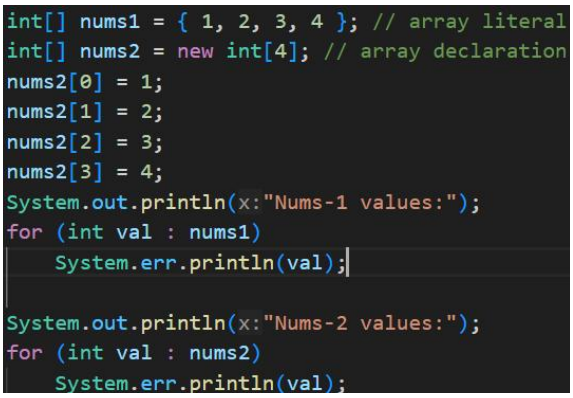
key	value
Calc	101
sum	7
data	10

        - 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.

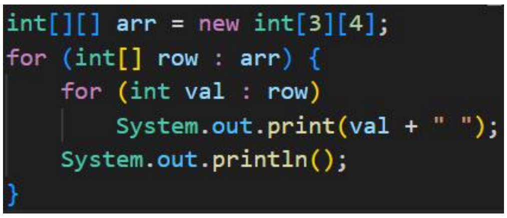
- Whenever you create an object out of a class, it is created inside the Heap.
  - In the heap, inside the memory block that the object acquires, is having 2 parts.
    - One is for **properties** (instance variables)
    - One is for **method definitions**
- When you call the method using the object, 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.
- 

```
class Calculator {
    int num = 5; // instance variable

    public int add(int a, int b) { // local variable
        System.out.println(num);
        return a + b;
    }
}
```

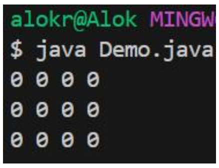
- Instance variables are specific to objects, not class. Means, each objects will have independent instance variables.
  - Because, each objects will be having different memory blocks inside the heap.
- The reference of the object inside heap, is stored inside the stack.

- In Java, everything, which is an object, gets created inside Heap.
  - Ex: Array,
- <u>**Array**</u>
  - **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.
  - 

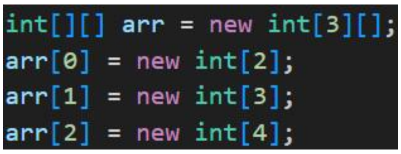
```
int[] nums1 = { 1, 2, 3, 4 }; // array literal
int[] nums2 = new int[4]; // array declaration
nums2[0] = 1;
nums2[1] = 2;
nums2[2] = 3;
nums2[3] = 4;
System.out.println(x:"Nums-1 values:");
for (int val : nums1)
    System.err.println(val);

System.out.println(x:"Nums-2 values:");
for (int val : nums2)
    System.err.println(val);
```

    - These are 2 types of Array declaration.
    - Both are **fixed sized** only.
    - An non-initialized array's values will be **0**.
- 2D Array:
    - 

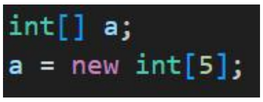
```
int[][] arr = new int[3][4];
for (int[] row : arr) {
    for (int val : row)
        System.out.print(val + " ");
    System.out.println();
}
```



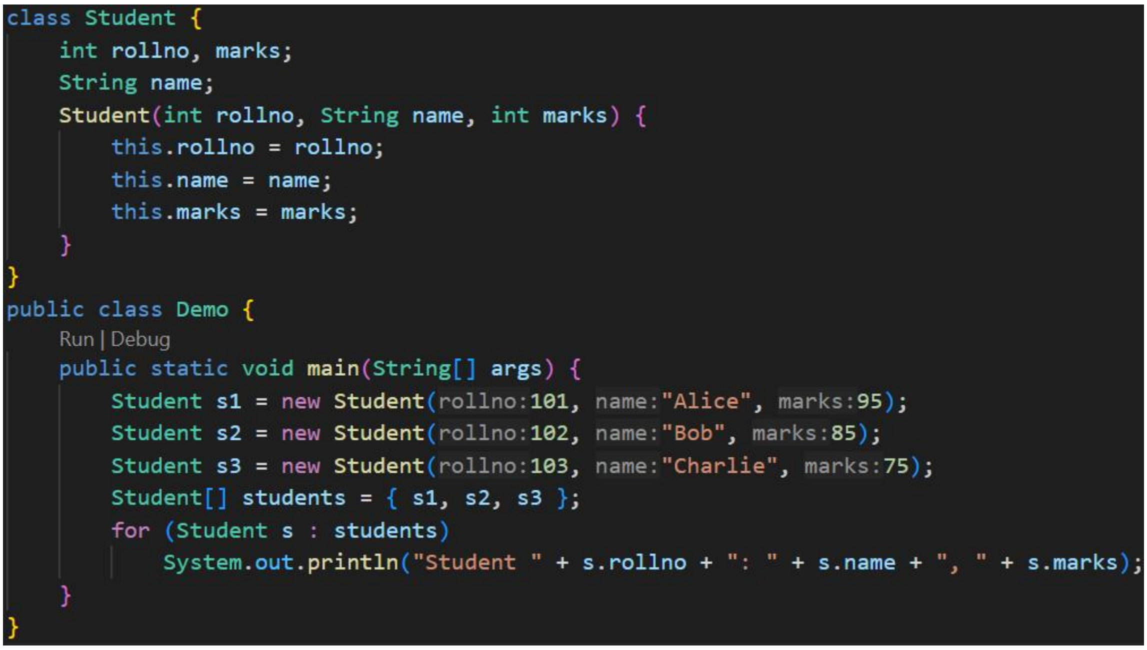
```
alokr@Alok MINGW
$ java Demo.java
0 0 0 0
0 0 0 0
0 0 0 0
```

    - 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.
    - 

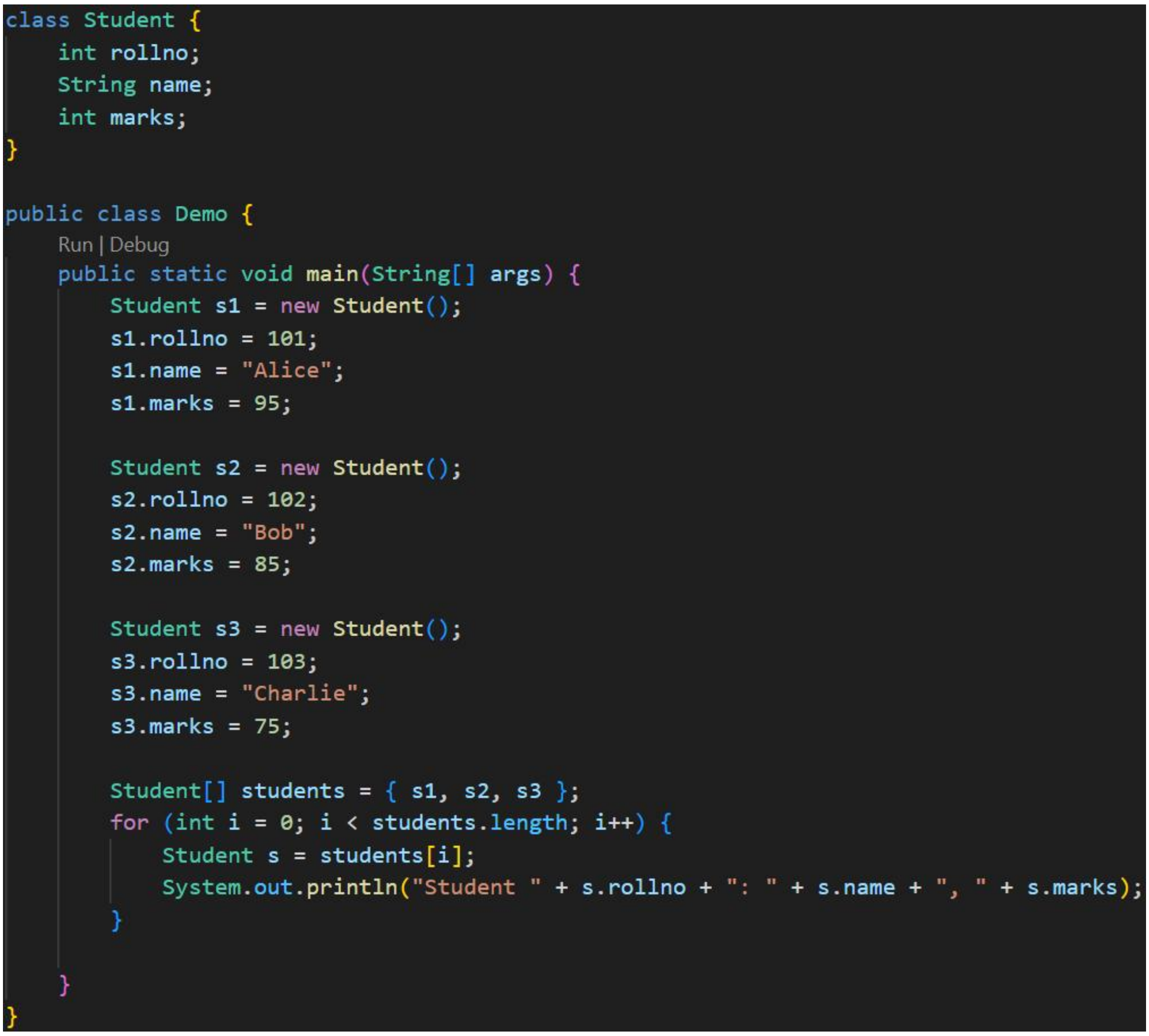
```
int[][] arr = new int[3][];
arr[0] = new int[2];
arr[1] = new int[3];
arr[2] = new int[4];
```

        - Just don't mention the column size while creating a 2D array and after that initialize the rows.
        - It is called jagged array.
        - 

```
int[] a;
a = new int[5];
```

            - 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:**
    - 

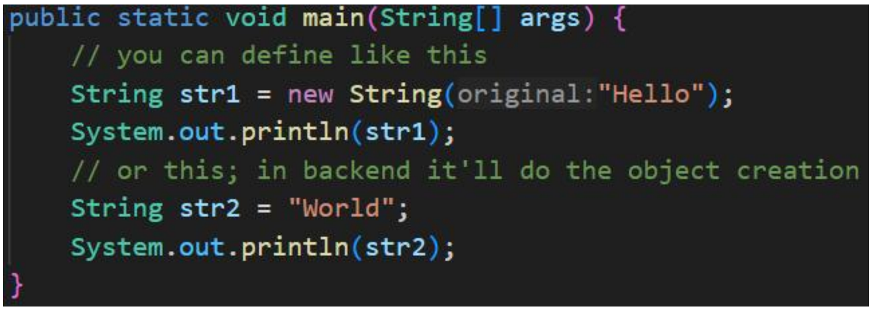
```
class Student {
    int rollno, marks;
    String name;
    Student(int rollno, String name, int marks) {
        this.rollno = rollno;
        this.name = name;
        this.marks = marks;
    }
}
public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Student s1 = new Student(rollno:101, name:"Alice", marks:95);
        Student s2 = new Student(rollno:102, name:"Bob", marks:85);
        Student s3 = new Student(rollno:103, name:"Charlie", marks:75);
        Student[] students = { s1, s2, s3 };
        for (Student s : students)
            System.out.println("Student " + s.rollno + ": " + s.name + ", " + s.marks);
    }
}
```



```java
class Student {
    int rollno;
    String name;
    int marks;
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.rollno = 101;
        s1.name = "Alice";
        s1.marks = 95;

        Student s2 = new Student();
        s2.rollno = 102;
        s2.name = "Bob";
        s2.marks = 85;

        Student s3 = new Student();
        s3.rollno = 103;
        s3.name = "Charlie";
        s3.marks = 75;

        Student[] students = { s1, s2, s3 };
        for (int i = 0; i < students.length; i++) {
            Student s = students[i];
            System.out.println("Student " + s.rollno + ": " + s.name + ", " + s.marks);
        }

    }
}
```

- **Drawbacks of Array:**
  - You can't change the size of the array.
  - O(n) for searching.

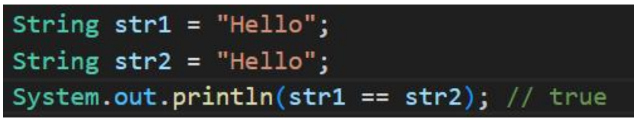
- <u>**Strings**</u>



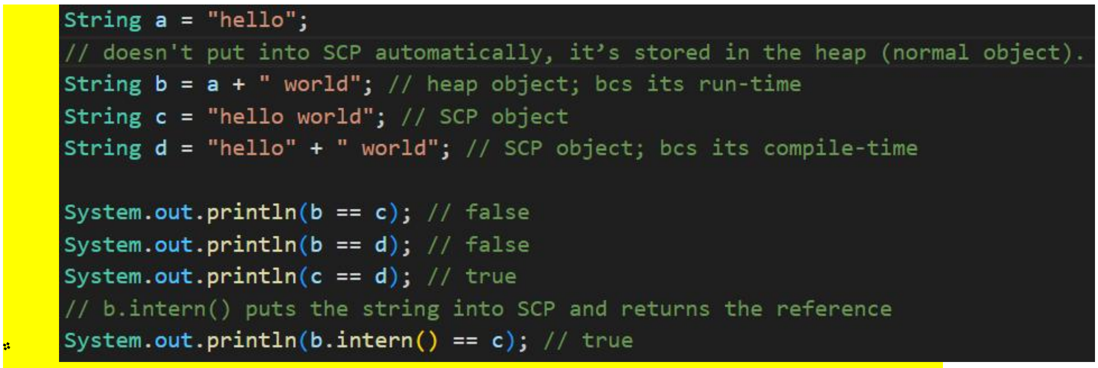
```java
public static void main(String[] args) {
    // you can define like this
    String str1 = new String(original:"Hello");
    System.out.println(str1);
    // or this; in backend it'll do the object creation
    String str2 = "World";
    System.out.println(str2);
}
```

- 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**,

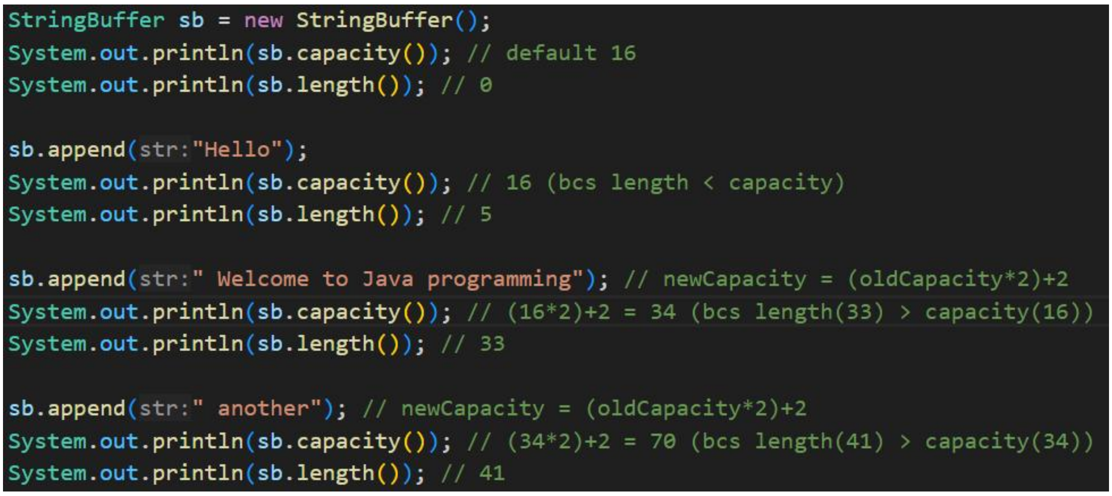
- Whenever you assign one *string value* to a variable, one *constant string literal* will get created inside that String Constant Pool and reference of that will be stored in a variable.

- **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.
  - 

```java
String str1 = "Hello";
String str2 = "Hello";
System.out.println(str1 == str2); // true
```

  - Whenever you assign one *string literal* to a variable, it first checks inside the **String Constant Pool (SCP)**;
    - 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:
    - 

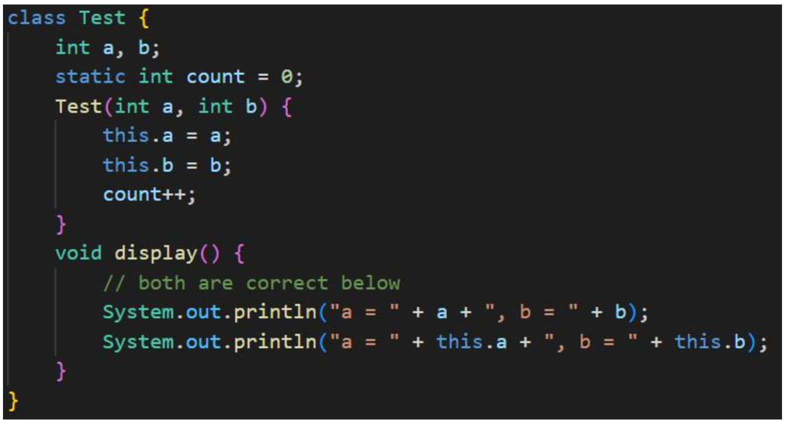
```java
String a = "hello";
// doesn't put into SCP automatically, it’s stored in the heap (normal object).
String b = a + " world"; // heap object; bcs its run-time
String c = "hello world"; // SCP object
String d = "hello" + " world"; // SCP object; bcs its compile-time

System.out.println(b == c); // false
System.out.println(b == d); // false
System.out.println(c == d); // true
// b.intern() puts the string into SCP and returns the reference
System.out.println(b.intern() == c); // true
```

- **Mutable Strings:**
  - **StringBuffer** is used to create mutable strings.
  - 

```java
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); // default 16
System.out.println(sb.length()); // 0

sb.append(str:"Hello");
System.out.println(sb.capacity()); // 16 (bcs length < capacity)
System.out.println(sb.length()); // 5

sb.append(str:" Welcome to Java programming"); // newCapacity = (oldCapacity*2)+2
System.out.println(sb.capacity()); // (16*2)+2 = 34 (bcs length(33) > capacity(16))
System.out.println(sb.length()); // 33

sb.append(str:" another"); // newCapacity = (oldCapacity*2)+2
System.out.println(sb.capacity()); // (34*2)+2 = 70 (bcs length(41) > capacity(34))
System.out.println(sb.length()); // 41
```

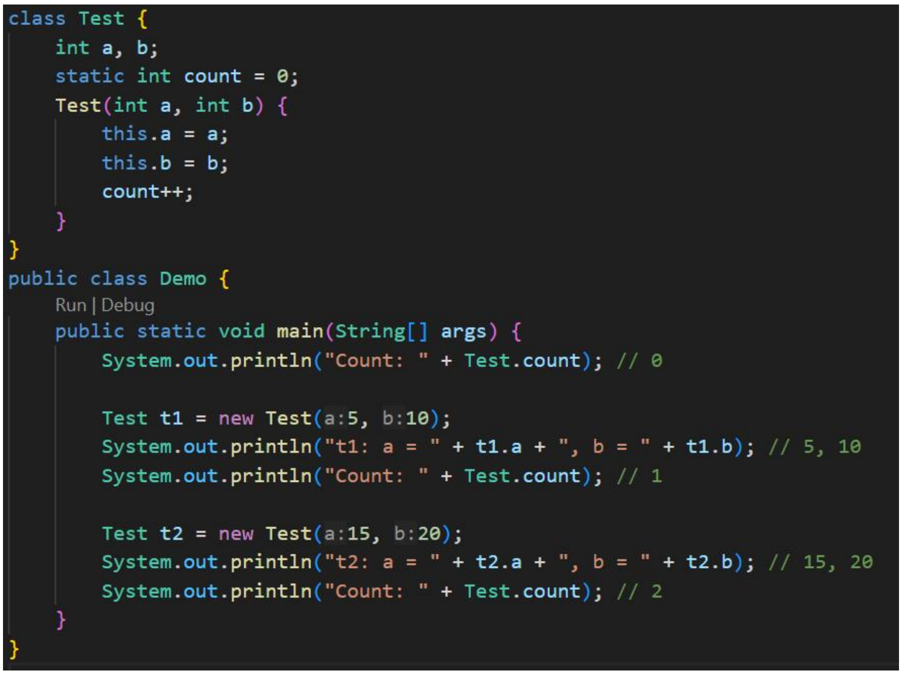
- By default, the capacity is **16+string length**.
- If the string size exceeds the capacity, then the capacity will be increased with the formula: **New Capacity = (Old Capacity) * 2 + 2**

- **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)



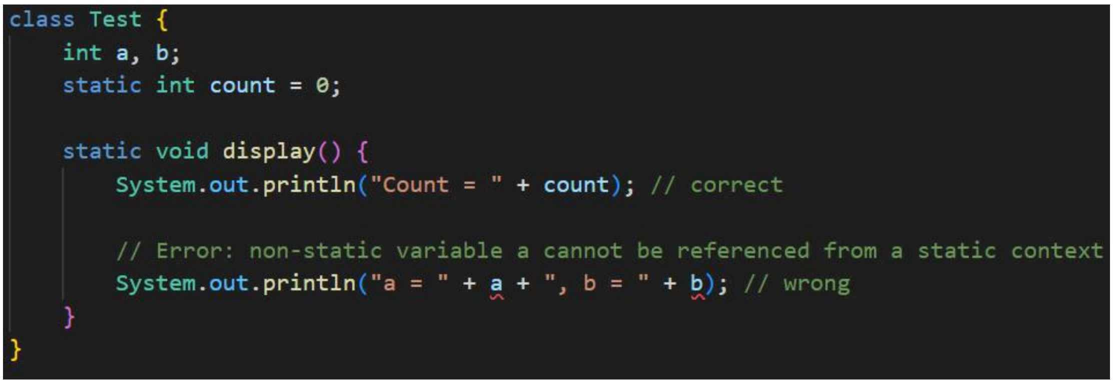
```
class Test {
    int a, b;
    static int count = 0;
    Test(int a, int b) {
        this.a = a;
        this.b = b;
        count++;
    }
    void display() {
        // both are correct below
        System.out.println("a = " + a + ", b = " + b);
        System.out.println("a = " + this.a + ", b = " + this.b);
    }
}
```

- **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/modify** the static variables using objects as well; but it is not preferable.
  - You should access the static variables using class only
    - **ClassName.StaticVariableName**

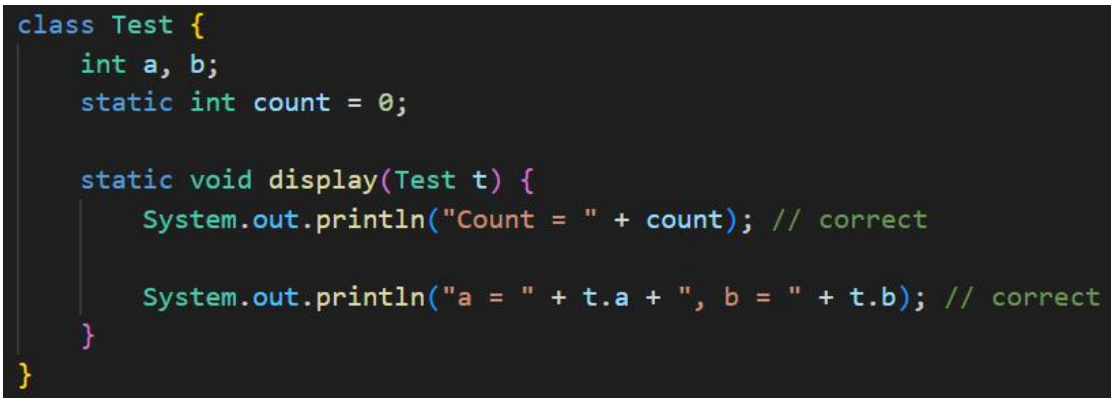


```
class Test {
    int a, b;
    static int count = 0;
    Test(int a, int b) {
        this.a = a;
        this.b = b;
        count++;
    }
}
public class Demo {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Count: " + Test.count); // 0

        Test t1 = new Test(a:5, b:10);
        System.out.println("t1: a = " + t1.a + ", b = " + t1.b); // 5, 10
        System.out.println("Count: " + Test.count); // 1

        Test t2 = new Test(a:15, b:20);
        System.out.println("t2: a = " + t2.a + ", b = " + t2.b); // 15, 20
        System.out.println("Count: " + Test.count); // 2
    }
}
```

- 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**.
  - 

```java
class Test {
    int a, b;
    static int count = 0;

    static void display() {
        System.out.println("Count = " + count); // correct

        // Error: non-static variable a cannot be referenced from a static context
        System.out.println("a = " + a + ", b = " + b); // wrong
    }
}
```

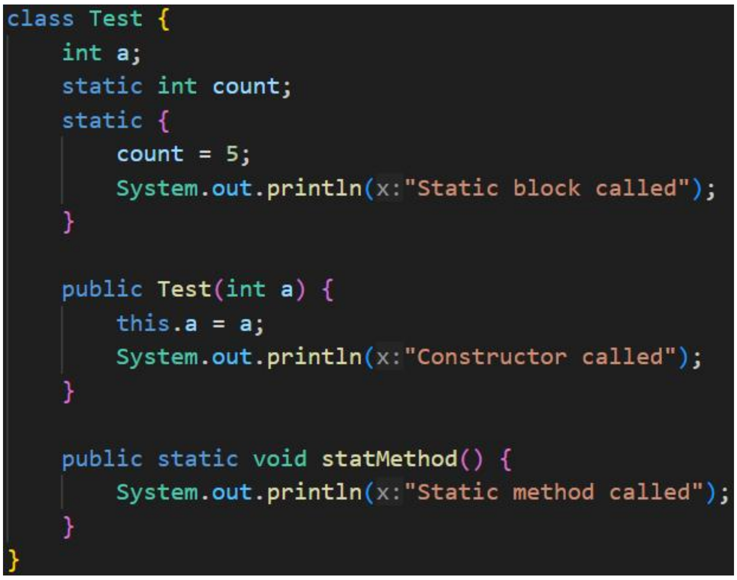
  - 

```java
class Test {
    int a, b;
    static int count = 0;

    static void display(Test t) {
        System.out.println("Count = " + count); // correct

        System.out.println("a = " + t.a + ", b = " + t.b); // correct
    }
}
```

    - 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.

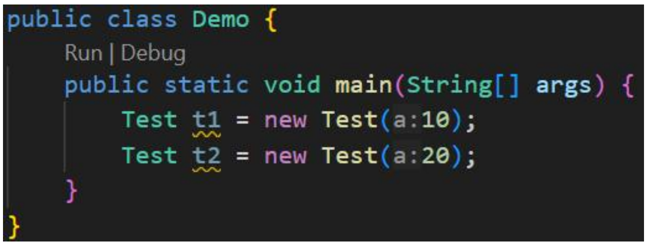
- <u>**Static Block**</u>
  - Whenever you instantiate a object with a class, then first the class gets loaded then the object will be created.
    - If you are instantiating more than one object with a single class, then **loading of class will happen only once.**
      - When the first object will 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.

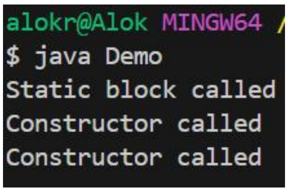


```
class Test {
    int a;
    static int count;
    static {
        count = 5;
        System.out.println(x:"Static block called");
    }

    public Test(int a) {
        this.a = a;
        System.out.println(x:"Constructor called");
    }

    public static void statMethod() {
        System.out.println(x:"Static method called");
    }
}
```

- This is the class having static block.

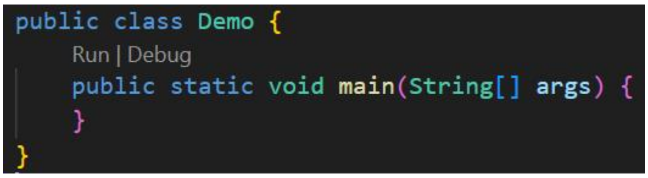


```
public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Test t1 = new Test(a:10);
        Test t2 = new Test(a:20);
    }
}
```

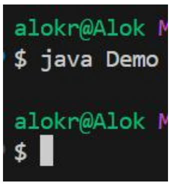


```
alokr@Alok MINGW64 /
$ java Demo
Static block called
Constructor called
Constructor called
```

- Because, class was loaded only once.

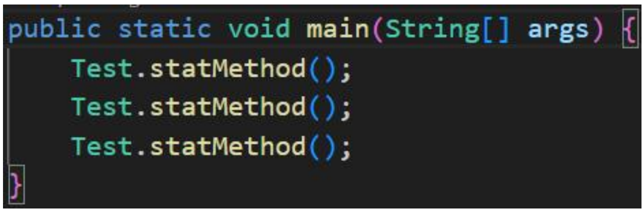


```
public class Demo {
    Run | Debug
    public static void main(String[] args) {
    }
}
```

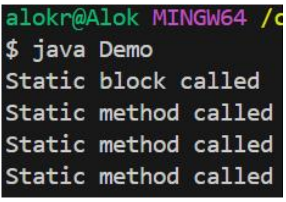


```
alokr@Alok M
$ java Demo

alokr@Alok M
$
```

- Because, as no object was created; so class loading didn't happen.

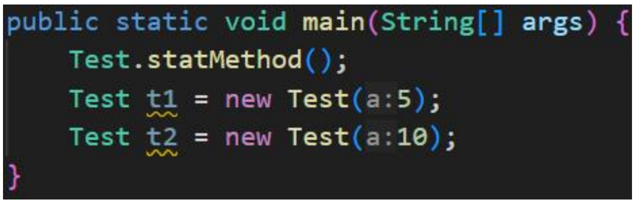


```
public static void main(String[] args) {
    Test.statMethod();
    Test.statMethod();
    Test.statMethod();
}
```

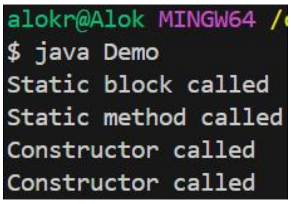


```
alokr@Alok MINGW64 /c
$ java Demo
Static block called
Static method called
Static method called
Static method called
```

- Because, only during the first static method call, class was loaded.

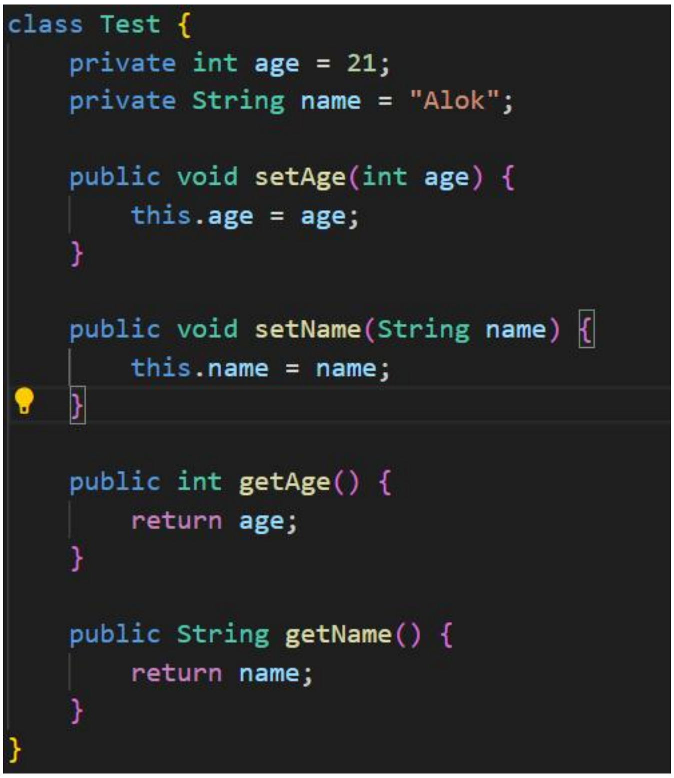


```
public static void main(String[] args) {
    Test.statMethod();
    Test t1 = new Test(a:5);
    Test t2 = new Test(a:10);
}
```



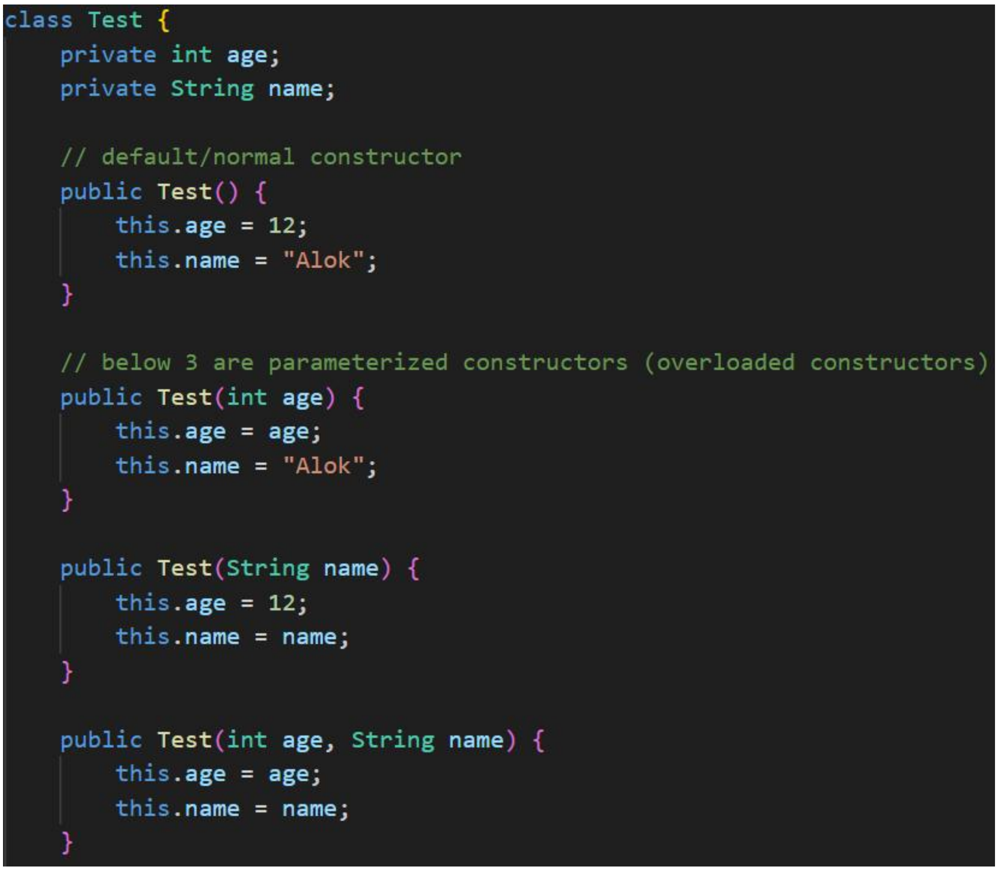
```
alokr@Alok MINGW64 /
$ java Demo
Static block called
Static method called
Constructor called
Constructor called
```

        - 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)
    - 

```
class Test {
    private int age = 21;
    private String name = "Alok";

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

        - 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.

- 

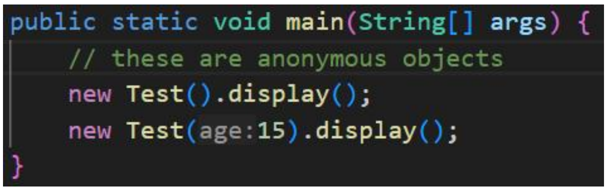
```java
class Test {
    private int age;
    private String name;

    // default/normal constructor
    public Test() {
        this.age = 12;
        this.name = "Alok";
    }

    // below 3 are parameterized constructors (overloaded constructors)
    public Test(int age) {
        this.age = age;
        this.name = "Alok";
    }

    public Test(String name) {
        this.age = 12;
        this.name = name;
    }

    public Test(int age, String name) {
        this.age = age;
        this.name = name;
    }
```

Naming Conventions

Class, Interfaces	: Pascal case (MyClass)
Variables, Methods	: Camel case (myVar)
Constants	: All capital (MY_CONST)

Anonymous Object

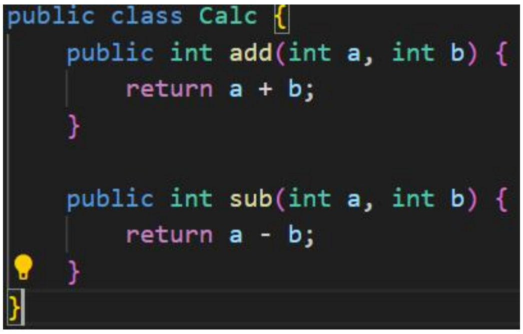
- It is just creating a object but not assigning it to any variable.

- 

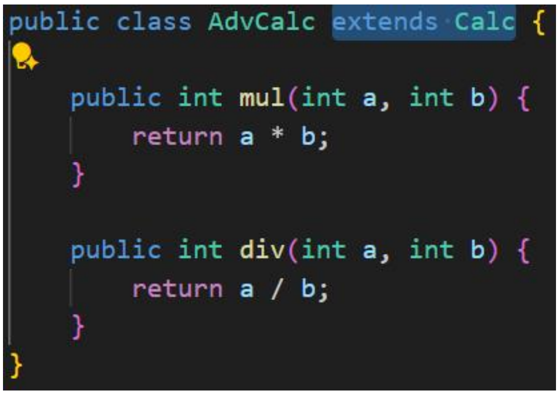
```java
public static void main(String[] args) {
    // these are anonymous objects
    new Test().display();
    new Test(age:15).display();
}
```

Inheritance

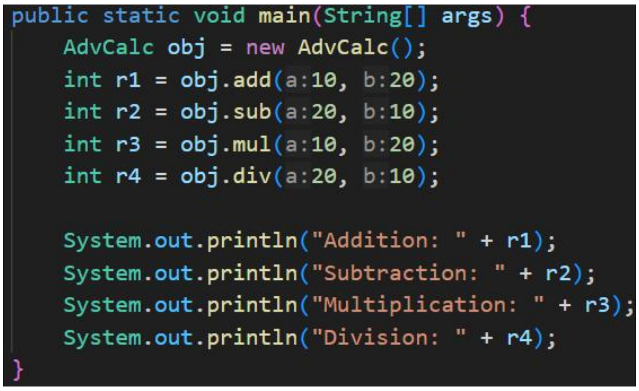
- **Single Level Inheritance:**



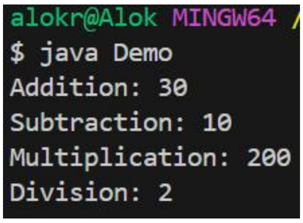
```java
public class Calc {
    public int add(int a, int b) {
        return a + b;
    }

    public int sub(int a, int b) {
        return a - b;
    }
}
```



```java
public class AdvCalc extends Calc {

    public int mul(int a, int b) {
        return a * b;
    }

    public int div(int a, int b) {
        return a / b;
    }
}
```

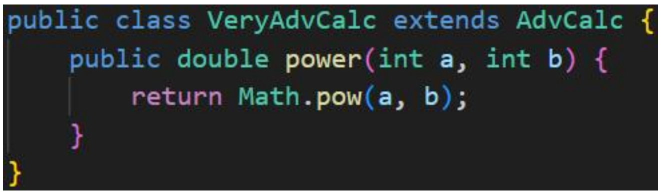


```java
public static void main(String[] args) {
    AdvCalc obj = new AdvCalc();
    int r1 = obj.add(a:10, b:20);
    int r2 = obj.sub(a:20, b:10);
    int r3 = obj.mul(a:10, b:20);
    int r4 = obj.div(a:20, b:10);

    System.out.println("Addition: " + r1);
    System.out.println("Subtraction: " + r2);
    System.out.println("Multiplication: " + r3);
    System.out.println("Division: " + r4);
}
```



```
alokr@Alok MINGW64 /
$ java Demo
Addition: 30
Subtraction: 10
Multiplication: 200
Division: 2
```

- **Multi Level Inheritance**

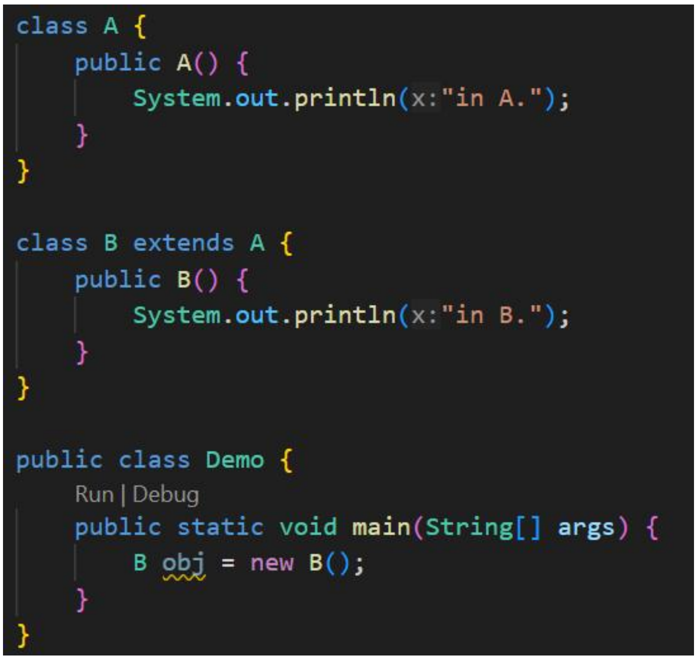


```java
public class VeryAdvCalc extends AdvCalc {
    public double power(int a, int b) {
        return Math.pow(a, b);
    }
}
```

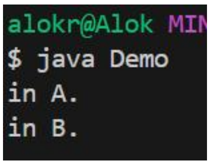
  - Now: VeryAdvCalc >> AdvCalc >> Calc

- **Multiple Inheritance**
  - Java doesn't support multiple inheritance.

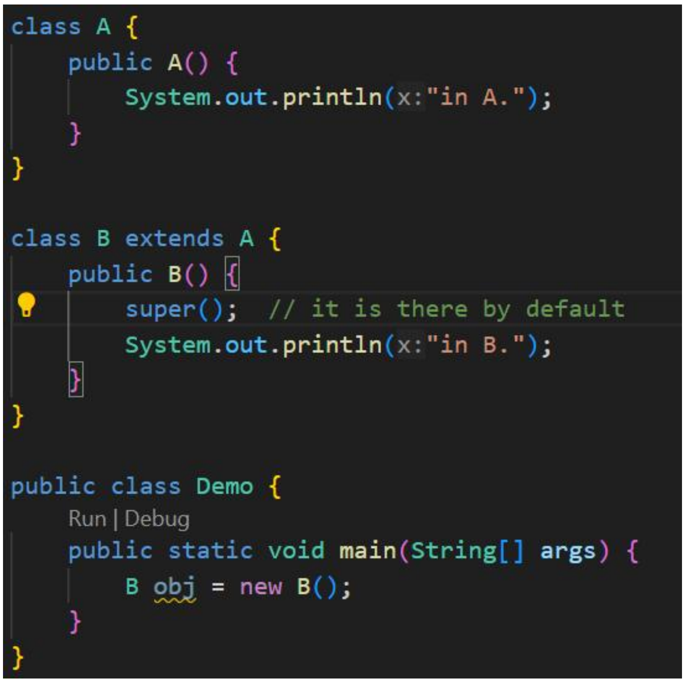
- **super method**
  - **super** is nothing but the alias of parent class's constructor.
    - When you write **super()**, it'll call the parent class's default (non-parameterized) constructor.
  - When a class inherits another class, even if you don't write **super()** inside the constructor of the child class, java executes **super()** by default.
  - **super.method_or_var_name** (here non static methods/variables are only valid)
  - **Super()** (calls the parent class's constructor)

- 

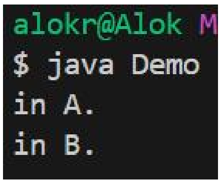
```java
class A {
    public A() {
        System.out.println(x:"in A.");
    }
}

class B extends A {
    public B() {
        System.out.println(x:"in B.");
    }
}

public class Demo {
    public static void main(String[] args) {
        B obj = new B();
    }
}
```



```
alokr@Alok MIN
$ java Demo
in A.
in B.
```

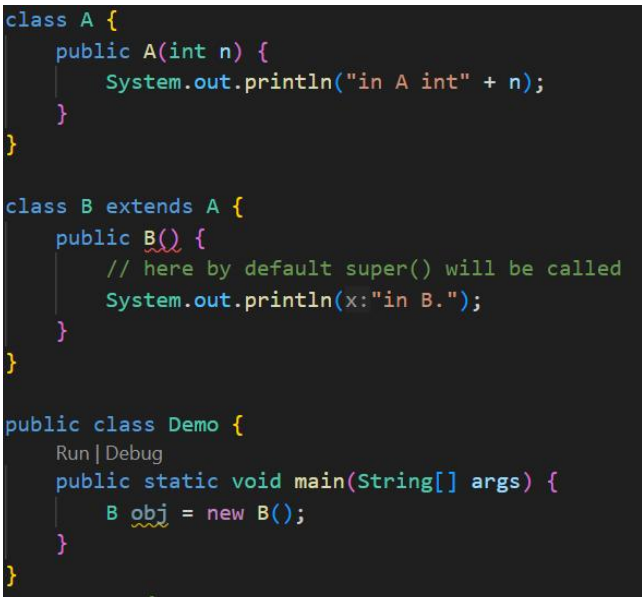
- 

```java
class A {
    public A() {
        System.out.println(x:"in A.");
    }
}

class B extends A {
    public B() {
        super();   // it is there by default
        System.out.println(x:"in B.");
    }
}

public class Demo {
    public static void main(String[] args) {
        B obj = new B();
    }
}
```

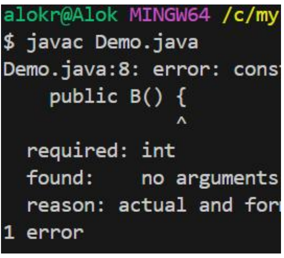


```
alokr@Alok M
$ java Demo
in A.
in B.
```

  - 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.

- 

```
class A {
    public A(int n) {
        System.out.println("in A int" + n);
    }
}

class B extends A {
    public B() {
        // here by default super() will be called
        System.out.println(x:"in B.");
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        B obj = new B();
    }
}
```

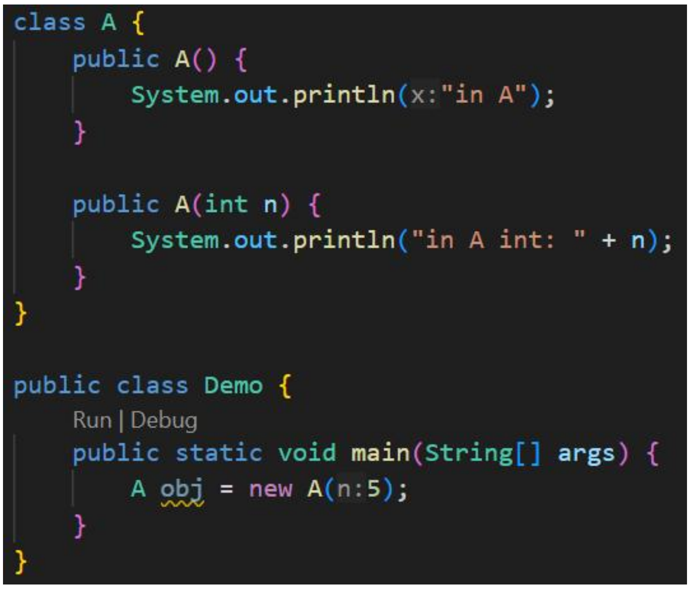


```
alokr@Alok MINGW64 /c/my
$ javac Demo.java
Demo.java:8: error: cons
    public B() {
               ^
  required: int
  found:     no arguments
  reason: actual and for
1 error
```

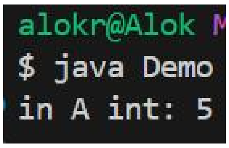
  - In this case, you need to call the parent's parameterized constructor explicitly.

➢ **this method**

- **this** refers to the current object instance; when you call **this()** it'll call the current class's constructor.

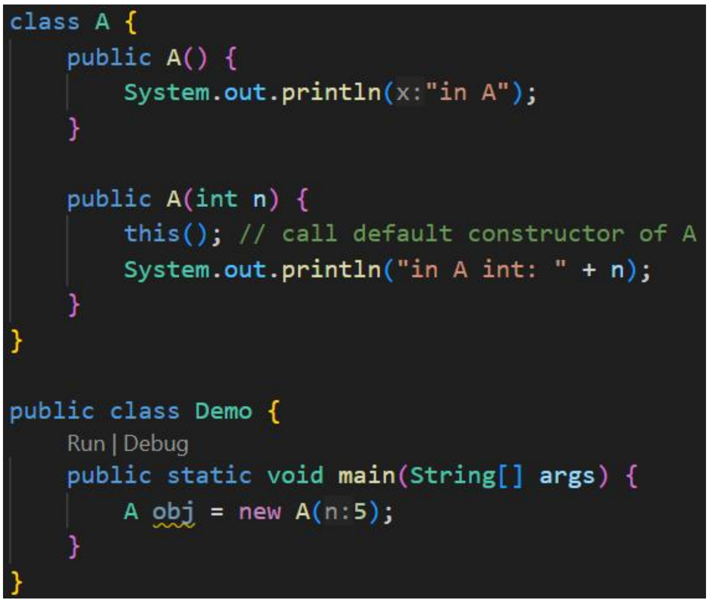
- 

```
class A {
    public A() {
        System.out.println(x:"in A");
    }

    public A(int n) {
        System.out.println("in A int: " + n);
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A obj = new A(n:5);
    }
}
```

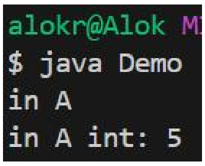


```
alokr@Alok M
$ java Demo
in A int: 5
```

  - 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.

- 

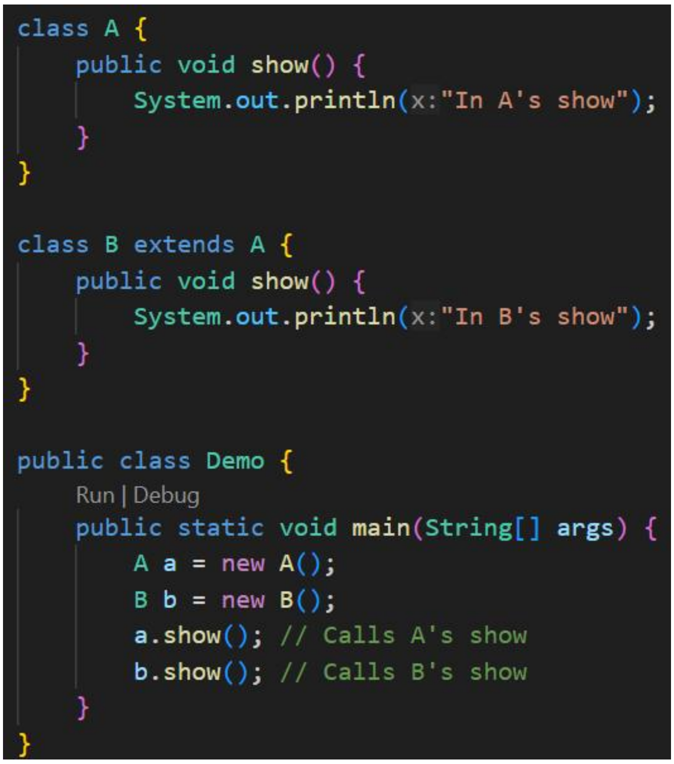
```
class A {
    public A() {
        System.out.println(x:"in A");
    }

    public A(int n) {
        this(); // call default constructor of A
        System.out.println("in A int: " + n);
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A obj = new A(n:5);
    }
}
```



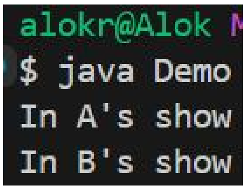
```
alokr@Alok M
$ java Demo
in A
in A int: 5
```

  - Now, as we called **this()** inside the parameterized constructor, so its calling the default constructor inside the class.

Method Overriding

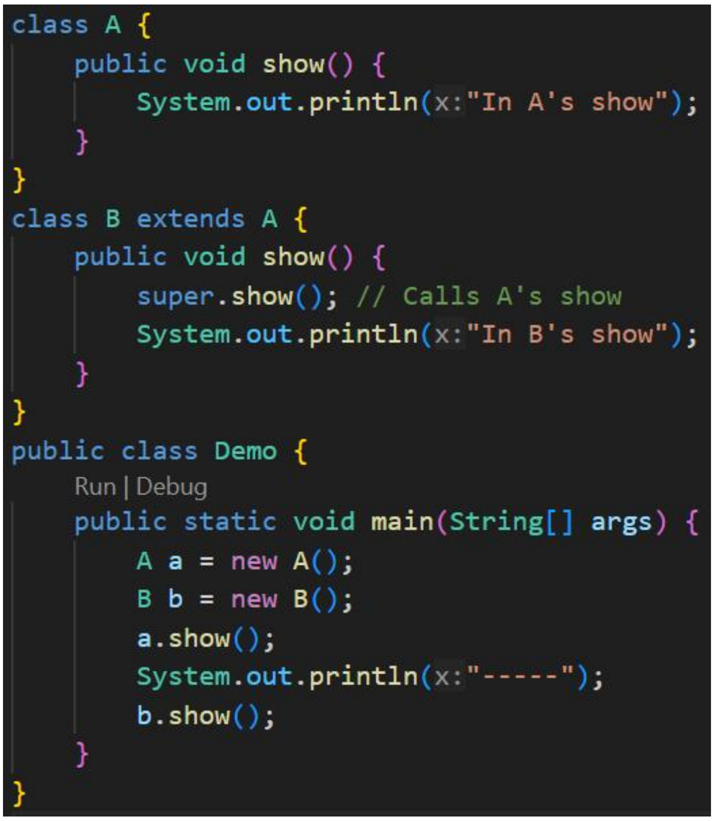
- When child class implements same method which is present in the parent's class as well; its called Method Overriding.
- **Same name, same type of arguments, same number of arguments, same return type; just different definition.** (unlike method overloading: name/type/number of args/return type => one or more of these should be different)
- 

```
class A {
    public void show() {
        System.out.println(x:"In A's show");
    }
}

class B extends A {
    public void show() {
        System.out.println(x:"In B's show");
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.show(); // Calls A's show
        b.show(); // Calls B's show
    }
}
```

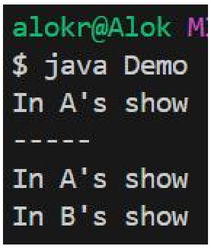


```
alokr@Alok M
$ java Demo
In A's show
In B's show
```

  - Its simple only; B's show function is overriding A's show function.

- 

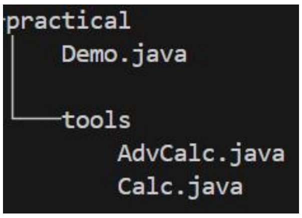
```java
class A {
    public void show() {
        System.out.println(x:"In A's show");
    }
}
class B extends A {
    public void show() {
        super.show(); // Calls A's show
        System.out.println(x:"In B's show");
    }
}
public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.show();
        System.out.println(x:"-----");
        b.show();
    }
}
```



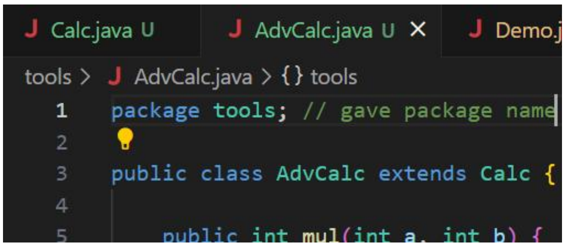
```
alokr@Alok MI
$ java Demo
In A's show
-----
In A's show
In B's show
```

  - 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.

➢ <u>**Packages**</u>

- 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.
- 

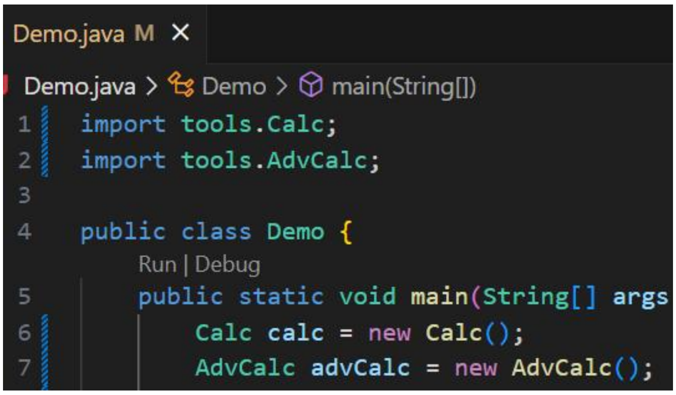
```
practical
│   Demo.java
│
└───tools
        AdvCalc.java
        Calc.java
```

  - It is my current directory structure; I kept *Calculator* related stuffs inside the **tools** directory
- 

```java
package tools; // gave package name

public class AdvCalc extends Calc {

    public int mul(int a, int b) {
```

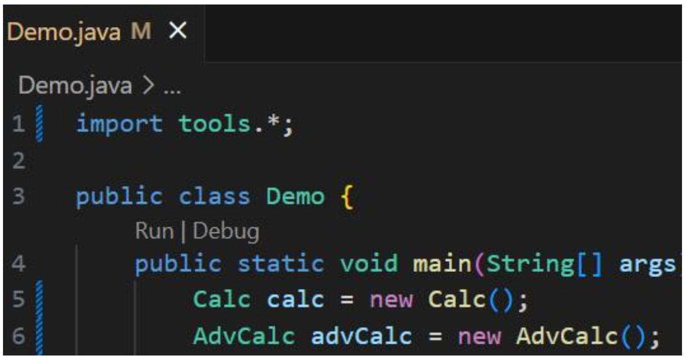
  - Gave the package name as **tools** in both **Calc.java** and **AdvCalc.java**

- 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.



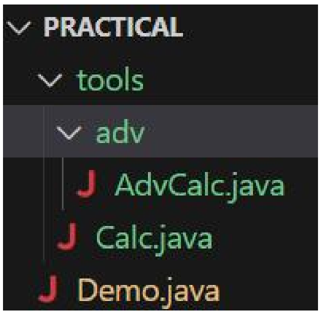
```
import tools.Calc;
import tools.AdvCalc;

public class Demo {
    public static void main(String[] args
        Calc calc = new Calc();
        AdvCalc advCalc = new AdvCalc();
```

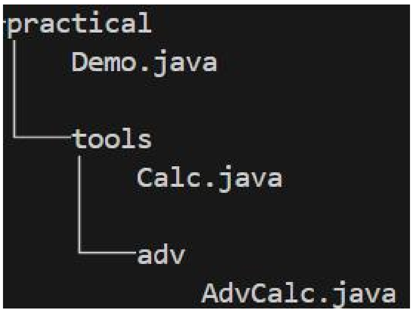
- Instead of importing the modules one by one; we can import all the modules present inside that package directly using *



```
import tools.*;

public class Demo {
    public static void main(String[] args
        Calc calc = new Calc();
        AdvCalc advCalc = new AdvCalc();
```

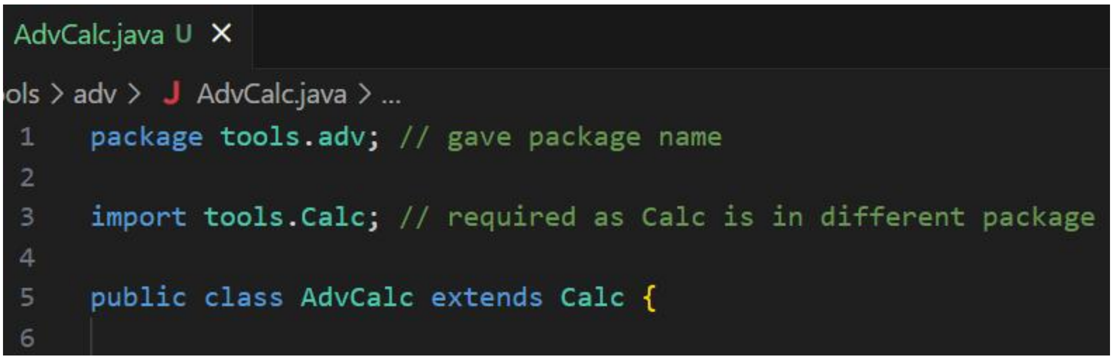
- Now lets see nested package structure:





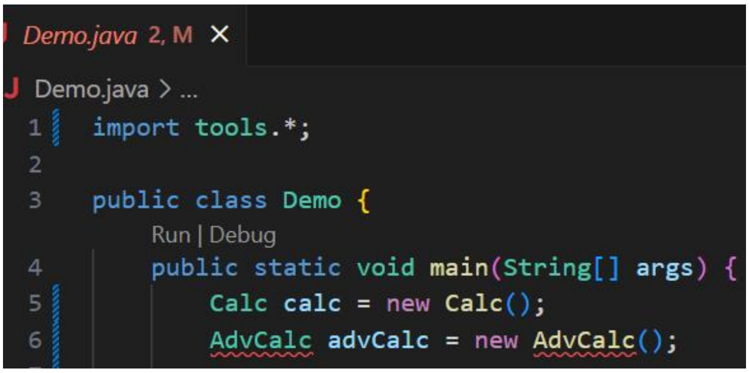
```
practical
│   Demo.java
│
└───tools
    │   Calc.java
    │
    └───adv
            AdvCalc.java
```

  - I transferred **AdvCalc.java** inside a new directory ***adv***.



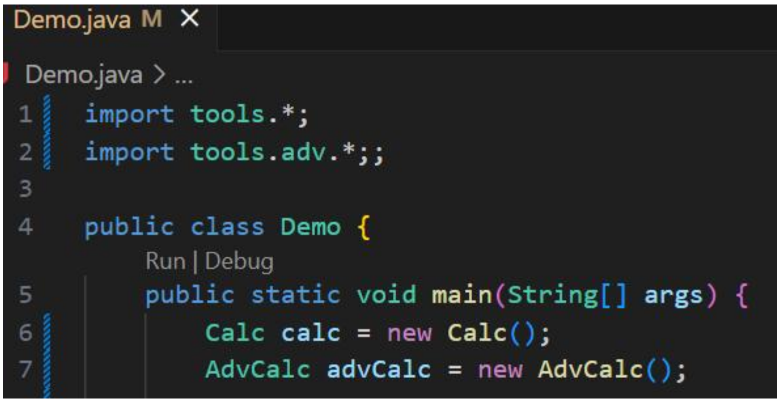
```
package tools.adv; // gave package name

import tools.Calc; // required as Calc is in different package

public class AdvCalc extends Calc {

```

  - Now we need to import Calc as well as it is in different package now.

- 

```java
import tools.*;

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Calc calc = new Calc();
        AdvCalc advCalc = new AdvCalc();
```

  - 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.

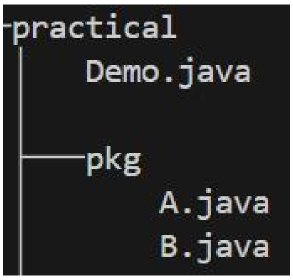
- 

```java
import tools.*;
import tools.adv.*;;

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Calc calc = new Calc();
        AdvCalc advCalc = new AdvCalc();
```

  - Now it'll work properly.

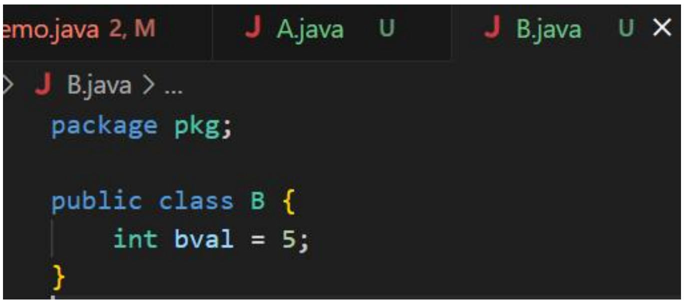
- You can't give different package names to the files which are siblings to each other i.e. present inside the same folder.

➢ <u>**Access Modifiers**</u>

- 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.
  - 

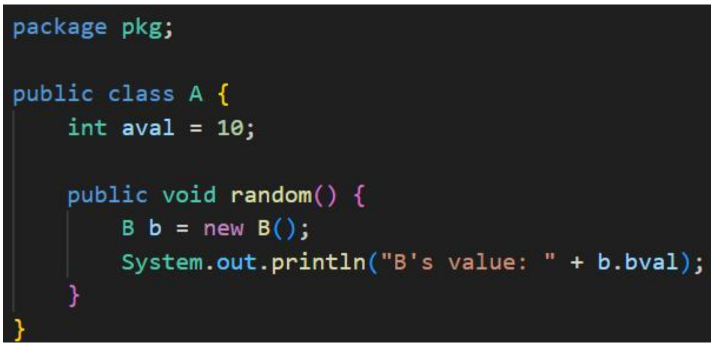
```
practical
│   Demo.java
│
├───pkg
│       A.java
│       B.java
```

(This is my file structure for now)

- 

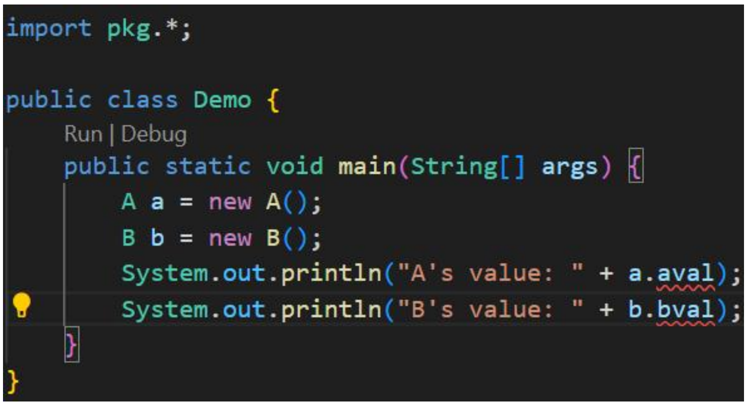
```java
package pkg;

public class B {
    int bval = 5;
}
```

  - (practical/pkg/B.java)

- 

```java
package pkg;

public class A {
    int aval = 10;

    public void random() {
        B b = new B();
        System.out.println("B's value: " + b.bval);
    }
}
```

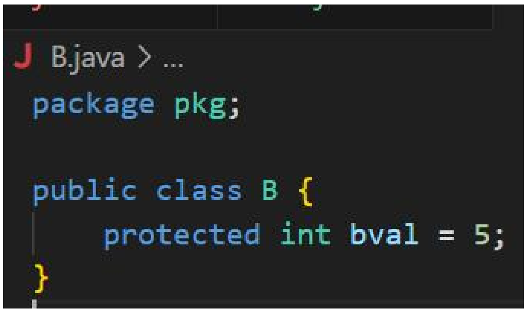
  - (practical/pkg/A.java)
  - You can see here, there is no error while accessing the variable of class B.

- 

```java
import pkg.*;

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        System.out.println("A's value: " + a.aval);
        System.out.println("B's value: " + b.bval);
    }
}
```

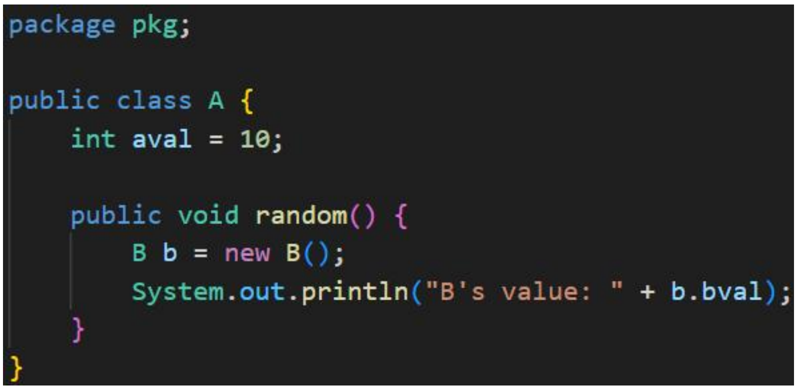
  - (practical/Demo.java)
  - Here, error is coming as **Demo.java** is not inside the package of **A** and **B**.

- **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).



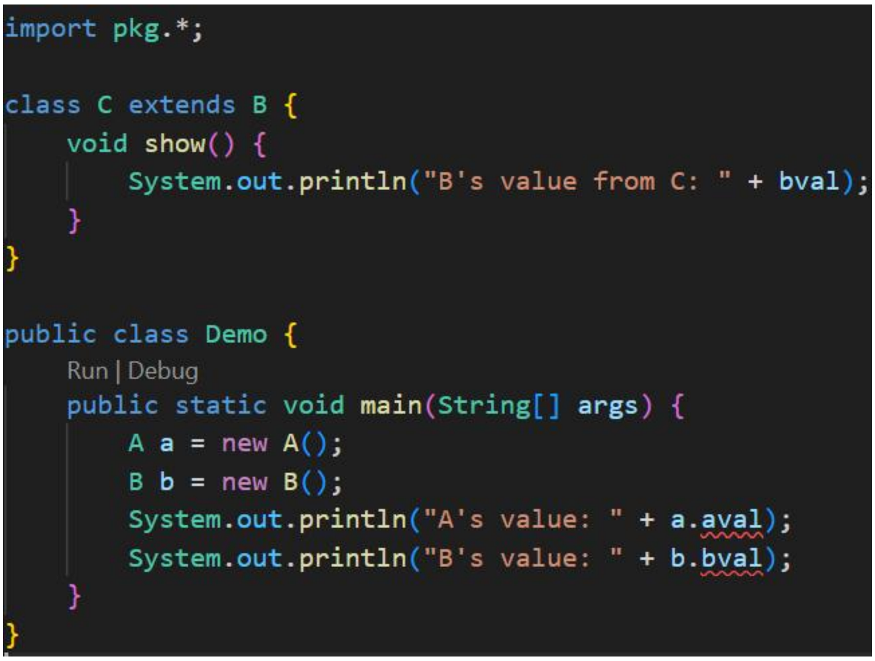
```java
package pkg;

public class B {
    protected int bval = 5;
}
```

- (practical/pkg/B.java)



```java
package pkg;

public class A {
    int aval = 10;

    public void random() {
        B b = new B();
        System.out.println("B's value: " + b.bval);
    }
}
```

- (practical/pkg/A.java)
- No error because A and B are inside the same package.

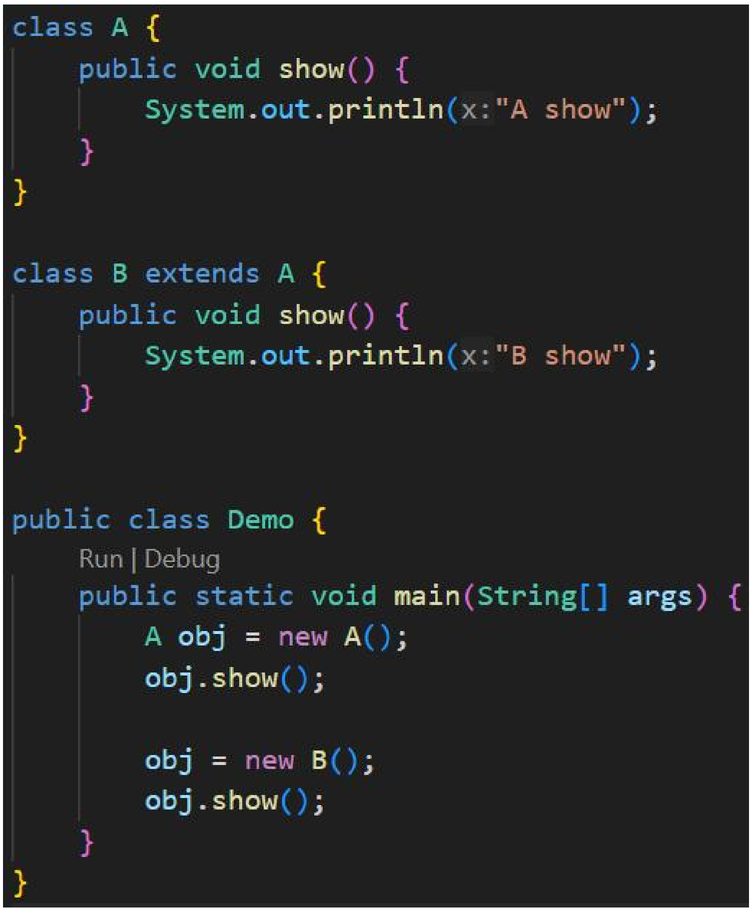


```java
import pkg.*;

class C extends B {
    void show() {
        System.out.println("B's value from C: " + bval);
    }
}

public class Demo {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        System.out.println("A's value: " + a.aval);
        System.out.println("B's value: " + b.bval);
    }
}
```

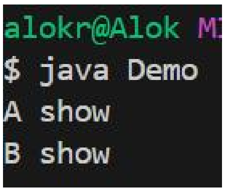
- (practical/Demo.java)
- **Class C** doesn’t give any error as it is inheriting **Class B**
- But, inside **Class Demo** it is giving error because neither it is inheriting those classes not it is present inside the same package where **A** and **B** are present.

- **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)**.
    - 

```java
class A {
    public void show() {
        System.out.println(x:"A show");
    }
}

class B extends A {
    public void show() {
        System.out.println(x:"B show");
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A obj = new A();
        obj.show();

        obj = new B();
        obj.show();
    }
}
```

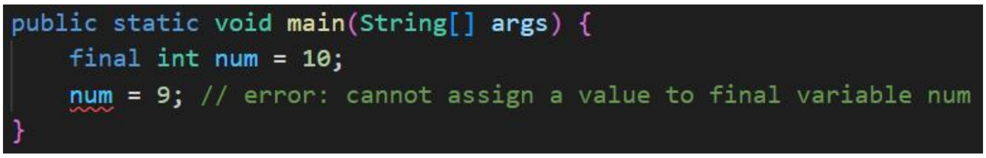


```
alokr@Alok MI
$ java Demo
A show
B show
```

        - Here, the variable is of type **A** (parent class).

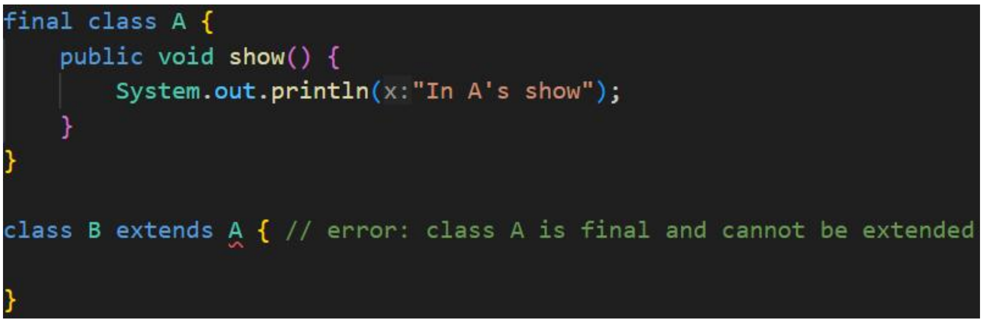
    - During the compile time, it is not sure that which show method will be called. It'll be decided during the run time only.
        - The method, that will be called, depends upon the **type of object**, not **type of variable**. **(it is only for overriding; for more check Upcasting *and Downcasting*)**

- **Final Keyword**
    - Its used to create a **constant** variable.
    - Can be used to create **variable, method, class**
    - **variable**



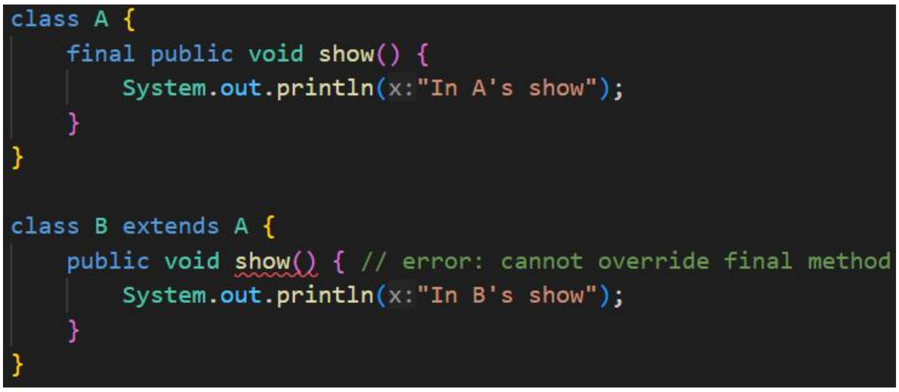
```
public static void main(String[] args) {
    final int num = 10;
    num = 9; // error: cannot assign a value to final variable num
}
```

    - **class**
        - When you make your class **final**, means you are stopping further inheritance.



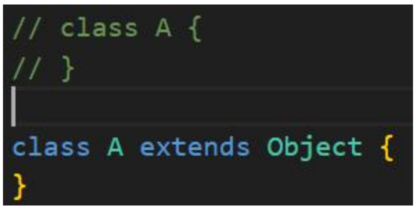
```
final class A {
    public void show() {
        System.out.println(x:"In A's show");
    }
}

class B extends A { // error: class A is final and cannot be extended

}
```

    - **method**
        - When you make the method **final**, means this method cannot be overridden.



```
class A {
    final public void show() {
        System.out.println(x:"In A's show");
    }
}

class B extends A {
    public void show() { // error: cannot override final method
        System.out.println(x:"In B's show");
    }
}
```

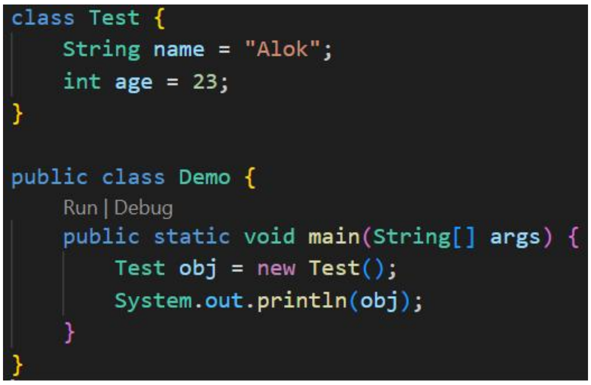
- By default, every classes inherits the class **Object**.

- 

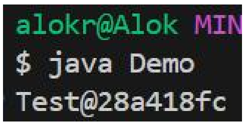
```
// class A {
// }

class A extends Object {
}
```

- These both are same only.

➢ Some methods inside Object Class (toString, equals, hashcode)

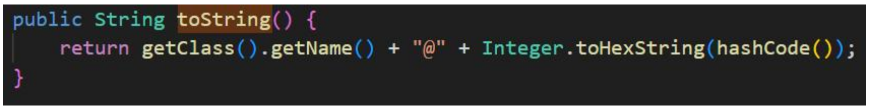
- 

```
class Test {
    String name = "Alok";
    int age = 23;
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Test obj = new Test();
        System.out.println(obj);
    }
}
```



```
alokr@Alok MIN
$ java Demo
Test@28a418fc
```

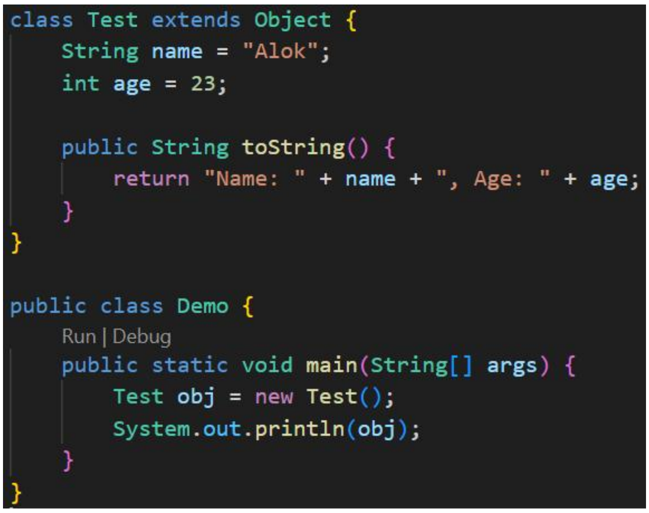
- When I printed this object of class Test, it gave some output like this.

- 

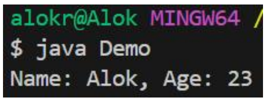
```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- That output is because of this method present inside the class **Object**

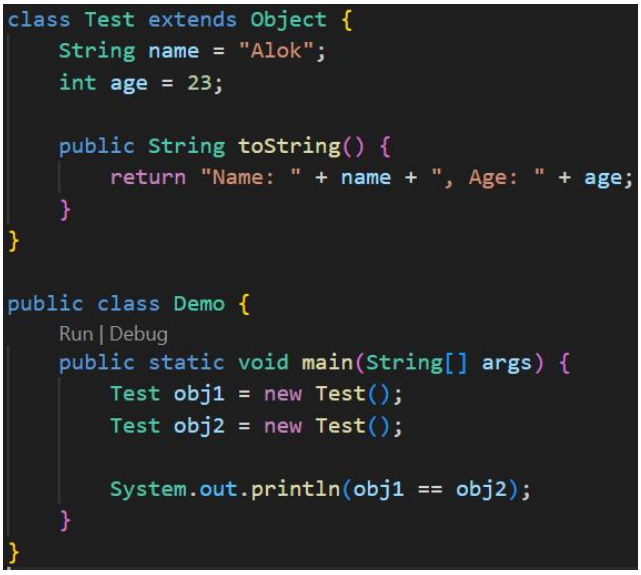
- We can modify that.

- 

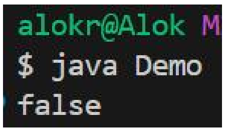
```
class Test extends Object {
    String name = "Alok";
    int age = 23;

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Test obj = new Test();
        System.out.println(obj);
    }
}
```



```
alokr@Alok MINGW64 /
$ java Demo
Name: Alok, Age: 23
```

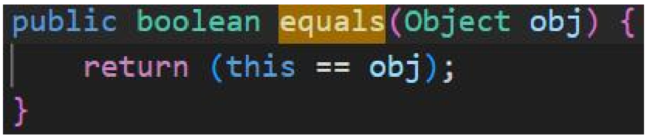


```java
class Test extends Object {
    String name = "Alok";
    int age = 23;

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Test obj1 = new Test();
        Test obj2 = new Test();

        System.out.println(obj1 == obj2);
    }
}
```

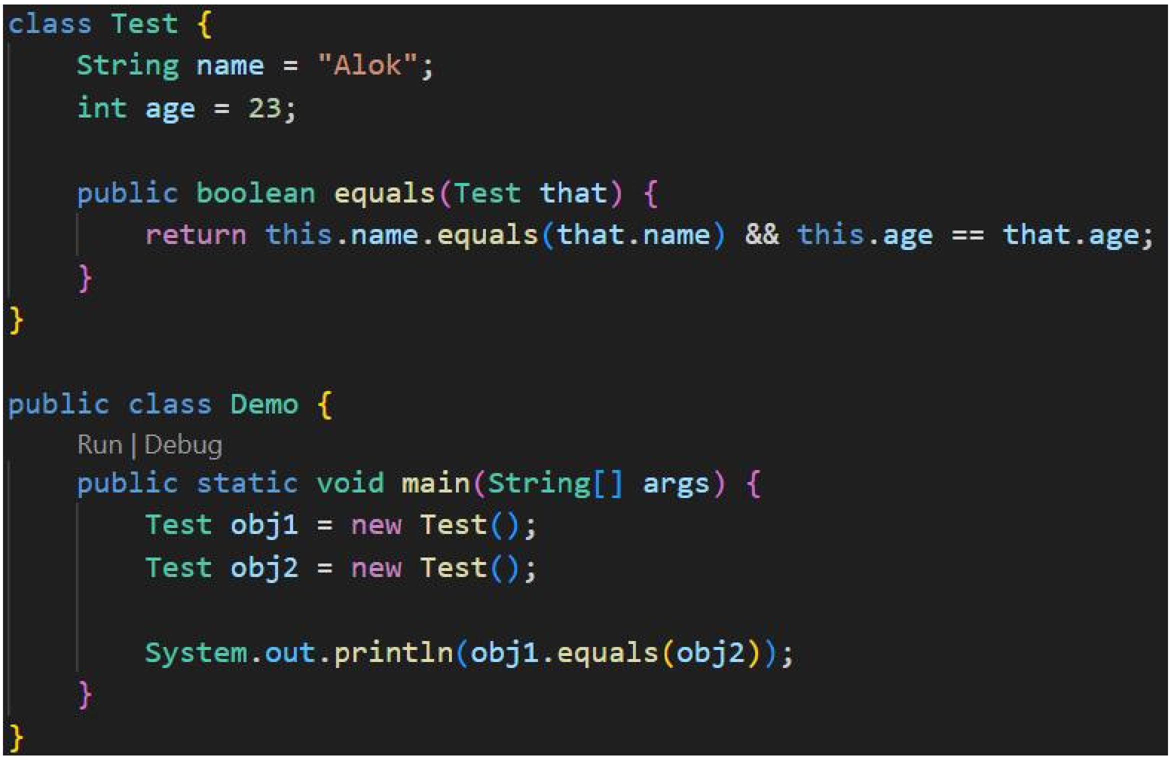


```
alokr@Alok M
$ java Demo
false
```

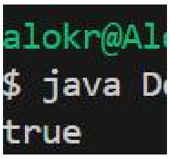
- Now, it is giving **false**; it is because of the method **equals** present inside the class **Object**



```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

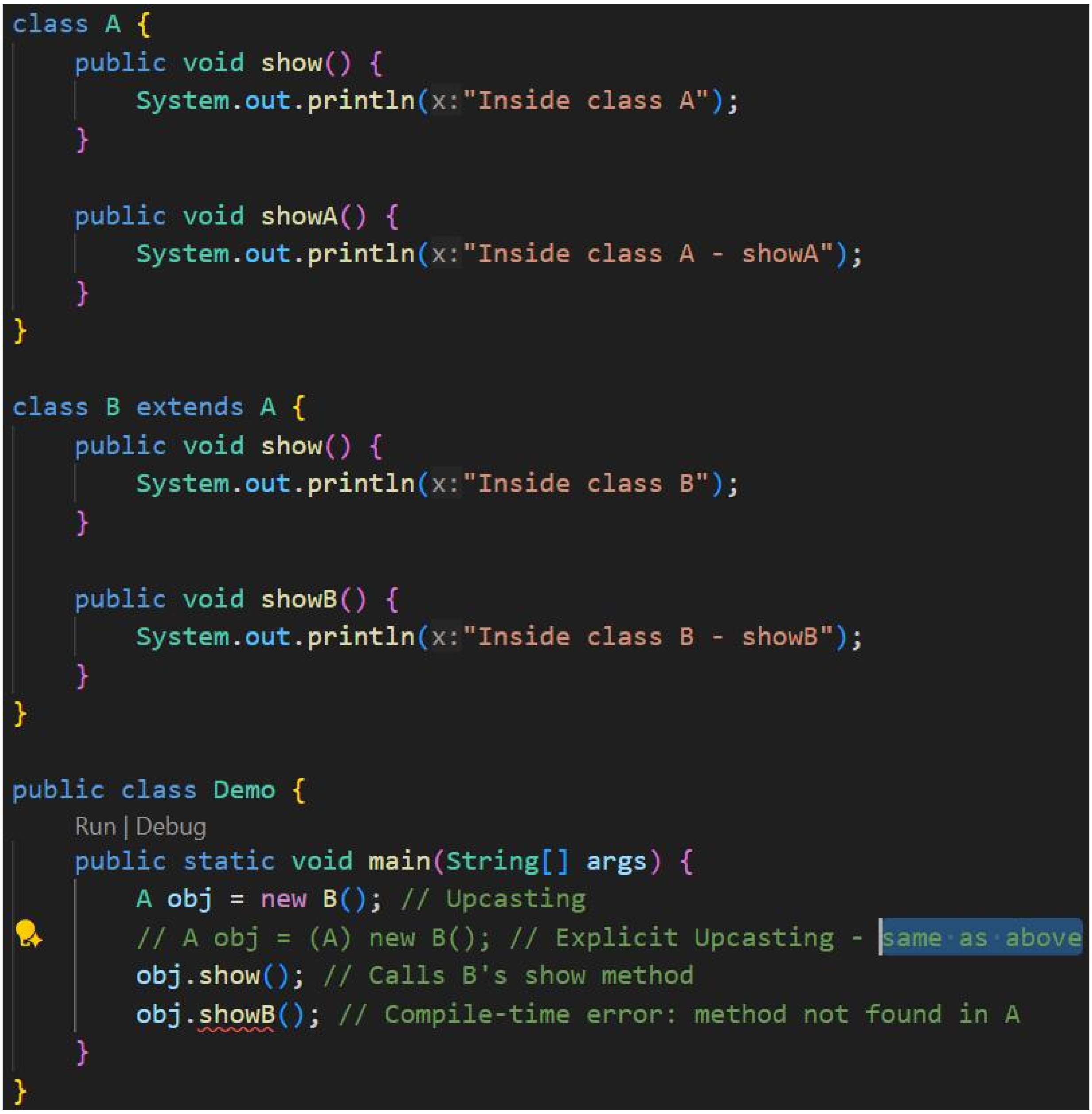


```java
class Test {
    String name = "Alok";
    int age = 23;

    public boolean equals(Test that) {
        return this.name.equals(that.name) && this.age == that.age;
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        Test obj1 = new Test();
        Test obj2 = new Test();

        System.out.println(obj1.equals(obj2));
    }
}
```



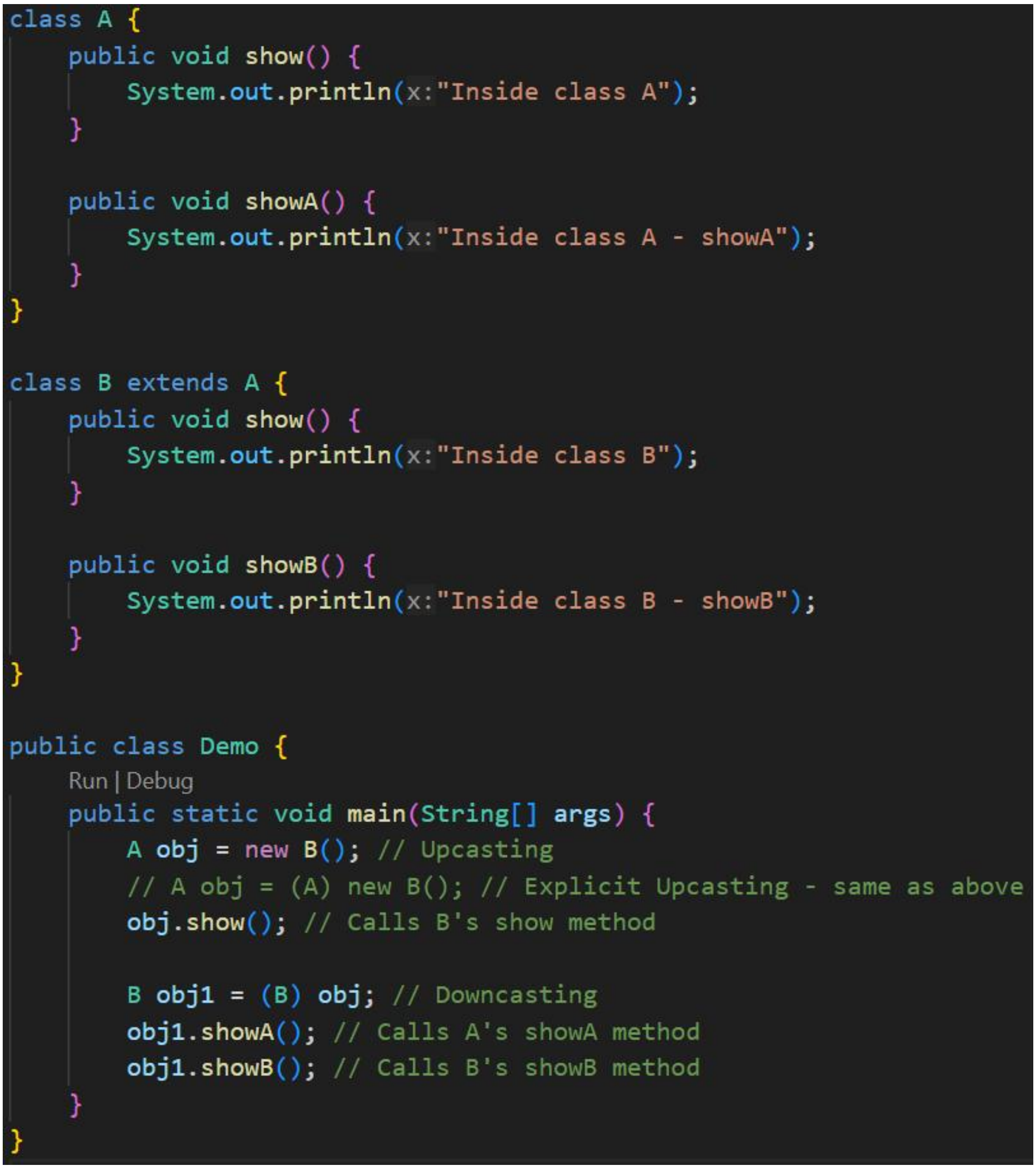
```
alokr@Al
$ java D
true
```

- **Upcasting and Downcasting**



```
class A {
    public void show() {
        System.out.println(x:"Inside class A");
    }

    public void showA() {
        System.out.println(x:"Inside class A - showA");
    }
}

class B extends A {
    public void show() {
        System.out.println(x:"Inside class B");
    }

    public void showB() {
        System.out.println(x:"Inside class B - showB");
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A obj = new B(); // Upcasting
        // A obj = (A) new B(); // Explicit Upcasting - same as above
        obj.show(); // Calls B's show method
        obj.showB(); // Compile-time error: method not found in A
    }
}
```

- 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.

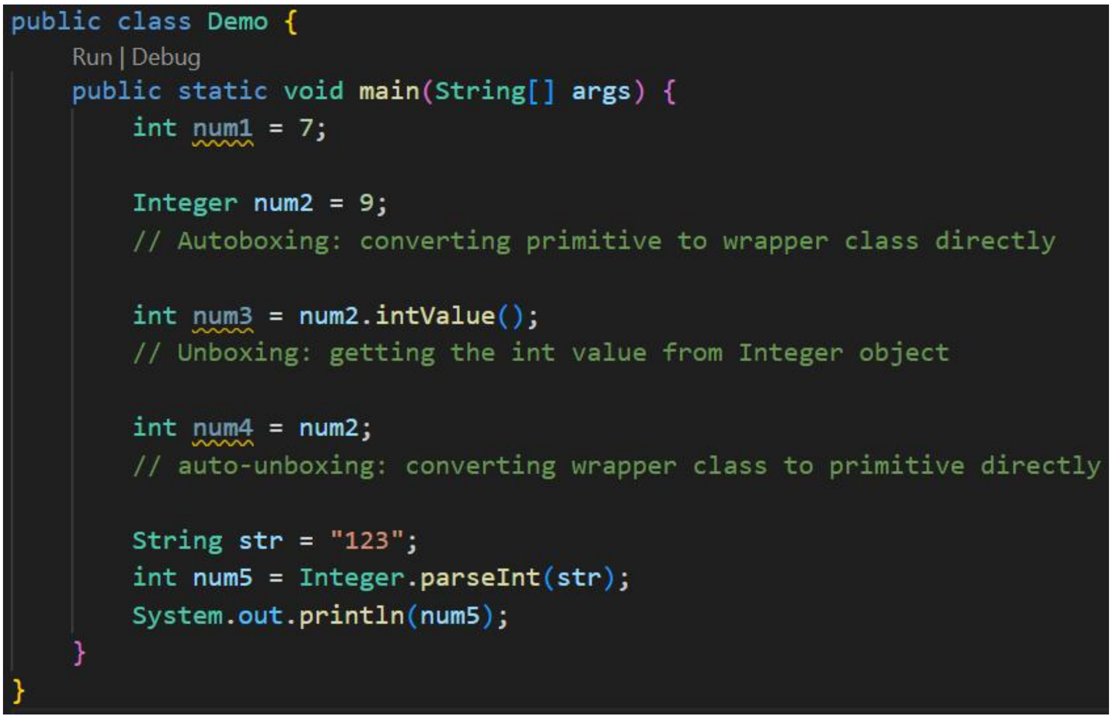


```
class A {
    public void show() {
        System.out.println(x:"Inside class A");
    }

    public void showA() {
        System.out.println(x:"Inside class A - showA");
    }
}

class B extends A {
    public void show() {
        System.out.println(x:"Inside class B");
    }

    public void showB() {
        System.out.println(x:"Inside class B - showB");
    }
}

public class Demo {
    Run | Debug
    public static void main(String[] args) {
        A obj = new B(); // Upcasting
        // A obj = (A) new B(); // Explicit Upcasting - same as above
        obj.show(); // Calls B's show method

        B obj1 = (B) obj; // Downcasting
        obj1.showA(); // Calls A's showA method
        obj1.showB(); // Calls B's showB method
    }
}
```

- 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



```
public class Demo {
    public static void main(String[] args) {
        int num1 = 7;

        Integer num2 = 9;
        // Autoboxing: converting primitive to wrapper class directly

        int num3 = num2.intValue();
        // Unboxing: getting the int value from Integer object

        int num4 = num2;
        // auto-unboxing: converting wrapper class to primitive directly

        String str = "123";
        int num5 = Integer.parseInt(str);
        System.out.println(num5);
    }
}
```