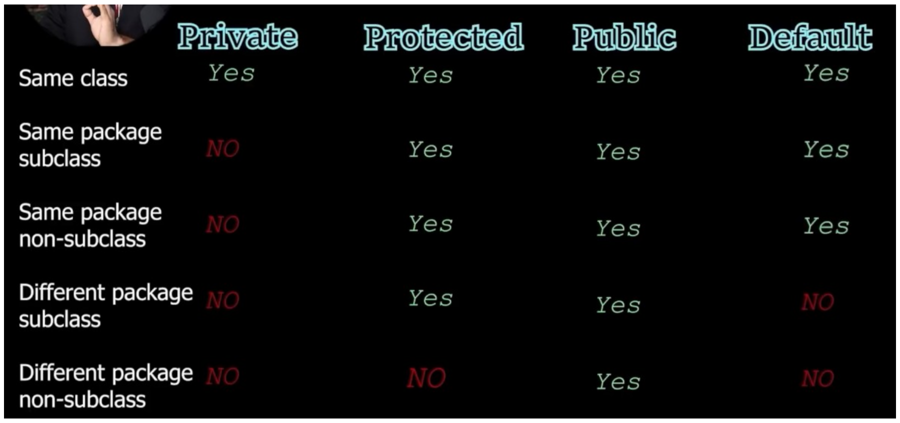
JAVA(Write once and run anywhere)

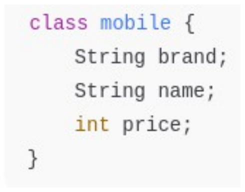
Emon_SUST 01714076452
**** U need basic of programming to read this. I came from C++. so didn't mentioned basics.*
U need to install JDK(Java dev kit). //sudo dnf install java-21-openjdk-devel

//read this section later
**** Every method has its own stack.*
**** Writing multiple code called **Redundancy**.*
**** Each .java file can have only one public class, and its name must match the filename.*
**** Access modifier: Public, Private, protected, default.*
If access modifier is not defined, then it always default modifier.

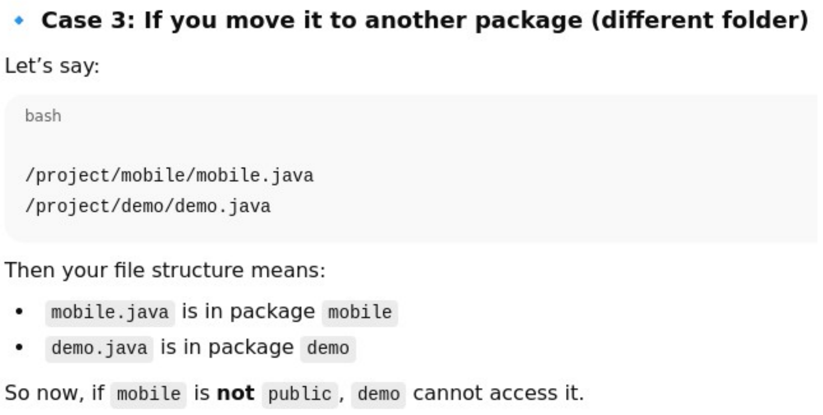


	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package non-subclass	NO	NO	Yes	NO

****You use public when the class should be accessible from anywhere, across all packages/folder.*
don't use public means this class is only visible inside the same package/folder.



```
class mobile {
    String brand;
    String name;
    int price;
}
```



Case 3: If you move it to another package (different folder)

Let's say:

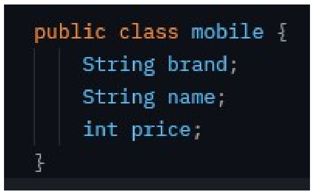
```bash
/project/mobile/mobile.java
/project/demo/demo.java
```

Then your file structure means:

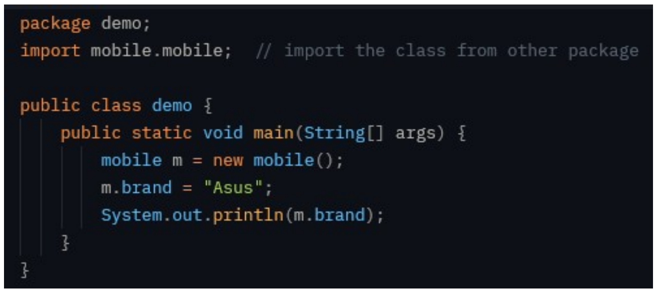
- `mobile.java` is in package `mobile`
- `demo.java` is in package `demo`

So now, if `mobile` is **not** `public`, `demo` cannot access it.

So we need to set "public" to access it from another package/folder. Then import the class to demo(main class file) file. Below:



```
public class mobile {
    String brand;
    String name;
    int price;
}
```



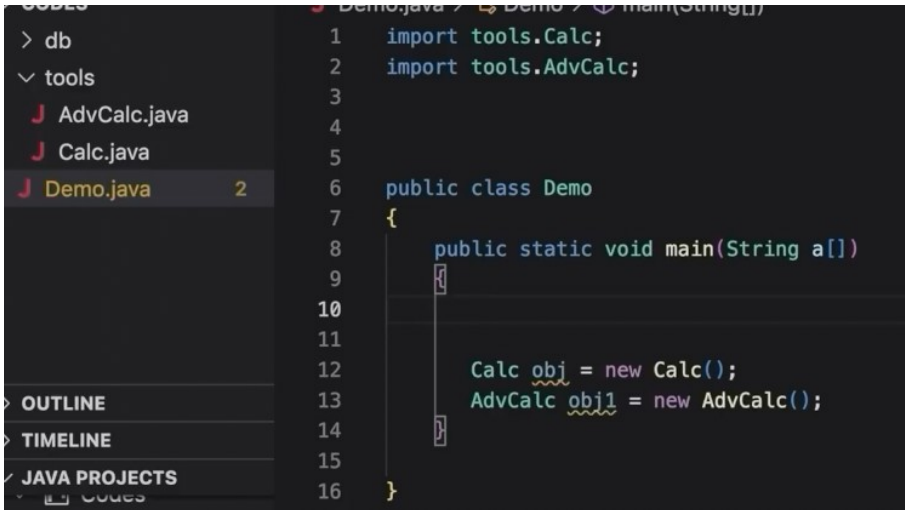
```
package demo;
import mobile.mobile;  // import the class from other package

public class demo {
    public static void main(String[] args) {
        mobile m = new mobile();
        m.brand = "Asus";
        System.out.println(m.brand);
    }
}
```

if mobile.java and demo.java locate in above mentioned directory, this is how import.
Line 2 means, import the Mobile class from mobile package.folder.(above)
“package demo” means // 👈 tells Java that this file belongs to the 'mobile' folder/package.

Q: *Why “import and package” required?*
Ans: *Because Java projects can become huge, with hundreds or thousands of classes —*
so package works like a folder name or namespace, telling the compiler where to find classes.
Without it, everything would live in one "global" space — messy and hard to maintain.

- `package` **= folder path**
- `import` **= shortcut to access another folder’s file**

If we not mention “import and package”, java application will work fine if we put all .java file into same package. But it will hard to manage.
So if a public class is on annother package, we have to import that class inside demo.



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



public class Demo
{
    public static void main(String a[])
    {


        Calc obj = new Calc();
        AdvCalc obj1 = new AdvCalc();
    }

}
```

Calc, Advcalc are on tools package. So need to import to use. Add multiple file on same package we can write: import.tools.; // this will import all file of that package. If there a folder. (tools/calfile)“ * ” will not bring all file from that folder. Need to specify that folder by import.*
If a variable is on other package. It have to be public to use from other packages. Same as class.

****2 public classes can’t be in same file.*
*** don’t use variable as default. Use other modifier.*

--------------x--------

How java works: *Java is platform independent. But that system has to have JVM(Java Virtual machine). Java file can’t run without JVM. JVM work onto OS. JVM don’t accept Java code. It only execute byte code Compiler(javac) will compile all java code and create a byte code file, which will execute by JVM. Start execute with “Main” method.*
To run java applications we need library’s in runtime. These library’s are in JRE(Java runtime environment). And JVM is a part of JRE. So JVM can use library’s from JRE to run the code.
For development purpose, we need JDK. JDK will have JRE and JRE will have JVM. JDK don’t need where we will run our application(client PC).

Component	Purpose
JDK (Java Development Kit)	Used by **developers** to *write, compile, and build* Java programs. Includes `javac`, debugger, etc.
JRE (Java Runtime Environment)	Used by **users (clients)** to *run* Java programs. Includes `java`, JVM, and core libraries.

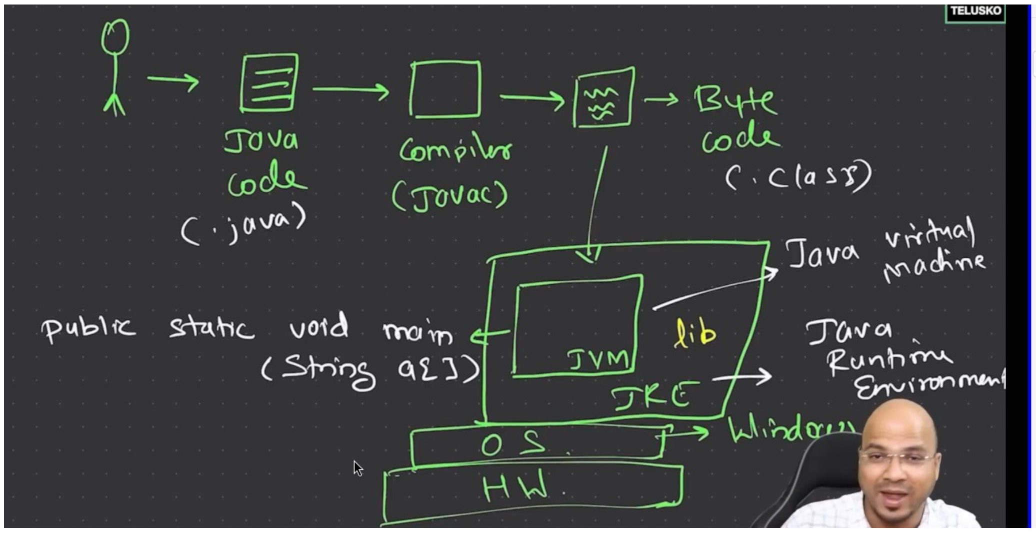
🧠 2️⃣ Why the client PC doesn't need JDK

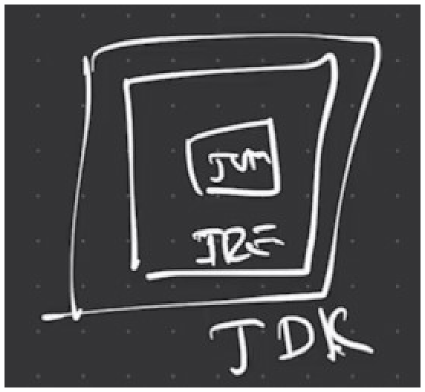
Because:

> Clients **don't write or compile** Java code — they only **run** the compiled `.class` or `.jar` files.

When you build an app:

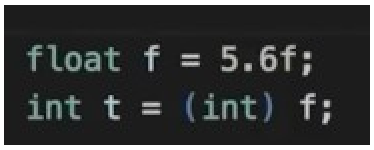
1. You write code (`.java`)
2. Compile it with `javac` → produces `.class` files (bytecode)
3. Package and ship `.jar` or `.war` file
4. On client PC, the **JRE** (or JVM) runs that bytecode — no compiler needed.





------------x---------

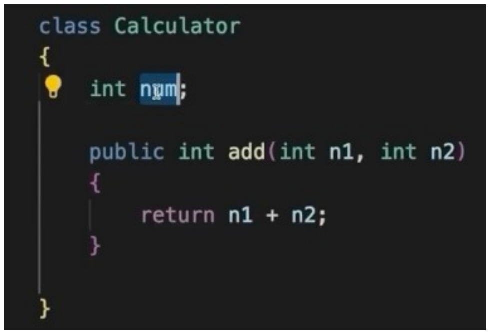
Type Casting:



```java
float f = 5.6f;
int t = (int) f;
```

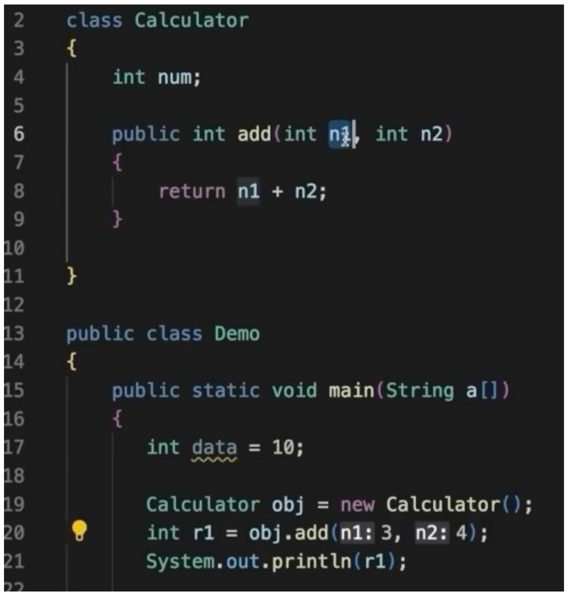
---------x---------

For Every method there will a own stack inside JVM memory. Another memory is Heap.



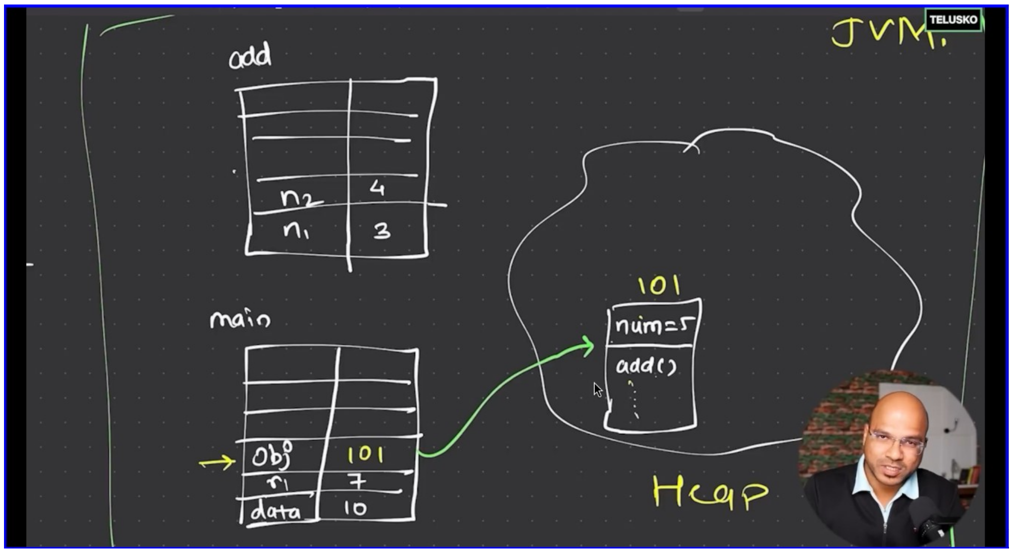
```java
class Calculator
{
    int num;

    public int add(int n1, int n2)
    {
        return n1 + n2;
    }

}
```

Here "Num" variable is instance variable, and "n1,n2" is local variable.(Below)



```java
2   class Calculator
3   {
4       int num;
5
6       public int add(int n1, int n2)
7       {
8           return n1 + n2;
9       }
10
11  }
12
13  public class Demo
14  {
15      public static void main(String a[])
16      {
17          int data = 10;
18
19          Calculator obj = new Calculator();
20          int r1 = obj.add(n1: 3, n2: 4);
21          System.out.println(r1);
```

There Will be a stack on for main method. Also for add(Lint 6;) method. On line 19; obj is not an object. Its a reference variable of the object. That object memory will be on heap memory. 'obj' will store memory location. On that object memory there are 2 part. 1 aprt will store all instance variables and other store method definition. But that have methods definition only. There will be a stack for that(add) method. And methods local variable will store on that stack.



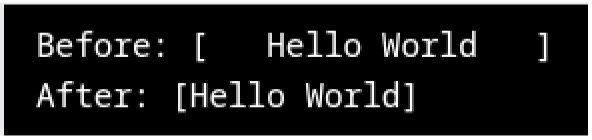
---------x--------

Methods:

Math.random(); *// return double, return range 0>=x<=1, Here "Math" is library*

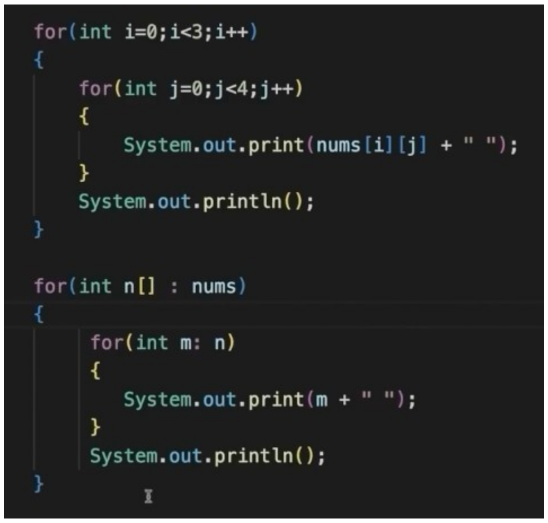
objName.capacity(); *// will give size. For string it will give extra 16 char. If string a="Emon" //4 char. Capacity will be 16+4=20 char. Extra 16 char space will give. This is for* ***stringBuffer****(Mutable string) see: few lecture ahead. Give extra 16 char, cause if no continuous memory available it have to reallocate memory. To reduce reallocation. "Length" and "capacity" diff method.*

Trim(); *// remove space from beginning and end of string.*



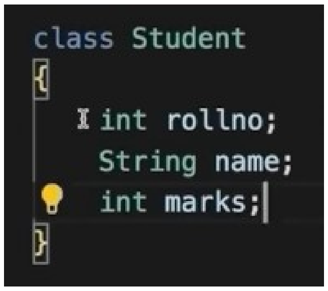
```
Before: [   Hello World   ]
After: [Hello World]
```

---------x---------

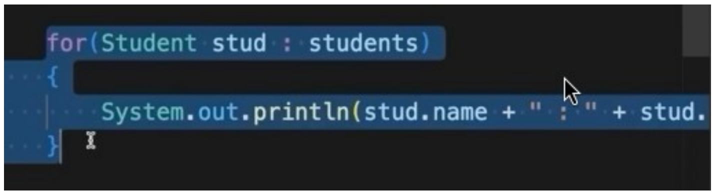


```
for(int i=0;i<3;i++)
{
    for(int j=0;j<4;j++)
    {
        System.out.print(nums[i][j] + " ");
    }
    System.out.println();
}

for(int n[] : nums)
{
     for(int m: n)
     {
        System.out.print(m + " ");
     }
     System.out.println();
}
```

Both are same.(Above) Second is useful when we don't know the array size. Is called "for each loop". This used only for array of any type. Like array of object. (Below)

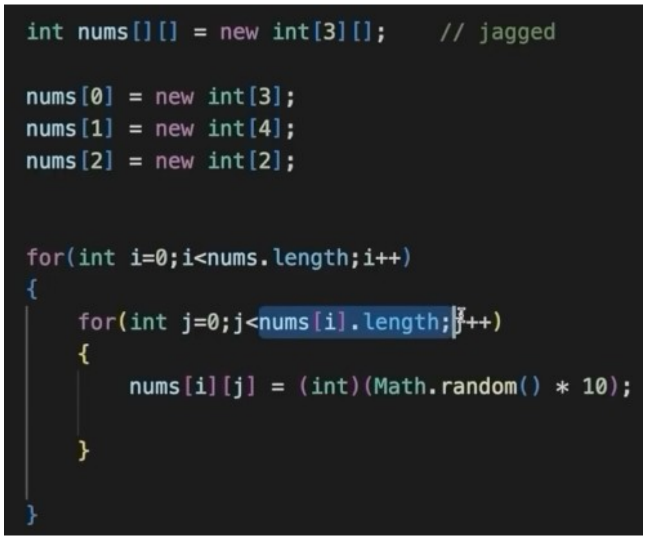


```
class Student
{
    int rollno;
    String name;
    int marks;
}
```



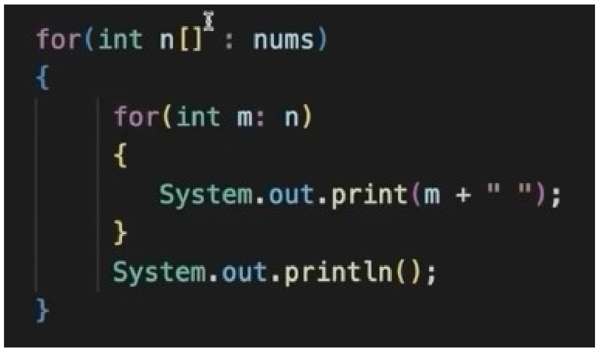
```
for(Student stud : students)
{
    System.out.println(stud.name + " : " + stud.
}
```

What if we know the row number. But the column size may vary of diff rows for 2D array. This called Jagged Array. (Below)



```
int nums[][] = new int[3][];    // jagged

nums[0] = new int[3];
nums[1] = new int[4];
nums[2] = new int[2];


for(int i=0;i<nums.length;i++)
{
    for(int j=0;j<nums[i].length;j++)
    {
        nums[i][j] = (int)(Math.random() * 10);

    }

}
```

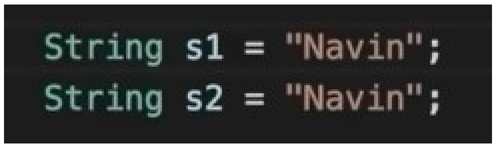
to Print:



```
for(int n[] : nums)
{
    for(int m: n)
    {
        System.out.print(m + " ");
    }
    System.out.println();
}
```

In Java everything is an object. Like here array is an object.
By default array element value is 0.
** Exceptions are runtime error.*

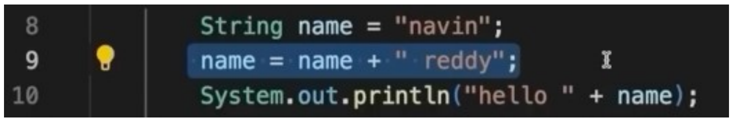
-------------x------------



```
String s1 = "Navin";
String s2 = "Navin";
```

In heap memory there is a section called “String const pool”, where all string stored. Here only 1 object will create. For creating “s2” object, Java will first search whether is present previously or not. Both “s1,s2” variable will contain same reference address(on stack) stored in heap.

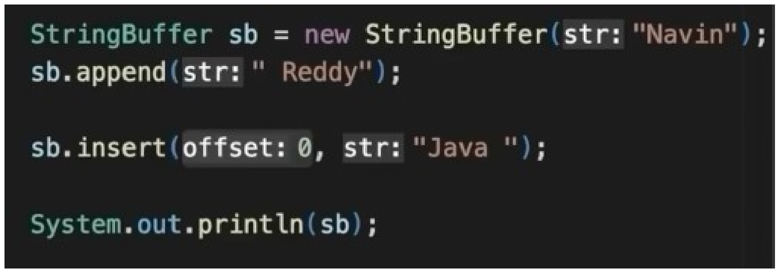
Strings are const.



```
8     String name = "navin";
9     name = name + " reddy";
10    System.out.println("hello " + name);
```

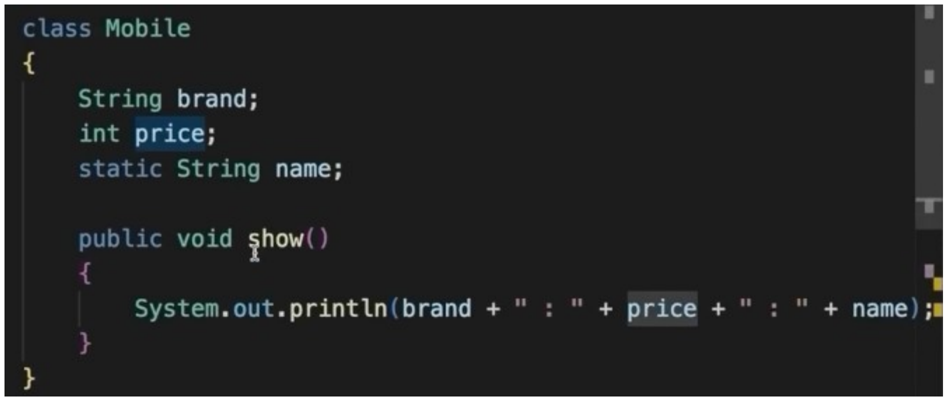
Here, on line 9; we are changing string. But they are const. Suppose name address is 100. when we are adding “reddy” it creates a new object(heap) after concatenating name. Replace name address(on stack) to that new object like 101. Here data is not changed. The “navin” object is eligible for garbage collection.

Strings can be changeable(Mutable). Using StringBuffer, StringBuilder



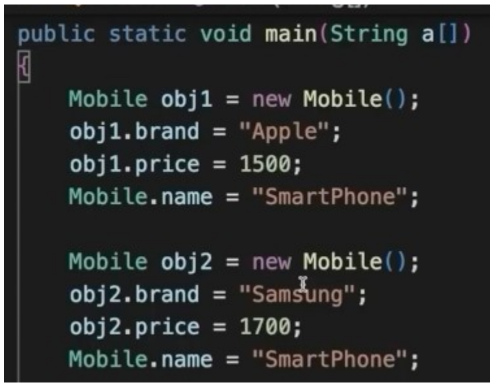
```
StringBuffer sb = new StringBuffer(str: "Navin");
sb.append(str: " Reddy");

sb.insert(offset: 0, str: "Java ");

System.out.println(sb);
```

StringBuffer is thread safe, StringBuilder is not. *We will learn later. Methods are same for both.*
**** All memory used by code is from JVM. Stack, heap are inside JVM.*
Static: **we can’t make a class static unless the class is inner class.(see inner class)*

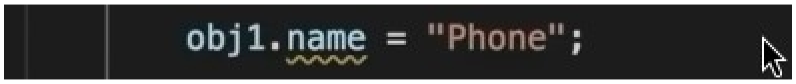


```
class Mobile
{
    String brand;
    int price;
    static String name;

    public void show()
    {
        System.out.println(brand + " : " + price + " : " + name);
    }
}
```

If we want that for all object 1 property will be same. So we can use static”. Static variable should called with class name, but it can be called with object name, but not recommended. If one object property of static variable is changed it effects all. It is class member, not object member. In Java, static means belonging to the class itself, not to any specific object (instance).

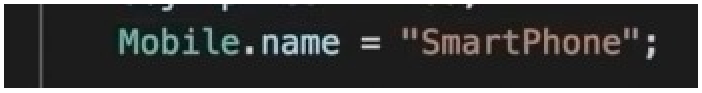


```
public static void main(String a[])
{
    Mobile obj1 = new Mobile();
    obj1.brand = "Apple";
    obj1.price = 1500;
    Mobile.name = "SmartPhone";

    Mobile obj2 = new Mobile();
    obj2.brand = "Samsung";
    obj2.price = 1700;
    Mobile.name = "SmartPhone";
```



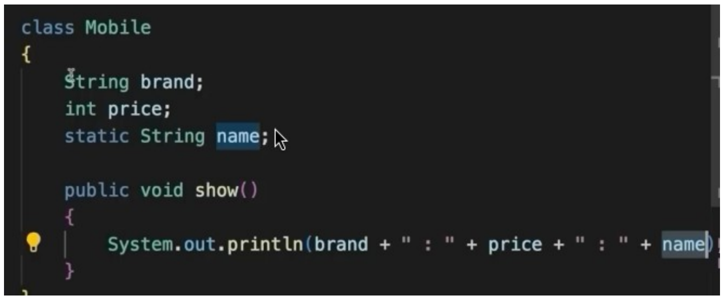
```java
obj1.name = "Phone";
```

Setting one obj name(inside main) changes all obj name properties. While using static



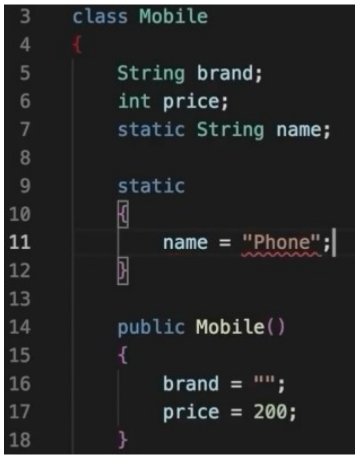
```java
Mobile.name = "SmartPhone";
```

Static variable should be called with class name(above). Static variable shared by diff object. We can us static variable on non-static variable. (below)



```java
class Mobile
{
    String brand;
    int price;
    static String name;

    public void show()
    {
        System.out.println(brand + " : " + price + " : " + name)
    }
```

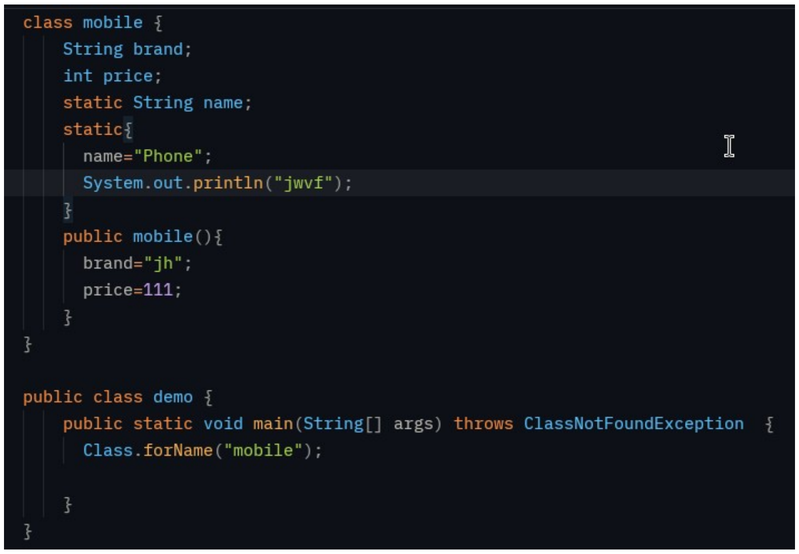
Static Block: *Static block(line 9;) only calls once. No matter how many object created. But constructor will be called every time obj created. Below:*



```java
3   class Mobile
4   {
5       String brand;
6       int price;
7       static String name;
8
9       static
10      {
11          name = "Phone";
12      }
13
14      public Mobile()
15      {
16          brand = "";
17          price = 200;
18      }
```

Which variable we don't want to change we can set inside static block.
We can set default value on constructor(Line 14;) (above).
Always static block will call firth then constructor.
An obj is called there are 2 steps. Class loads & object instantiated. Class is only load once(while1st obj created).
Since class loads once, static block loads only once. So if we don't create an obj, no class loaded and not static block will be execute.

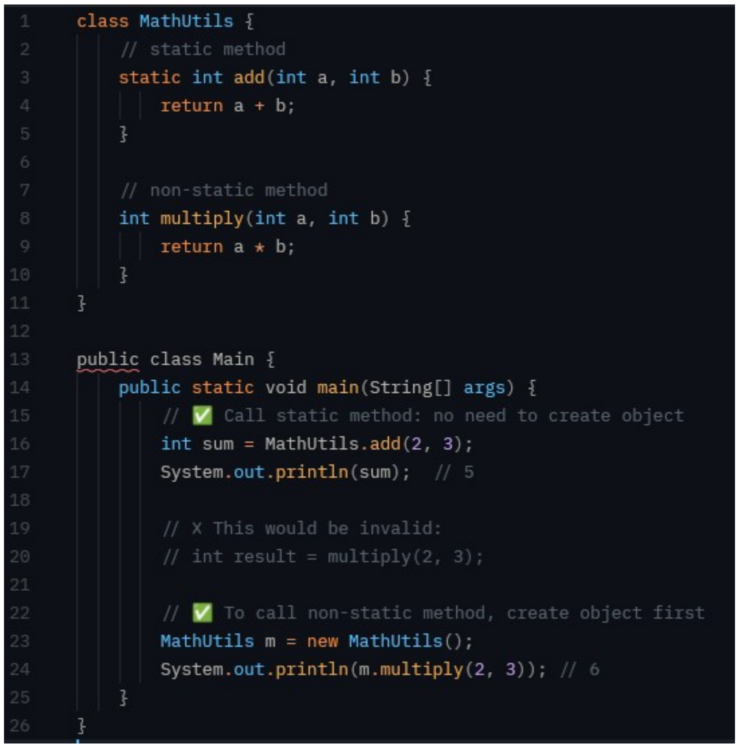
Without loading an obj we can load class. Below:



```java
class mobile {
    String brand;
    int price;
    static String name;
    static{
      name="Phone";
      System.out.println("jwvf");
    }
    public mobile(){
      brand="jh";
      price=111;
    }
}

public class demo {
    public static void main(String[] args) throws ClassNotFoundException  {
      Class.forName("mobile");

    }
}
```

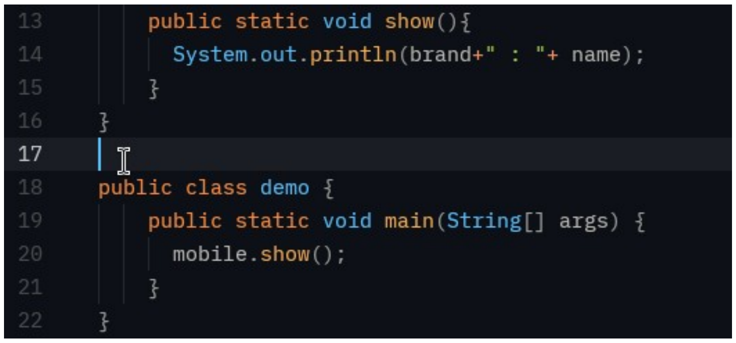
Here there is an exception. We will learn later.

Static Method: *No need to create object to call.*



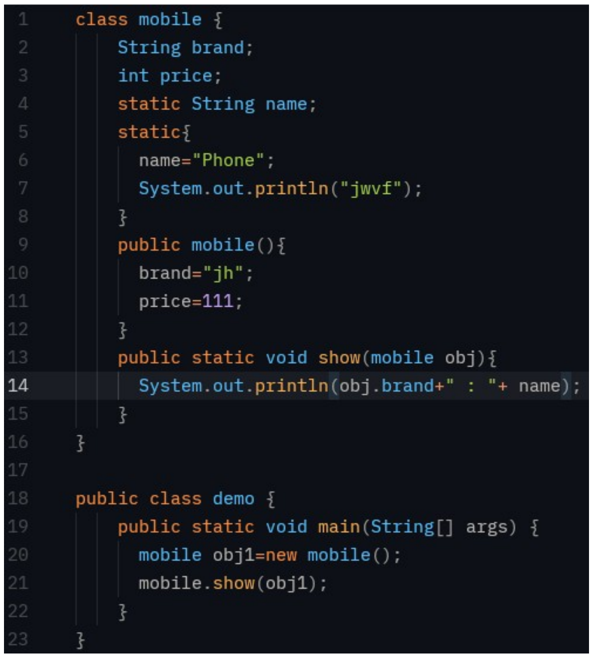
```java
class MathUtils {
    // static method
    static int add(int a, int b) {
        return a + b;
    }

    // non-static method
    int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // ✅ Call static method: no need to create object
        int sum = MathUtils.add(2, 3);
        System.out.println(sum);  // 5

        // X This would be invalid:
        // int result = multiply(2, 3);

        // ✅ To call non-static method, create object first
        MathUtils m = new MathUtils();
        System.out.println(m.multiply(2, 3)); // 6
    }
}
```

Non-static variable can't use in static variable. Below: line 14;



```java
13      public static void show(){
14        System.out.println(brand+" : "+ name);
15      }
16    }
17
18    public class demo {
19      public static void main(String[] args) {
20        mobile.show();
21      }
22    }
```

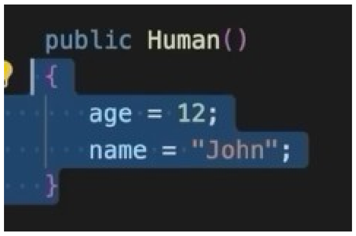
So if we want to use non-static variable in static variable we can pass object as an argument.



```java
class mobile {
    String brand;
    int price;
    static String name;
    static{
      name="Phone";
      System.out.println("jwvf");
    }
    public mobile(){
      brand="jh";
      price=111;
    }
    public static void show(mobile obj){
      System.out.println(obj.brand+" : "+ name);
    }
}

public class demo {
    public static void main(String[] args) {
      mobile obj1=new mobile();
      mobile.show(obj1);
    }
}
```

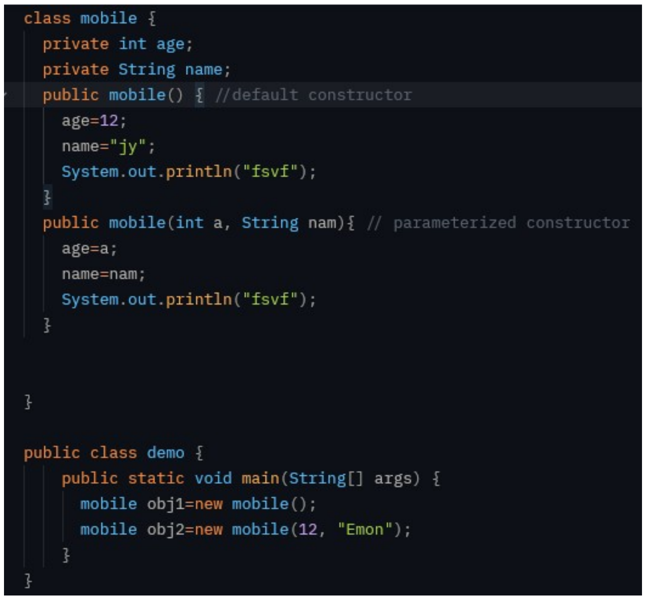
****Q: So why do main method "static" keyword?*
Ans: if main method is non-static, to call main we need to create a an object of class to call main. So main is not executed how can we create an object of demo. That's why static.

Constructor: *Is a special kind of method which is being called during obj is created. Method with class name. No return value, not even "void". We can use to set default value.*



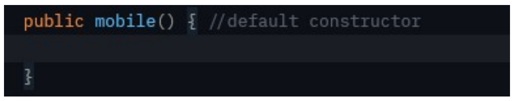
```java
public Human()
{
    age = 12;
    name = "John";
}
```

We can create another constructor which will accept some value. Called Parameterized constructor. (above is default constructor).



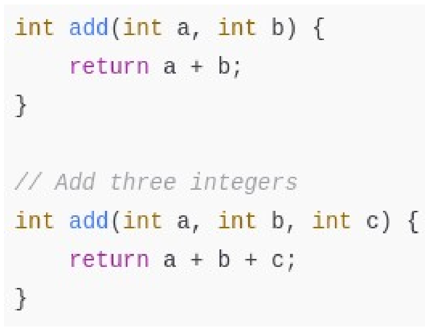
```java
class mobile {
  private int age;
  private String name;
  public mobile() { //default constructor
    age=12;
    name="jy";
    System.out.println("fsvf");
  }
  public mobile(int a, String nam){ // parameterized constructor
    age=a;
    name=nam;
    System.out.println("fsvf");
  }


}

public class demo {
    public static void main(String[] args) {
      mobile obj1=new mobile();
      mobile obj2=new mobile(12, "Emon");
    }
}
```

If no constructor mentioned implicitly java will create a blank default constructor.
Use: set connection bet^n application and database.



```java
public mobile() { //default constructor

}
```

Method overloading: *Method can be same name but parameter type/size/order must be diff.*



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

// Add three integers
int add(int a, int b, int c) {
    return a + b + c;
}
```

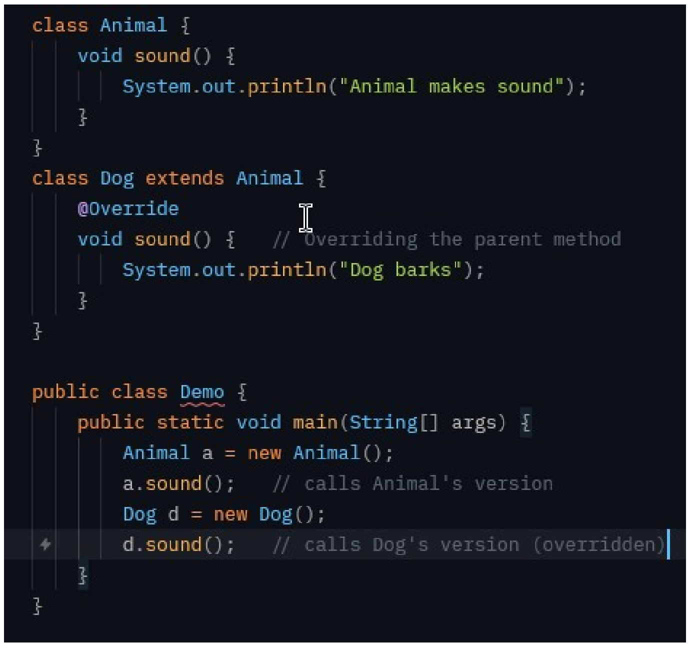
-----------x----------

Method Overriding *(Need inheritance concept): Overriding means redefining a method from the parent class in the child class with the same signature (name, parameters, and return type).*

Key points:

1. *Happens between superclass and subclass.(Inheritance)*
2. *Must have exact same method signature.*
3. *Return type must be same (or covariant).*
4. *Happens at runtime → called runtime polymorphism.*
5. *The overridden method in the child replaces the one in the parent when called via a child object.*

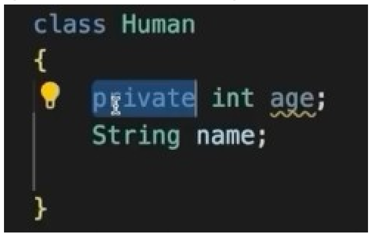
***Static method can't override. Cause static method is bound to class. Methods are bound to object.*



```java
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {    // Overriding the parent method
        System.out.println("Dog barks");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.sound();    // calls Animal's version
        Dog d = new Dog();
        d.sound();    // calls Dog's version (overridden)
    }
}
```

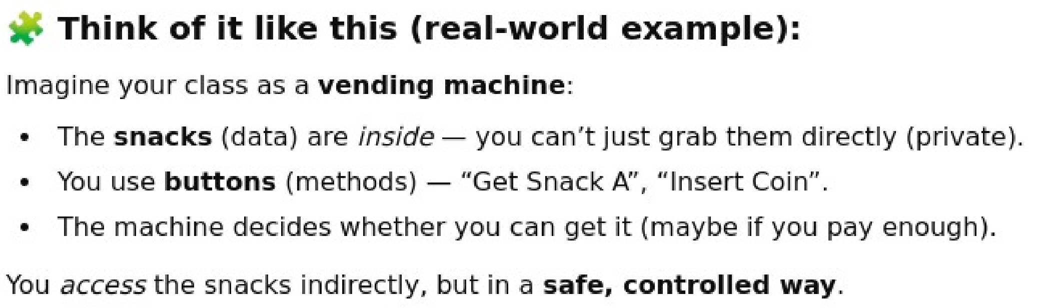
Same name same parametered methods on all parent and child class. Object with parent class will call parent "sound" method and object with child will call child "sound" method.

Encapsulation: *Encapsulation(data hiding) means somethings we don't want someone to use it from outside.*



```
class Human
{
    private int age;
    String name;

}
```

Using "Private" this variable is accessible inside class. Instance variable should be private. This type of date can be accessible with method inside that class.

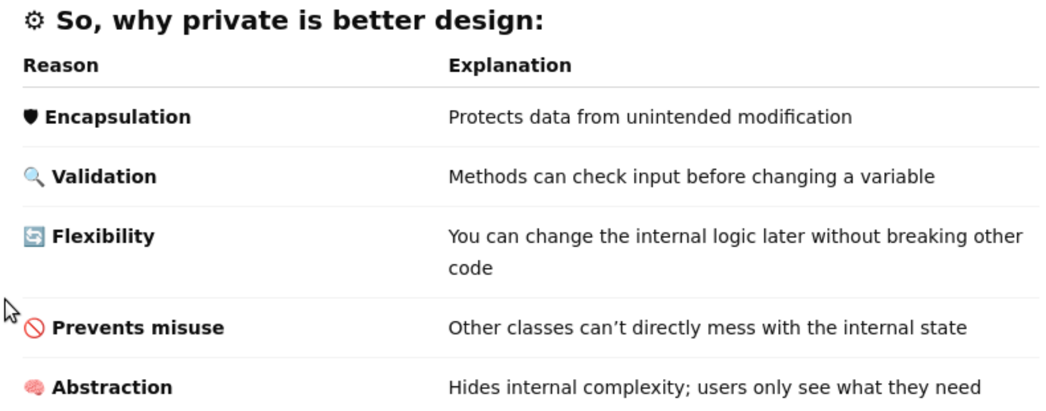


🧩 Think of it like this (real-world example):

Imagine your class as a **vending machine**:

- The **snacks** (data) are *inside* — you can't just grab them directly (private).
- You use **buttons** (methods) — "Get Snack A", "Insert Coin".
- The machine decides whether you can get it (maybe if you pay enough).

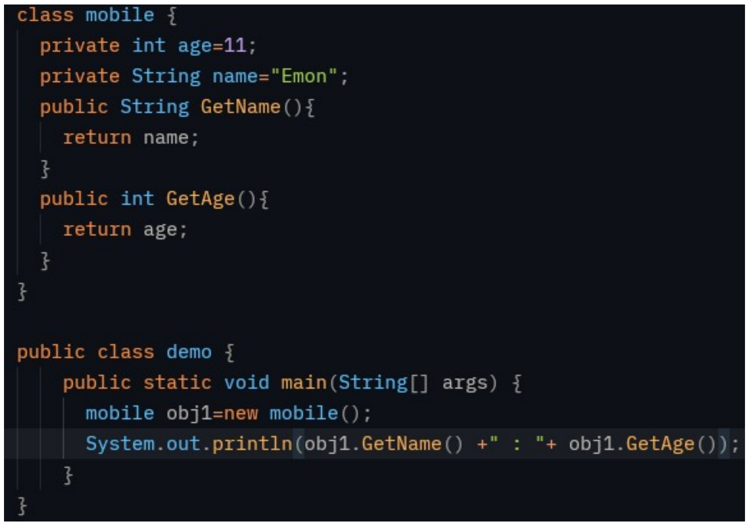
You *access* the snacks indirectly, but in a **safe, controlled way**.



⚙ So, why private is better design:

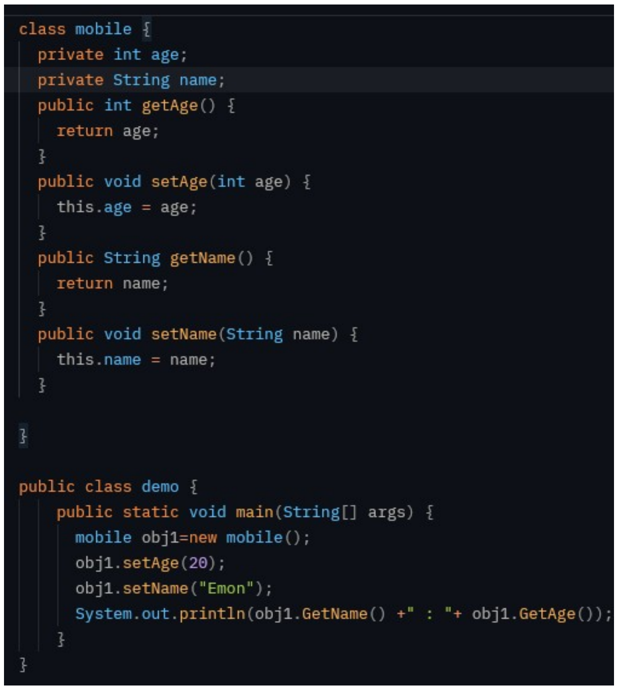
Reason	Explanation
🛡 **Encapsulation**	Protects data from unintended modification
🔍 **Validation**	Methods can check input before changing a variable
🔄 **Flexibility**	You can change the internal logic later without breaking other code
🚫 **Prevents misuse**	Other classes can't directly mess with the internal state
🧠 **Abstraction**	Hides internal complexity; users only see what they need

To control access we can use private variable and access with methods.



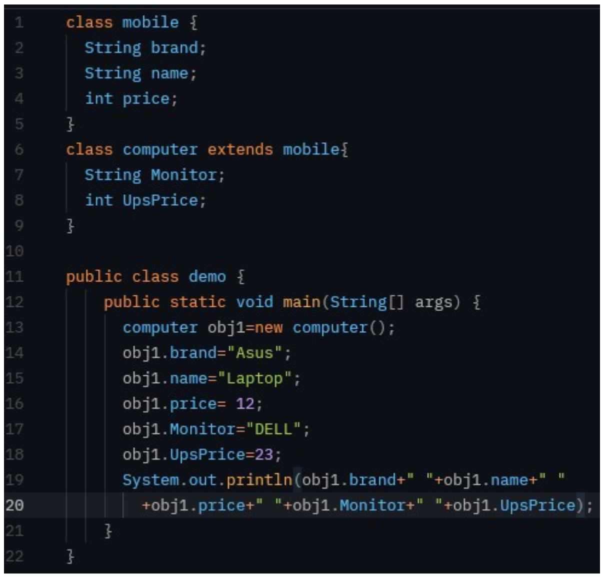
```
class mobile {
  private int age=11;
  private String name="Emon";
  public String GetName(){
    return name;
  }
  public int GetAge(){
    return age;
  }
}

public class demo {
    public static void main(String[] args) {
      mobile obj1=new mobile();
      System.out.println(obj1.GetName() +" : "+ obj1.GetAge());
    }
}
```

To getting these data creating a method called "getter".
To set the data of a private variable by a method called "Setter". All same just create a method which accept data as argument and assign it to the variable. "This" is a keyword represents current object.



```java
class mobile {
  private int age;
  private String name;
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }

}

public class demo {
    public static void main(String[] args) {
      mobile obj1=new mobile();
      obj1.setAge(20);
      obj1.setName("Emon");
      System.out.println(obj1.GetName() +" : "+ obj1.GetAge());
    }
}
```

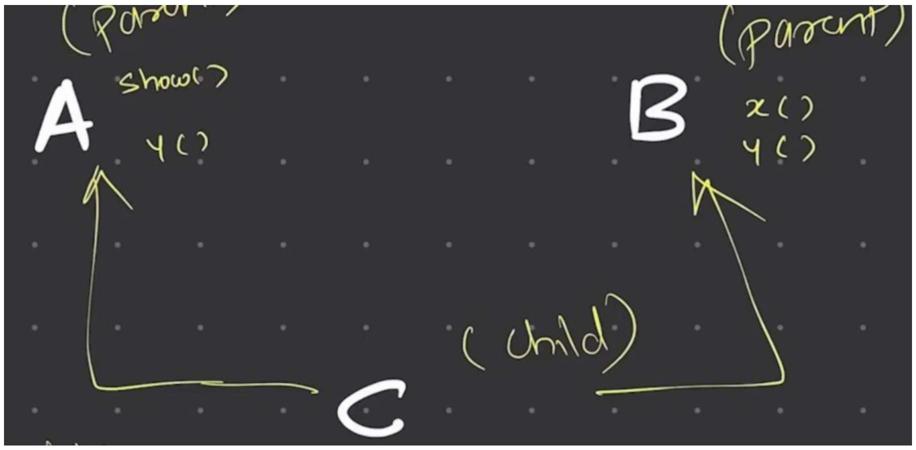
Inheritance: *Suppose u make a calculator. It has features like addition, substraction etc. all in a class(A). Now want to add another feature like sin(), sqrt() etc. these are in another class(B). so if we want all feature from class(A) to class(B). this is called Inheritance. Class B inherite class A. A is parent/super class, B is child/sub class.*



```java
class mobile {
  String brand;
  String name;
  int price;
}
class computer extends mobile{
  String Monitor;
  int UpsPrice;
}

public class demo {
    public static void main(String[] args) {
      computer obj1=new computer();
      obj1.brand="Asus";
      obj1.name="Laptop";
      obj1.price= 12;
      obj1.Monitor="DELL";
      obj1.UpsPrice=23;
      System.out.println(obj1.brand+" "+obj1.name+" "
        +obj1.price+" "+obj1.Monitor+" "+obj1.UpsPrice);
    }
}
```

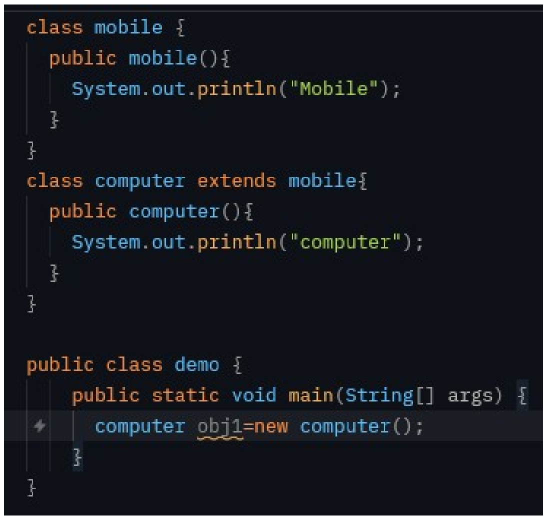
*Here computer class inherite mobile class. So mobile class instance can be accessible from computer class object. Other class can inherite computer class, so that we have all properties of mobile class and computer class in that class. This is called **multi-level inheritance**. Mobile class to computer class is called single level inheritance.*

***If A class is parent of C class and B is also parent of C class. This is called multiple inheritance.*



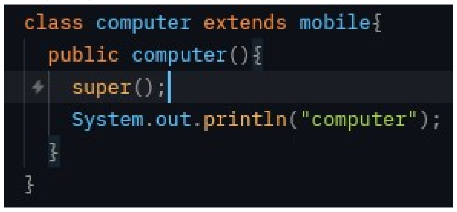
Java doesn't support multiple inheritance cause both A,B may have same method. It will conflict which to take. This called Ambiguity.

Super(): *When we create an object it calls constructors of that class and super class of that class.*



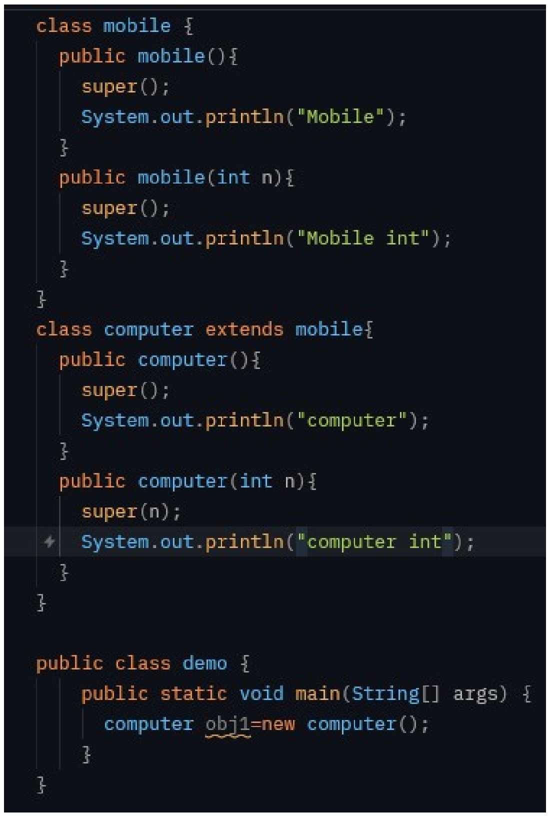
```
class mobile {
  public mobile(){
    System.out.println("Mobile");
  }
}
class computer extends mobile{
  public computer(){
    System.out.println("computer");
  }
}

public class demo {
    public static void main(String[] args) {
      computer obj1=new computer();
    }
}
```

creating computer obj will print "mobile\n computer" How? See below.
Every constructor has a super() method by default. Hidden. We don't have to mention that unless pass a value.



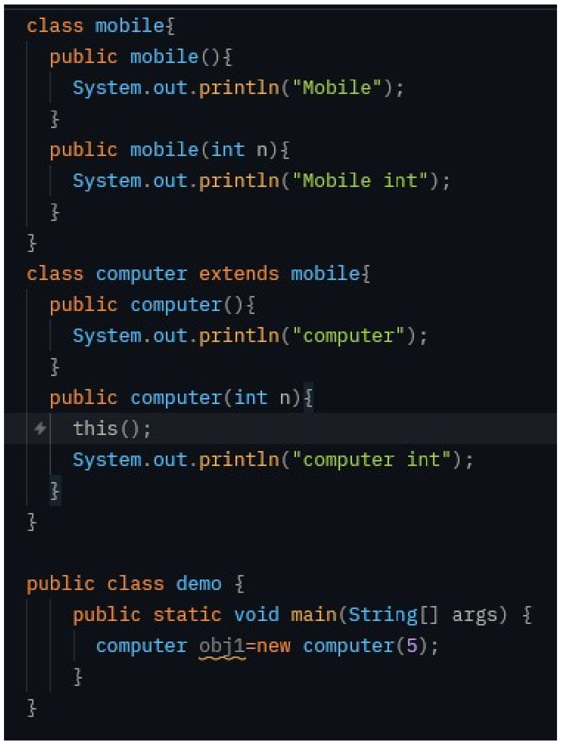
```
class computer extends mobile{
  public computer(){
    super();
    System.out.println("computer");
  }
}
```

When constructor executes. It first see super(). It tells, "Call the constructor of super class(Parent default constructor)"
Here super class of computer class is mobile class. So it call mobile class constructor.
But if want to execute parameterized constructor of super class(mobile) we can pass value according to method through super(3) method.



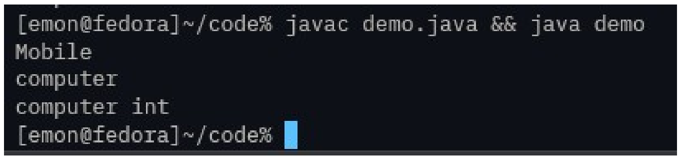
```java
class mobile {
  public mobile(){
    super();
    System.out.println("Mobile");
  }
  public mobile(int n){
    super();
    System.out.println("Mobile int");
  }
}
class computer extends mobile{
  public computer(){
    super();
    System.out.println("computer");
  }
  public computer(int n){
    super(n);
    System.out.println("computer int");
  }
}

public class demo {
    public static void main(String[] args) {
      computer obj1=new computer();
    }
}
```

Every class(if class is not inherite other) inherite object class. So when super class constructor see super() method. It calls its super class(object) constructor. This is below. But this no need to mention "extends object".



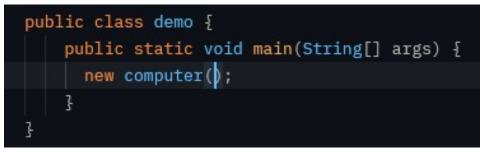
```java
class mobile{
  public mobile(){
    System.out.println("Mobile");
  }
  public mobile(int n){
    System.out.println("Mobile int");
  }
}
class computer extends mobile{
  public computer(){
    System.out.println("computer");
  }
  public computer(int n){
    this();
    System.out.println("computer int");
  }
}

public class demo {
    public static void main(String[] args) {
      computer obj1=new computer(5);
    }
}
```

If we want both constructor of computer will execute we can use this() method. "this()" will execute default constructor of a same class. Output:



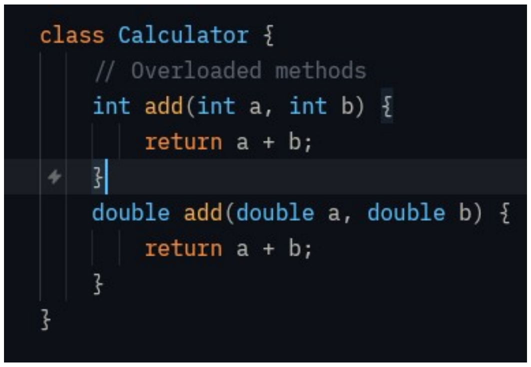
```
[emon@fedora]~/code% javac demo.java && java demo
Mobile
computer
computer int
[emon@fedora]~/code%
```

computer obj; //reference creation. Getting reference.
=new computer(); // creating an object.



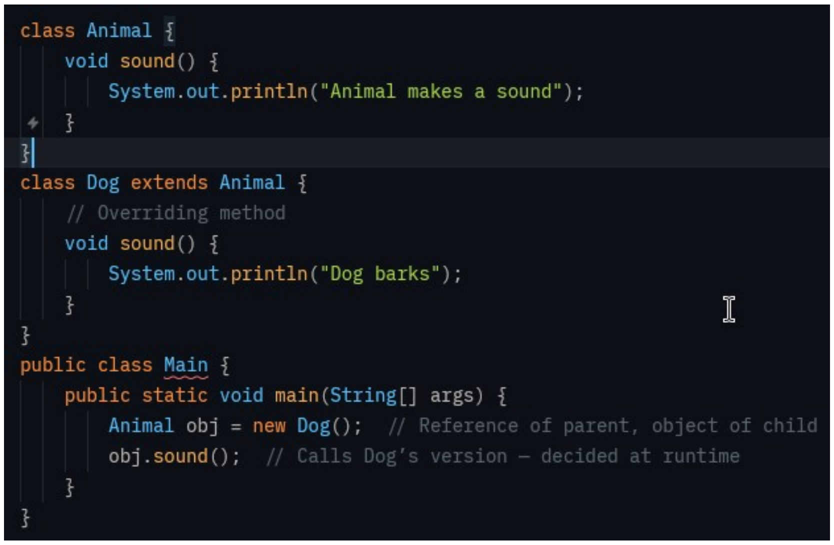
```java
public class demo {
    public static void main(String[] args) {
        new computer();
    }
}
```

Here only object created. No reference. So not a variable containing object reference onto stack. This is call anonymous object. We can't use them more than once. Like "new computer.show". Here suppose show() is a method.
--------x------
Polymorphism: *Means "many forms". It allows the same function or method name to behave* ***differently*** *based on: The* ***object type****, or The* ***parameters passed****.*
2 types:,
Compile time polymorphism: *Ex: Method overloading, operator overloading.*
Decide by the compiler which method to call.



```java
class Calculator {
    // Overloaded methods
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

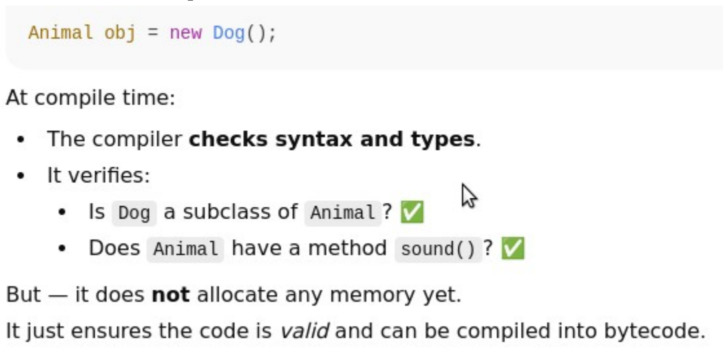
The compiler chooses which add() method to call based on the argument types. This is compile time polymorphism.
Run time polymorphism: *Decided at runtime — the method that gets executed depends on the actual object (not the reference type).*



```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    // Overriding method
    void sound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal obj = new Dog();  // Reference of parent, object of child
        obj.sound();  // Calls Dog's version — decided at runtime
    }
}
```

Here, even though obj is of type Animal, Java checks the actual object type (which is Dog) at runtime and executes the overridden sound() method.

***Q:** Why compiler doesn't decide at compile time, which method have evaluate?*
***Ans:** Compiler only checks declaration type of the reference(Animal). And the actual object will be created at run time. So compiler will check declaration type(Animal) and when it sees "obj.sound();" it search for sound() method called on Animal class or not. If not give compilation error. If yes compilation successful.*
At Runtime: JVM(not compiler) see "obj=new Dog();" it will allocate memory for object on heap.
So when it executes "obj.sound()",
the JVM looks up the real object type (Dog) and calls the overridden sound() method from the Dog class.
*This process is called **Dynamic method dispatch**.*

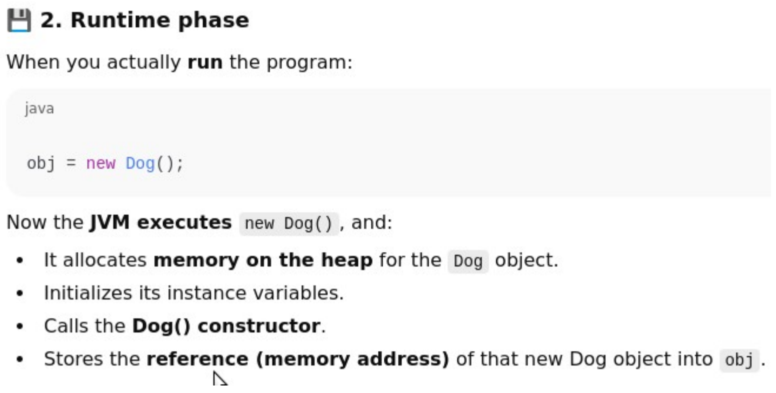


```java
Animal obj = new Dog();
```

At compile time:

- The compiler **checks syntax and types**.
- It verifies:
  - Is `Dog` a subclass of `Animal`? ✅
  - Does `Animal` have a method `sound()`? ✅

But — it does **not** allocate any memory yet.
It just ensures the code is *valid* and can be compiled into bytecode.



💾 2. Runtime phase

When you actually **run** the program:

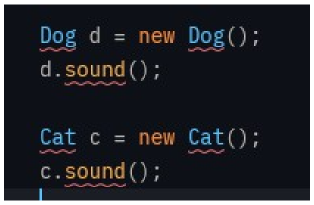
```java
obj = new Dog();
```

Now the **JVM executes** `new Dog()`, and:

- It allocates **memory on the heap** for the `Dog` object.
- Initializes its instance variables.
- Calls the **Dog() constructor**.
- Stores the **reference (memory address)** of that new Dog object into `obj`.

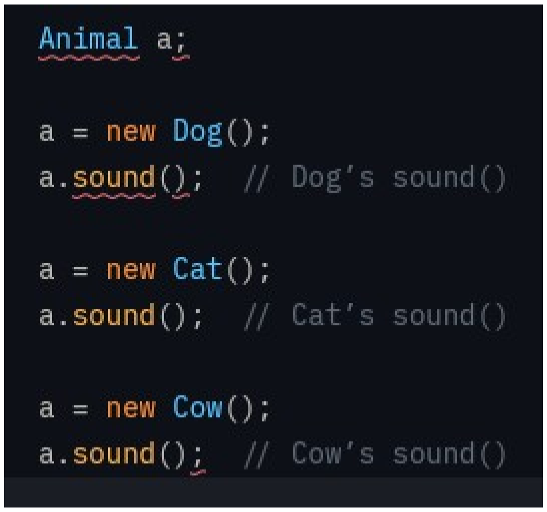
Q: If we know we want Dog sound to be execute. Why don't we create Dog reference variable? So there will be no runtime decision.
*Ans: We may have many Animal subclass like Dog, Cat, Cow. If you had 10 animal types (Dog, Cat, Cow, Lion, etc.), you'd need **10 different variable types** and **10 code paths** — not flexible. If we refer Parent class,it will flexible, reusable, and dynamic.*

Without Polymorphism:



```java
Dog d = new Dog();
d.sound();

Cat c = new Cat();
c.sound();
```

With Polymorphism:



```
Animal a;

a = new Dog();
a.sound();  // Dog’s sound()

a = new Cat();
a.sound();  // Cat’s sound()

a = new Cow();
a.sound();  // Cow’s sound()
```

Q: in java polymorphism let, car a=new bmw(); here 'a' is object with class car reference. why we use this type instead of bmw a=new bmw(); what is the benefit?

Ans: Because using the parent class type (Car) gives you flexibility, extensibility, and polymorphism.
1.
Car a = new BMW();
Car b = new Audi();
Car c = new Tesla();
All 3 objects can be handled using the same Car type.

If you use:
BMW a = new BMW();
then a can only reference BMW objects.
You lose flexibility.
…………
2.
Car myCar;
myCar = new BMW();
// Later you want to change
myCar = new Tesla();
You don’t need to change variable type.

But if you used:
BMW myCar = new BMW();
You must rewrite code everywhere when you change BMW → Tesla.
This is extremely bad for large projects.
…………….
3.
```
void service(Car c) {
   c.start();
   c.stop();
}
```

Now you can pass any Car child:
service(new BMW());
service(new Audi());
service(new Tesla());
If you used BMW type, then the method could only take BMW.
…………..
4.
Car car;

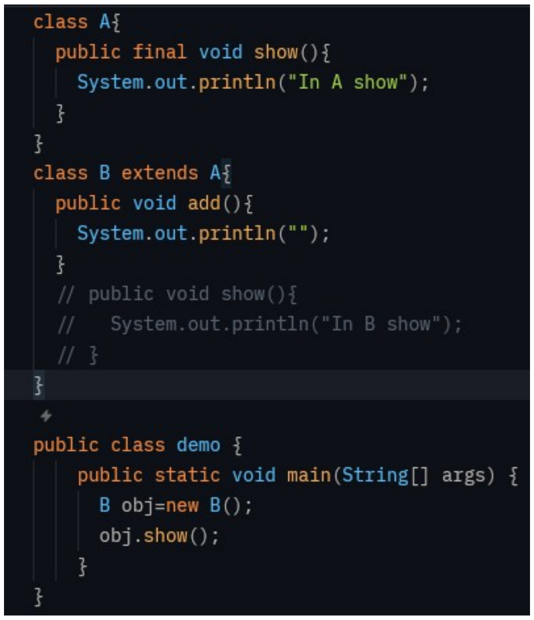
if(choice == 1) car = new BMW();
else if(choice == 2) car = new Tesla();
else car = new Audi();

car.start();
car.stop();
car.start() works because of runtime polymorphism.

If you used:
BMW car = new BMW();
Then you could never switch to Tesla or Audi.

----------x----------

Final: *make variable constant.*
For Class: if class B wants to inherit Class A. If we don't want inherit A use final in class A. So B can't inherit A.
For Method:



```
class A{
  public final void show(){
    System.out.println("In A show");
  }
}
class B extends A{
  public void add(){
    System.out.println("");
  }
  // public void show(){
  //   System.out.println("In B show");
  // }
}

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

if we don't use "final" in class A show() method, we can override that method on class B which inherit Class A. if Use then can't override show() method(Class B). That's why commented(otherwise compiler will give error). So "final" stops method overriding.

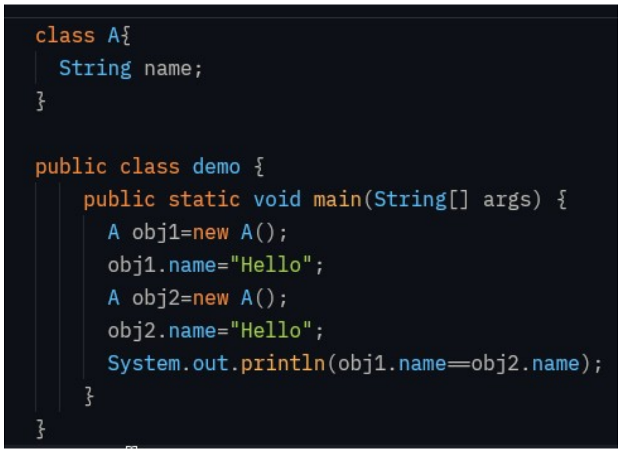
-----x-----

Learn Object Class. Min 7.22hr
https://www.youtube.com/watch?v=4XTsAAHW_Tc&t=27731s

----------x--------

Data comparing:
****If 2 object created with same values. Still they are not same. Cause Java allocates new memory on the heap each time. Even though the values are equal, the memory addresses (references) are different. If we check:*
return obj1==obj2; // let set all value for both object same.
Still return false.

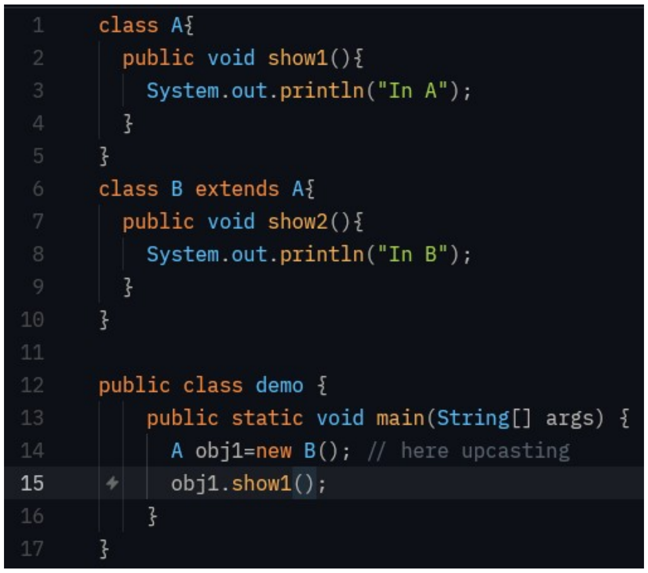
Find a way to check. Below mentioned is not good.
Return true



```java
class A{
  String name;
}

public class demo {
    public static void main(String[] args) {
      A obj1=new A();
      obj1.name="Hello";
      A obj2=new A();
      obj2.name="Hello";
      System.out.println(obj1.name==obj2.name);
    }
}
```

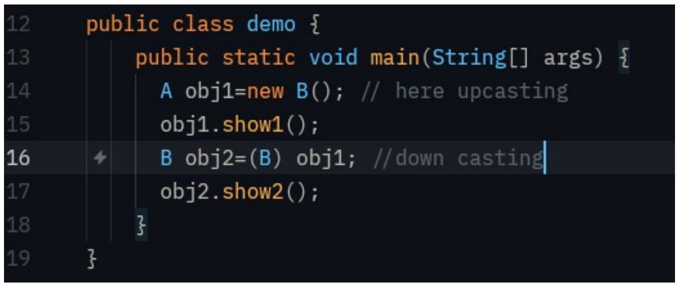
---------x--------

Up casting/ Down casting:



```java
1   class A{
2     public void show1(){
3       System.out.println("In A");
4     }
5   }
6   class B extends A{
7     public void show2(){
8       System.out.println("In B");
9     }
10  }
11
12  public class demo {
13      public static void main(String[] args) {
14        A obj1=new B(); // here upcasting
15        obj1.show1();
16      }
17  }
```

In line 14; this is up-casting. We are creating B object referring to class A.
Down casting: *We need down-casting because we can't print "show2()" method. To call :*

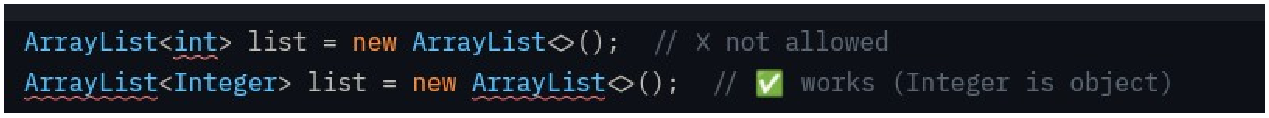


```java
12  public class demo {
13      public static void main(String[] args) {
14        A obj1=new B(); // here upcasting
15        obj1.show1();
16        B obj2=(B) obj1; //down casting
17        obj2.show2();
18      }
19  }
```

Here "obj1" is object of class B. But in line 16; we casting to class B, because type of 'obj1' is referred to class A. That's why we need to down-cast from Class A to Class B.
These castings are similar to Typecasting on number value.(int)(double)

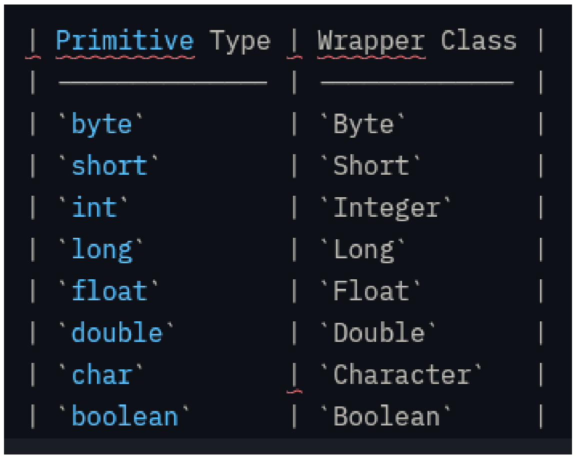
--------x-------

Wrapper Classes: *Java has primitive data type like int, double, char, short, byte, long, float, boolean. These 8 type. In some cases Java features work only with objects, not primitives like Collections (ArrayList, HashMap, etc.).*



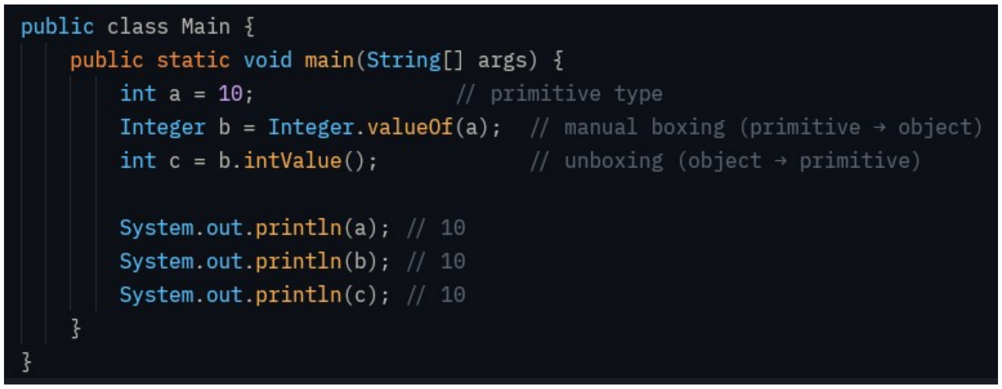
```java
ArrayList<int> list = new ArrayList<>();  // ✗ not allowed
ArrayList<Integer> list = new ArrayList<>();  // ✅ works (Integer is object)
```

Wrapper classes converts a ***primitive*** *→* ***object*** *and vice versa.*



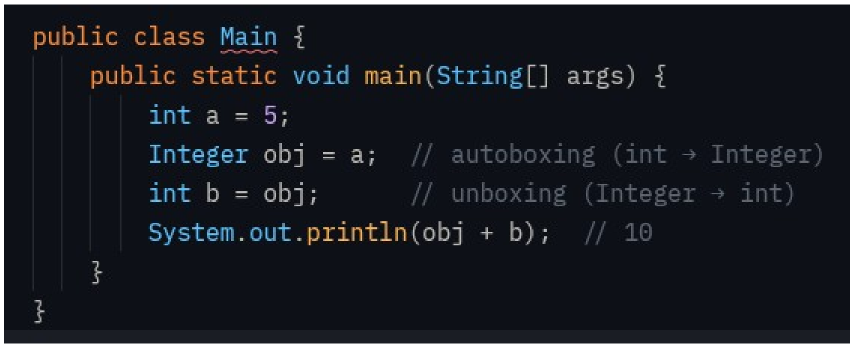
Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Example:



```java
public class Main {
    public static void main(String[] args) {
        int a = 10;                  // primitive type
        Integer b = Integer.valueOf(a);  // manual boxing (primitive → object)
        int c = b.intValue();            // unboxing (object → primitive)

        System.out.println(a); // 10
        System.out.println(b); // 10
        System.out.println(c); // 10
    }
}
```

You don't need to call valueOf() or intValue() manually anymore — compiler does it for you.

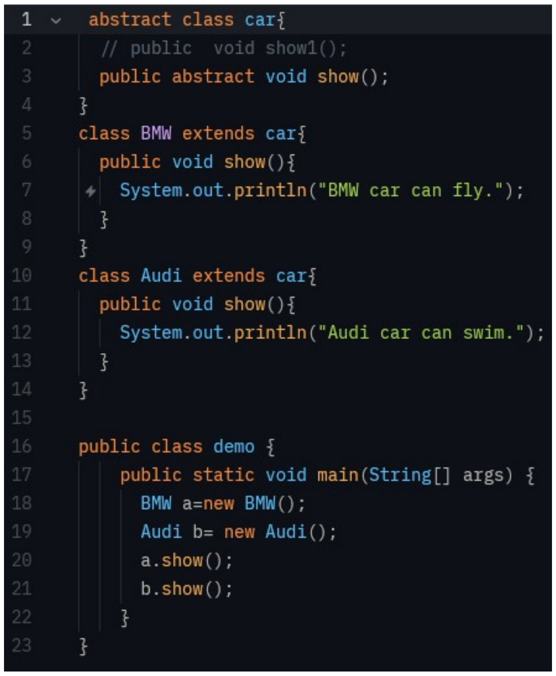


```java
public class Main {
    public static void main(String[] args) {
        int a = 5;
        Integer obj = a;   // autoboxing (int → Integer)
        int b = obj;       // unboxing (Integer → int)
        System.out.println(obj + b);   // 10
    }
}
```

--------------x-------------

Abstract/Abstraction:*Data abstraction is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either abstract classes or* ***interfaces.***

The abstract keyword in Java is used to create incomplete classes or methods—things that cannot be used directly and must be completed by subclasses.

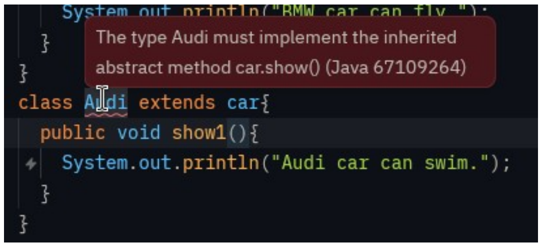


```java
1  abstract class car{
2    // public  void show1();
3    public abstract void show();
4  }
5  class BMW extends car{
6    public void show(){
7      System.out.println("BMW car can fly.");
8    }
9  }
10 class Audi extends car{
11   public void show(){
12     System.out.println("Audi car can swim.");
13   }
14 }
15
16 public class demo {
17     public static void main(String[] args) {
18       BMW a=new BMW();
19       Audi b= new Audi();
20       a.show();
21       b.show();
22     }
23 }
```

**Abstract class can't be instantiated.(cna't create object).*

**Class which contain abstract method, that class have to be Abstract Class(Line:1). Abstract method must be in abstract class. But abstract class may not have abstract method at all.*

** Abstract method must be defined in that subclass(BMW, Audi) which class inherit the parent class(Car). Line:5,10. See Error image below:*



```java
    System.out.println("BMW car can fly.");
  }
}
class Audi extends car{
  public void show1(){
    System.out.println("Audi car can swim.");
  }
}
```

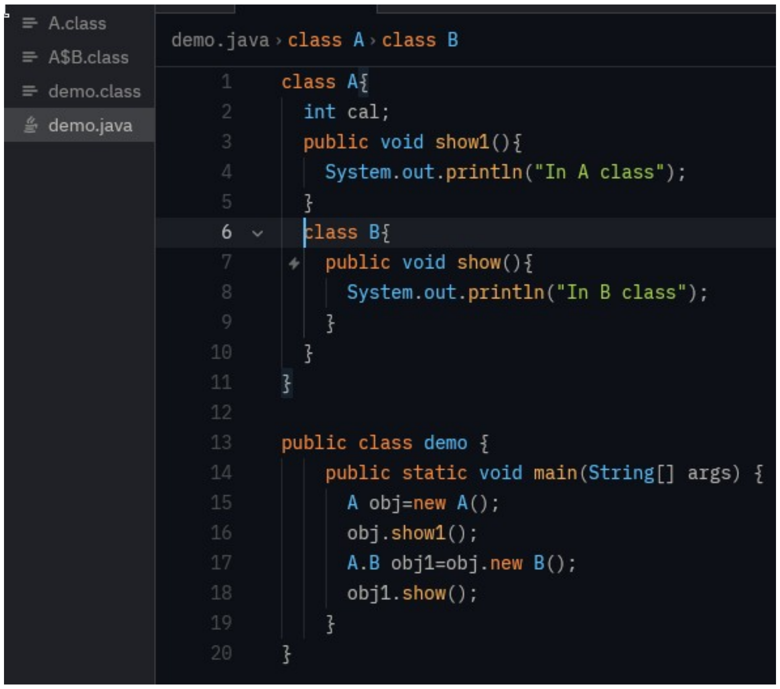
The type Audi must implement the inherited abstract method car.show() (Java 67109264)

if that class(BMW, Audi) can't define the method, then those class(BMW,Audi) must be Abstract class. Here function overriding.

** Abstract Methods are only declared in class. Definitions will be on subclass.*

-----------x---------

Inner Class: *Class inside class.*

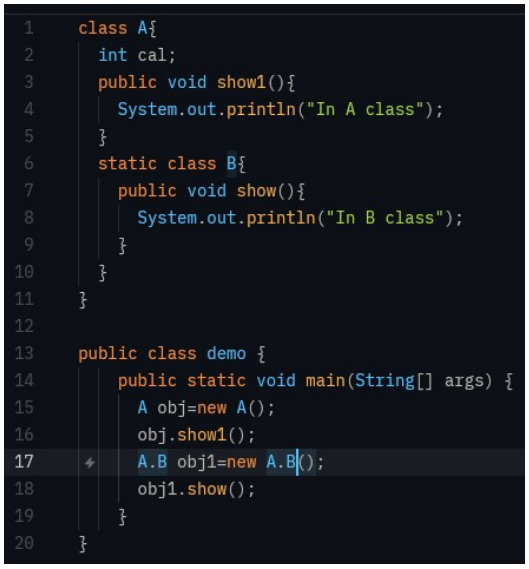


```
A.class
A$B.class
demo.class
demo.java
```

```
demo.java › class A › class B
```

```java
 1  class A{
 2    int cal;
 3    public void show1(){
 4      System.out.println("In A class");
 5    }
 6    class B{
 7      public void show(){
 8        System.out.println("In B class");
 9      }
10    }
11  }
12
13  public class demo {
14      public static void main(String[] args) {
15        A obj=new A();
16        obj.show1();
17        A.B obj1=obj.new B();
18        obj1.show();
19      }
20  }
```

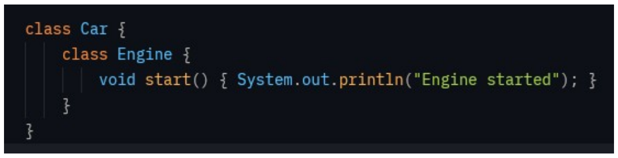
*When compile java file, for every class we have *.class file on file section. But for inner class it will be like (A$B) means B class belongs to class A. to create inner class obj we need outer class object. See above. Line:17;*
in case we make inner class static:



```java
 1  class A{
 2    int cal;
 3    public void show1(){
 4      System.out.println("In A class");
 5    }
 6    static class B{
 7      public void show(){
 8        System.out.println("In B class");
 9      }
10    }
11  }
12
13  public class demo {
14      public static void main(String[] args) {
15        A obj=new A();
16        obj.show1();
17        A.B obj1=new A.B();
18        obj1.show();
19      }
20  }
```

Only Inner class can be static. Outer class(class A) never be static.

Uses Case:



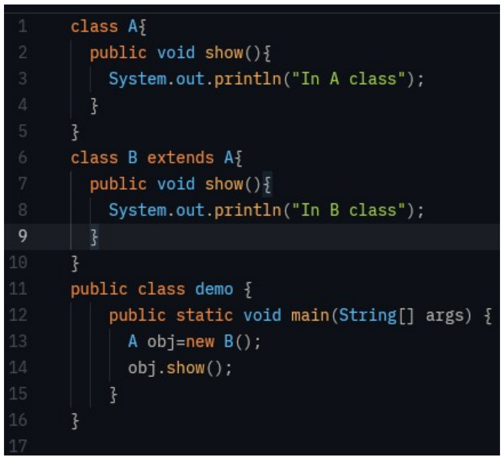
```java
class Car {
    class Engine {
        void start() { System.out.println("Engine started"); }
    }
}
```

**Engine is part of Car*
**No need to make it a separate file*
**Code becomes clean and organized*

**Inner class can access private variable of parent class(Car).*
Also many other uses.

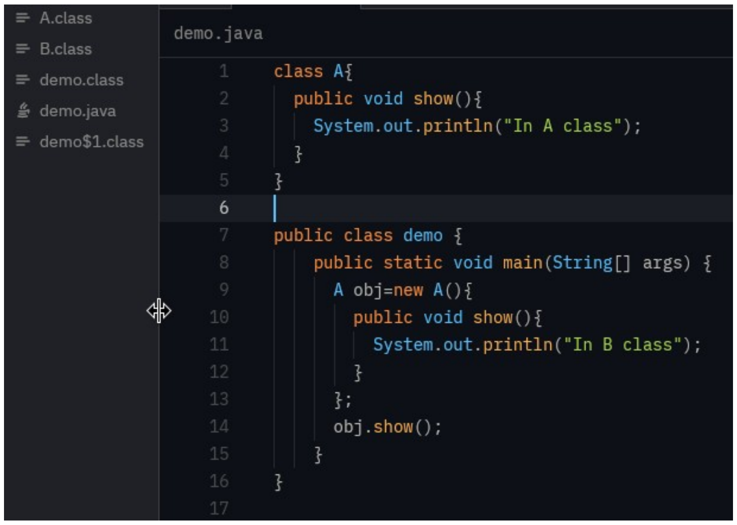
----------x--------

Anonymous Inner class:



```
1   class A{
2     public void show(){
3       System.out.println("In A class");
4     }
5   }
6   class B extends A{
7     public void show(){
8       System.out.println("In B class");
9     }
10  }
11  public class demo {
12      public static void main(String[] args) {
13        A obj=new B();
14        obj.show();
15      }
16  }
17
```

Fig:1

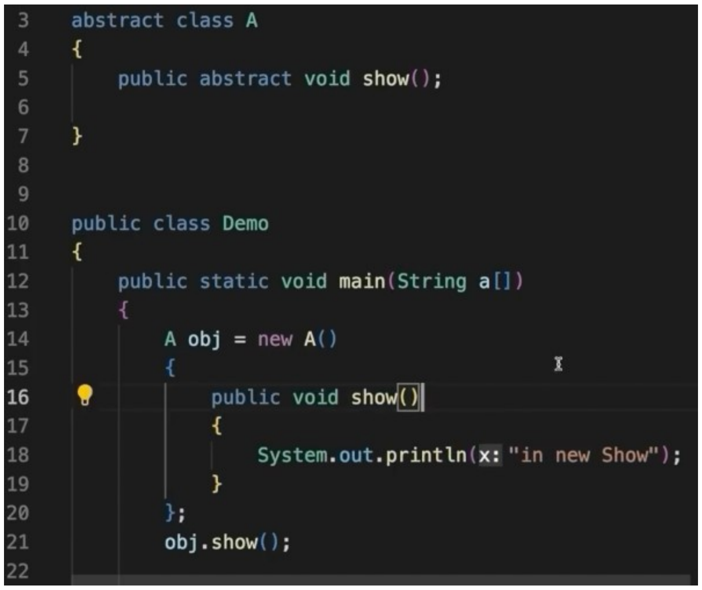


```
1   class A{
2     public void show(){
3       System.out.println("In A class");
4     }
5   }
6
7   public class demo {
8       public static void main(String[] args) {
9         A obj=new A(){
10          public void show(){
11            System.out.println("In B class");
12          }
13        };
14        obj.show();
15      }
16  }
17
```

Fig:2

in fig:1 function overriding. If class B show() function may need only for once. In that case we don't need another class like B class. We can use this show function while class A object creating (Fig:2). From line:10-12 this is inner class where parent class is demo. This class has no name. So it is anonymous inner class. On files section we see "demo$1" means that inner class belongs to demo class and it has no name that's why name "demo$1".

Abstract and anonymous inner class combined:

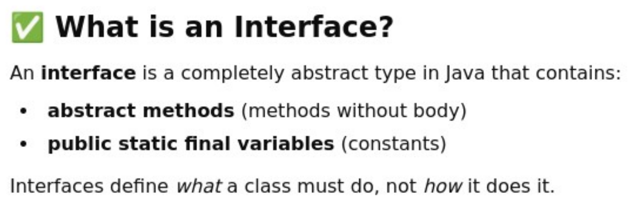


```
3   abstract class A
4   {
5       public abstract void show();
6
7   }
8
9
10  public class Demo
11  {
12      public static void main(String a[])
13      {
14          A obj = new A()
15          {
16              public void show()
17              {
18                  System.out.println(x: "in new Show");
19              }
20          };
21          obj.show();
22
```

Here class A is Abstract class. As we know we can't instantiate a abstract class. But we are doing on line 14; right? But no. we are not creating an object of class A, we are creating an object of inner class.(above). We can do this with polymorphism as we seen before. But if this show() method need only once. We don't need to create another *class instead we can do this, as above mentioned process. For single use. If we have multiple method we can do the same.*

------------x---------

Interface: *An interface is a blueprint of a class. It contains only method declarations, no method body. Interface is a blueprint of a class because interface will tell which method a class need to implement, its a class job to implement them. Interface show the design.*

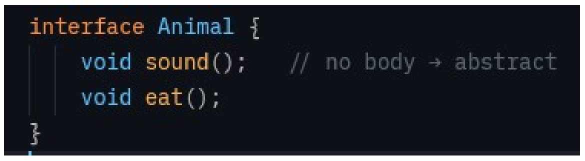


✅ What is an Interface?

An **interface** is a completely abstract type in Java that contains:

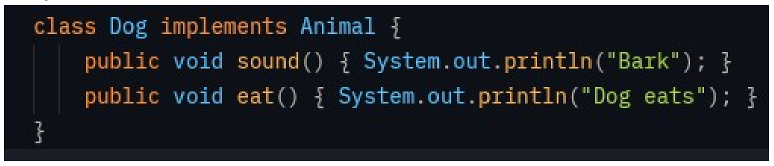
- **abstract methods** (methods without body)
- **public static final variables** (constants)

Interfaces define *what* a class must do, not *how* it does it.

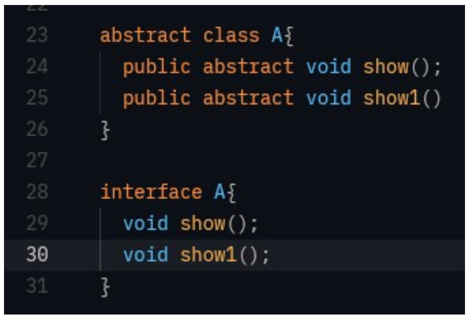


```java
interface Animal {
    void sound();   // no body → abstract
    void eat();
}
```

Every Animal must have sound() and eat() method. But interface doesn't tell how to do them.
A class implements the interface.

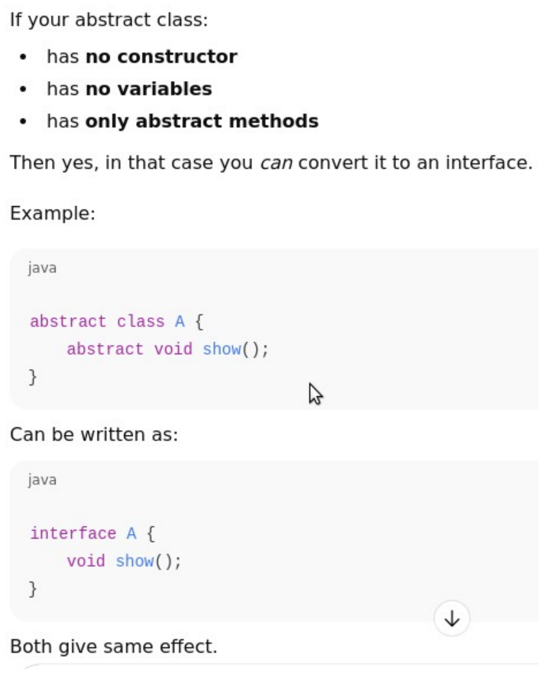


```java
class Dog implements Animal {
    public void sound() { System.out.println("Bark"); }
    public void eat() { System.out.println("Dog eats"); }
}
```



```java
23    abstract class A{
24      public abstract void show();
25      public abstract void show1()
26    }
27
28    interface A{
29      void show();
30      void show1();
31    }
```

Here line:23-26 and line:28-31(above); are similar. Means interface is abstract class with abstract methods only. Interface may contain variable(int, string etc type). But they are final(constant) and static. They must define during declaring. Thus interface provide 100% abstraction. Can't instantiate(create object) of an interface. By default all methods on interface are public. On line: 29,30 both methods are abstract. Others class which implement interface must define these methods.
** Interface reference type can be possible.*
Abstract class and interface are similar, not same. See below:



If your abstract class:

- has **no constructor**
- has **no variables**
- has **only abstract methods**

Then yes, in that case you *can* convert it to an interface.

Example:

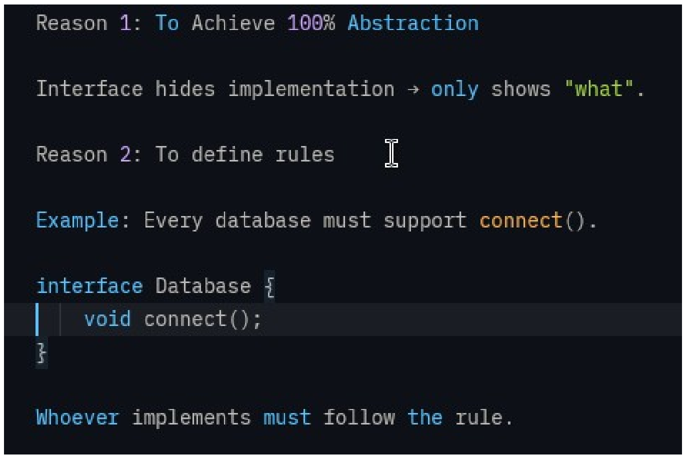
```java
abstract class A {
    abstract void show();
}
```

Can be written as:

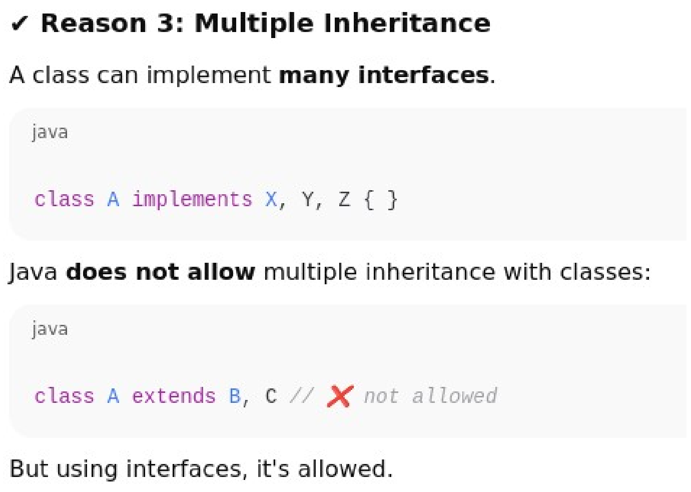
```java
interface A {
    void show();
}
```

Both give same effect.

Use of interface:



```
Reason 1: To Achieve 100% Abstraction

Interface hides implementation → only shows "what".

Reason 2: To define rules

Example: Every database must support connect().

interface Database {
    void connect();
}

Whoever implements must follow the rule.
```



✔ Reason 3: Multiple Inheritance

A class can implement **many interfaces**.

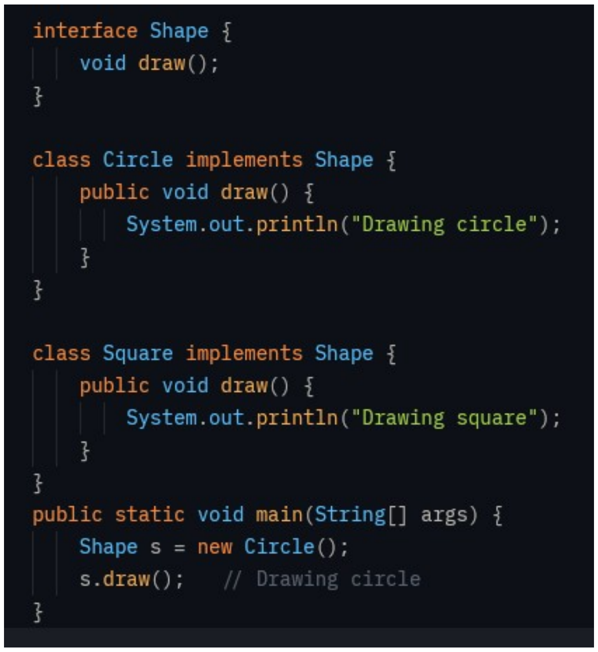
```java
class A implements X, Y, Z { }
```

Java **does not allow** multiple inheritance with classes:

```java
class A extends B, C // ❌ not allowed
```

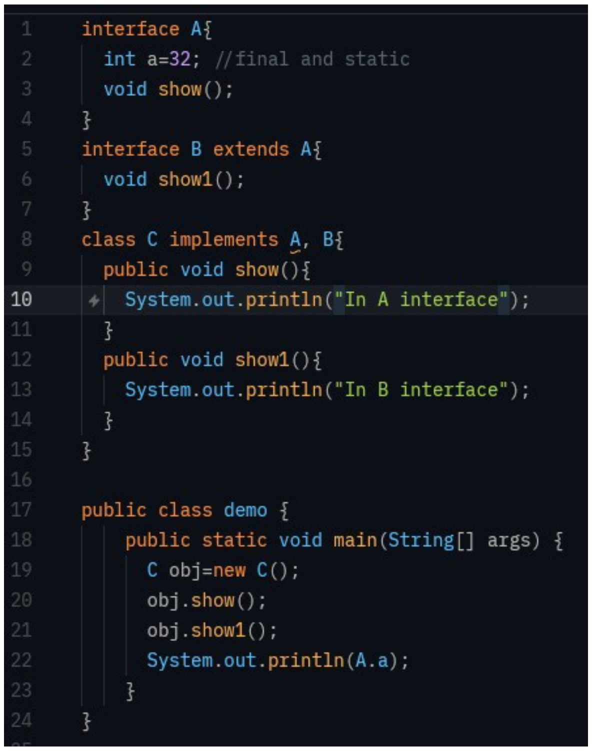
But using interfaces, it's allowed.

Example of using Interface:



```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing circle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing square");
    }
}
public static void main(String[] args) {
    Shape s = new Circle();
    s.draw();   // Drawing circle
}
```

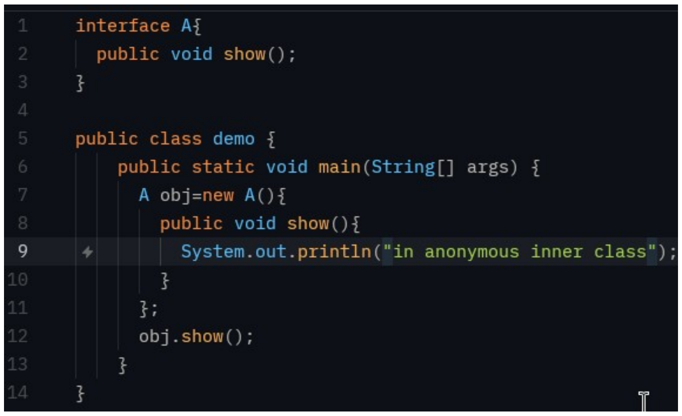
Another Example:



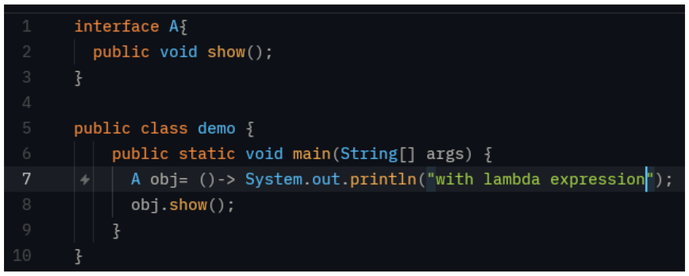
```
1   interface A{
2     int a=32; //final and static
3     void show();
4   }
5   interface B extends A{
6     void show1();
7   }
8   class C implements A, B{
9     public void show(){
10      System.out.println("In A interface");
11    }
12    public void show1(){
13      System.out.println("In B interface");
14    }
15  }
16
17  public class demo {
18      public static void main(String[] args) {
19        C obj=new C();
20        obj.show();
21        obj.show1();
22        System.out.println(A.a);
23      }
24  }
```

here(Above)
**all methods and variables are public. No need to mention. But while implementing its necessary to mention public.*
**interface can inherit another interface.*
** multiple inheritance for interface. In java multiple inheritance for class not allowed. Allowed for interface.*
** Variables are abstract. It belongs to interface as class. So object need to use them. Use class/interface name as line:22;*
** Find why obj reference is "B"?*
** after java 8, we can define methods on interface.*

Types of Interface:
1. Normal Interface: *Interface that have more than 1 abstract method. Used for abstraction + multiple inheritance.*
2. Functional interface(SAM): *Which have only 1 abstract method. Single abstract method(SAM). Used mainly in Lambda Expressions.*

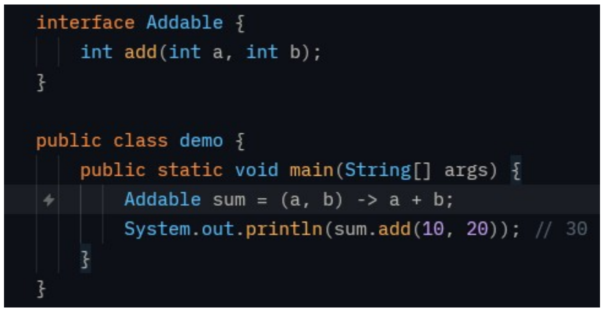


```java
1   interface A{
2     public void show();
3   }
4
5   public class demo {
6       public static void main(String[] args) {
7           A obj=new A(){
8               public void show(){
9                   System.out.println("in anonymous inner class");
10              }
11          };
12          obj.show();
13      }
14  }
```



```java
1   interface A{
2     public void show();
3   }
4
5   public class demo {
6       public static void main(String[] args) {
7           A obj= ()-> System.out.println("with lambda expression");
8           obj.show();
9       }
10  }
```

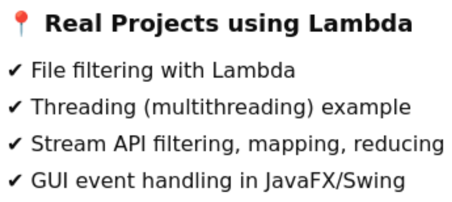
Line:7; "- >" is lambda expression/operator. With lambda expression we can reduce some code.
For more statement we can use "{}" after "- >".
with parameterized lambda expression:(Below)



```java
interface Addable {
    int add(int a, int b);
}

public class demo {
    public static void main(String[] args) {
        Addable sum = (a, b) -> a + b;
        System.out.println(sum.add(10, 20)); // 30
    }
}
```

***** *Lambda expression only works with functional interface.*
A ***Lambda Expression*** *in Java is a short form of writing a function (method) without a name.*
It is mainly used to reduce code when working with functional interfaces (interfaces that have only one abstract method).
Before lambda, writing an anonymous method was long and verbose.
Lambda makes code short, clean, and readable—specially useful in streams, event handling, sorting, threading, etc.

**** This is java 8 feature.*



📍 **Real Projects using Lambda**

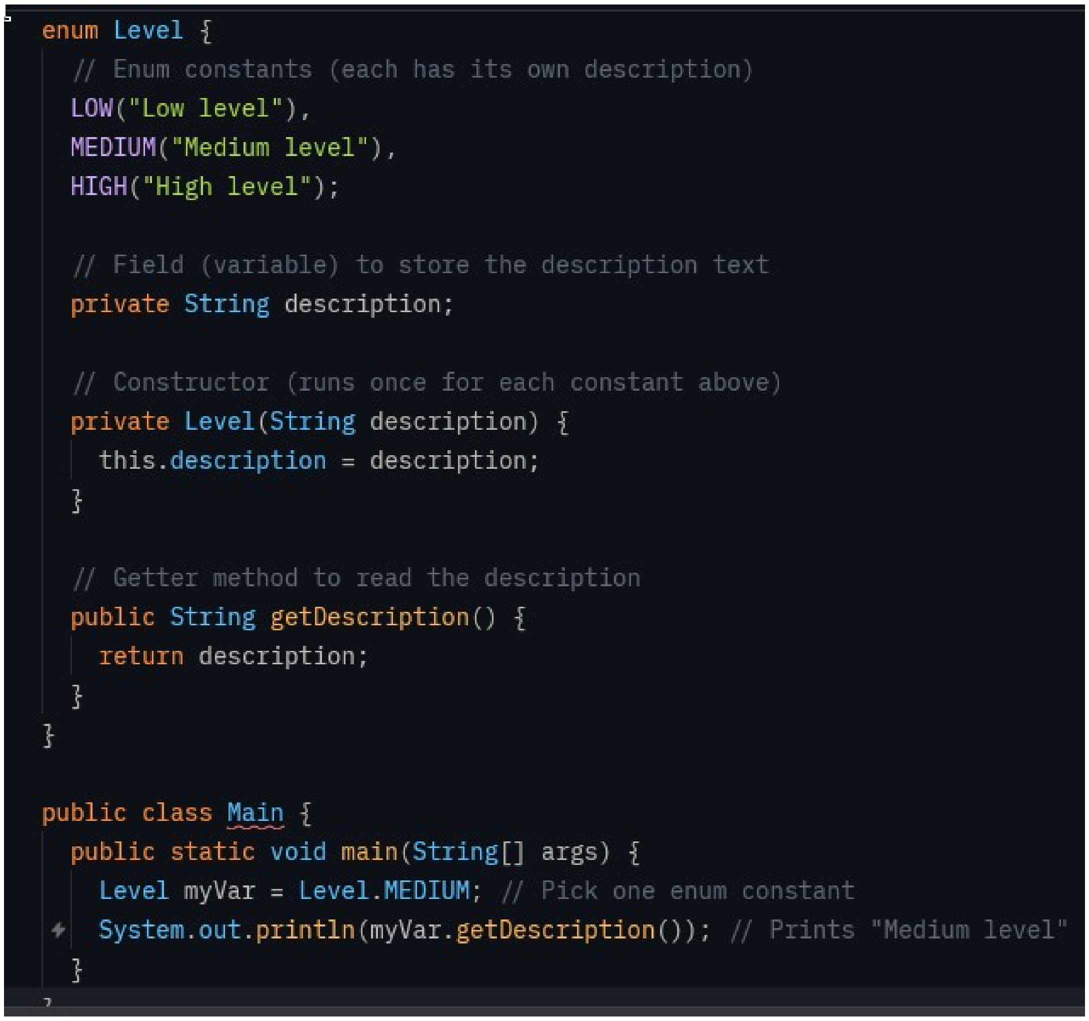
✔ File filtering with Lambda
✔ Threading (multithreading) example
✔ Stream API filtering, mapping, reducing
✔ GUI event handling in JavaFX/Swing

3. Marker Interface: *Which have no method. Used to give a special tag/marker to a class. Used as a label.*
Compiler/JVM treats marked classes differently.
Examples:
Serializable, Cloneable, Remote

Serialization means: Let we have an object. We want to store that object values on our HDD. Suppose we playing a game. In the middle of the game we pause and save the current status, so can resume later. By serialization we can save some date and continue from that. This need special permission. By marker interface we can give permission.

----------x----------

Enum(Enumeration): *see w3school*



```
enum Level {
  // Enum constants (each has its own description)
  LOW("Low level"),
  MEDIUM("Medium level"),
  HIGH("High level");

  // Field (variable) to store the description text
  private String description;

  // Constructor (runs once for each constant above)
  private Level(String description) {
    this.description = description;
  }

  // Getter method to read the description
  public String getDescription() {
    return description;
  }
}

public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM; // Pick one enum constant
    System.out.println(myVar.getDescription()); // Prints "Medium level"
  }
```

full execution flow from chatgpt(below):

-----0. *JVM starts the program*
JVM loads the Main class.
Before it can run main(), it must load everything the class uses — including the enum Level.

-----1. *JVM loads the enum class Level*
When Level is loaded, Java creates all enum constants automatically.
This line in enum:

LOW("Low level"),
MEDIUM("Medium level"),
HIGH("High level");

causes three constructor calls, BEFORE main() runs.
So JVM executes:

new Level("Low level")
new Level("Medium level")
new Level("High level")

—in this exact order.

-----2. *For each constant, JVM runs the constructor*
👉 *For LOW*
private Level(String description) {
   this.description = description;
}

JVM calls it like:

Level LOW = new Level("Low level");
this.description = "Low level";
----------Same for medium and high

Now all enum constants are fully created.

-----3. *JVM now starts executing main()*

public static void main(String[] args) {
   Level myVar = Level.MEDIUM;
   System.out.println(myVar.getDescription());
}

-----4. *Level myVar = Level.MEDIUM;*
No new object is created.
myVar simply references the already-created object MEDIUM.
Think of it as a pointer:

myVar ---> MEDIUM object

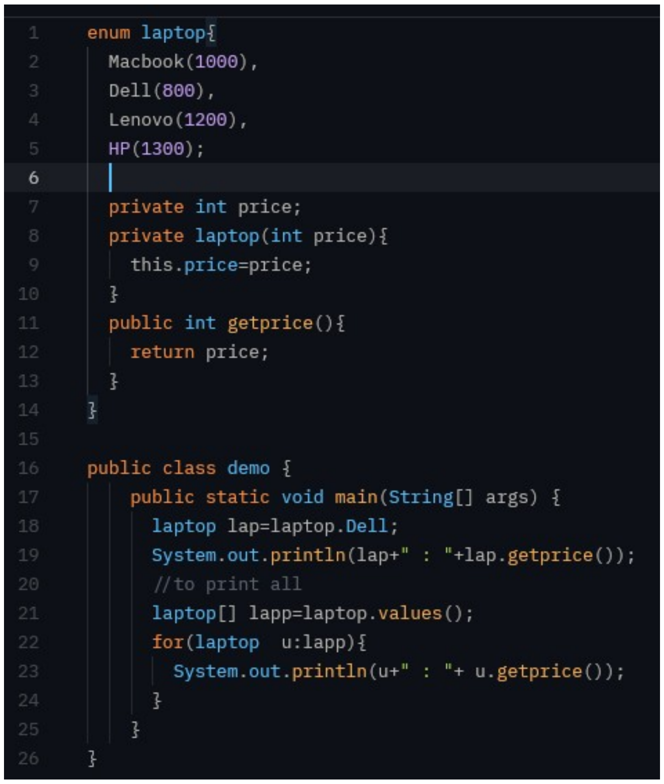
------5. *myVar.getDescription()*
Java calls this method:

public String getDescription() {
   return description;
}

Since myVar is pointing to the MEDIUM constant:
return "Medium level";

Output: *Output printed*
"Medium level"

Another One:

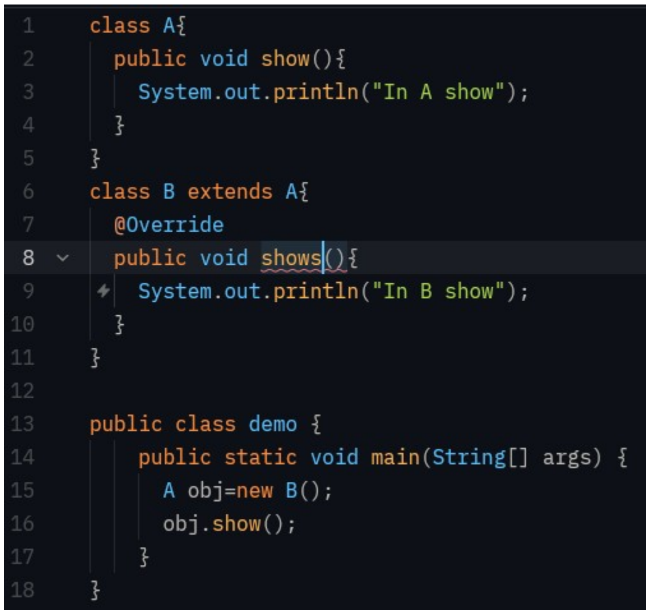


```
1   enum laptop{
2       Macbook(1000),
3       Dell(800),
4       Lenovo(1200),
5       HP(1300);
6
7       private int price;
8       private laptop(int price){
9           this.price=price;
10      }
11      public int getprice(){
12          return price;
13      }
14  }
15
16  public class demo {
17      public static void main(String[] args) {
18          laptop lap=laptop.Dell;
19          System.out.println(lap+" : "+lap.getprice());
20          //to print all
21          laptop[] lapp=laptop.values();
22          for(laptop  u:lapp){
23              System.out.println(u+" : "+ u.getprice());
24          }
25      }
26  }
```

Here compiler doesn't know (line:2-5) what values are they. To identify we need a variable. A constructor to store the specific value to specific enum value(dell, lenovo etc).

----------x---------

Annotation: *Annotations in Java are* ***special markers*** *(like labels or metadata) that you attach to classes, methods, variables, etc.*
They ***do NOT change the code behavior directly****, but they give* ***extra information*** *to:*



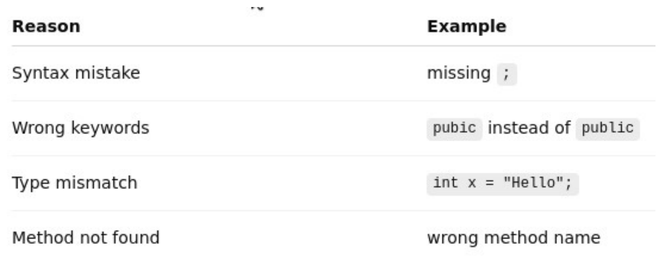
```
class A{
    public void show(){
        System.out.println("In A show");
    }
}
class B extends A{
    @Override
    public void shows(){
        System.out.println("In B show");
    }
}

public class demo {
    public static void main(String[] args) {
        A obj=new B();
        obj.show();
    }
}
```

- *the* ***compiler***
- ***tools*** *(IDEs, frameworks)*
- *the* ***JVM*** *at runtime*

Here we want to print “In B show” by method override. But will print “In A show”.(if @override annotation not used)
Here in line:8; this shows() method meant to override of class A show() method. (But intentionally make those name diff, but I real life(unintentionallys) we may make mistake by making diff name of the method). While using “@override” annotation, compiler will check to override, but will not find show() method on class B. So give error. See line:8; red underline on shows() method.
** It means it want to supply extra info to the compiler or to the run time.*
** Annotations are usually on framework heavily.*
There are many other annotations.

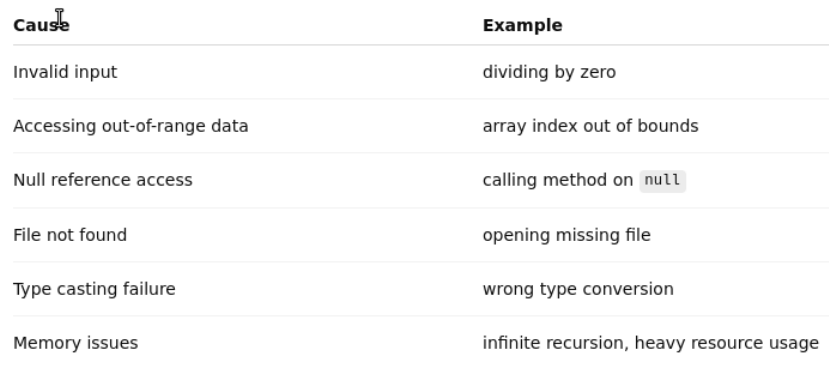
------------***x***---------

Exception Handling: *There are 3 types of error.*
1. Compile time error: These errors occur before the program runs — during compilation. They happen due to incorrect syntax or rules violations.



Reason	Example
Syntax mistake	missing `;`
Wrong keywords	`pubic` instead of `public`
Type mismatch	`int x = "Hello";`
Method not found	wrong method name

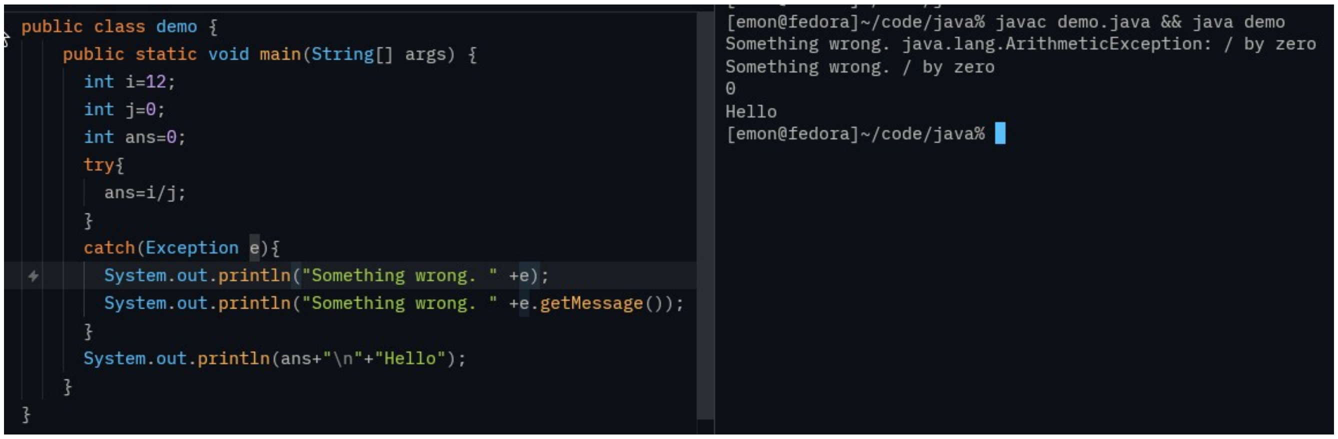
2. Runtime error: A ***runtime error*** *is an error that occurs* ***while the program is running****, after successful compilation. The program compiles correctly (no compile-time error), but crashes or behaves unexpectedly during execution.*



Cause	Example
Invalid input	dividing by zero
Accessing out-of-range data	array index out of bounds
Null reference access	calling method on `null`
File not found	opening missing file
Type casting failure	wrong type conversion
Memory issues	infinite recursion, heavy resource usage

3. Logical error: Wrong logic.

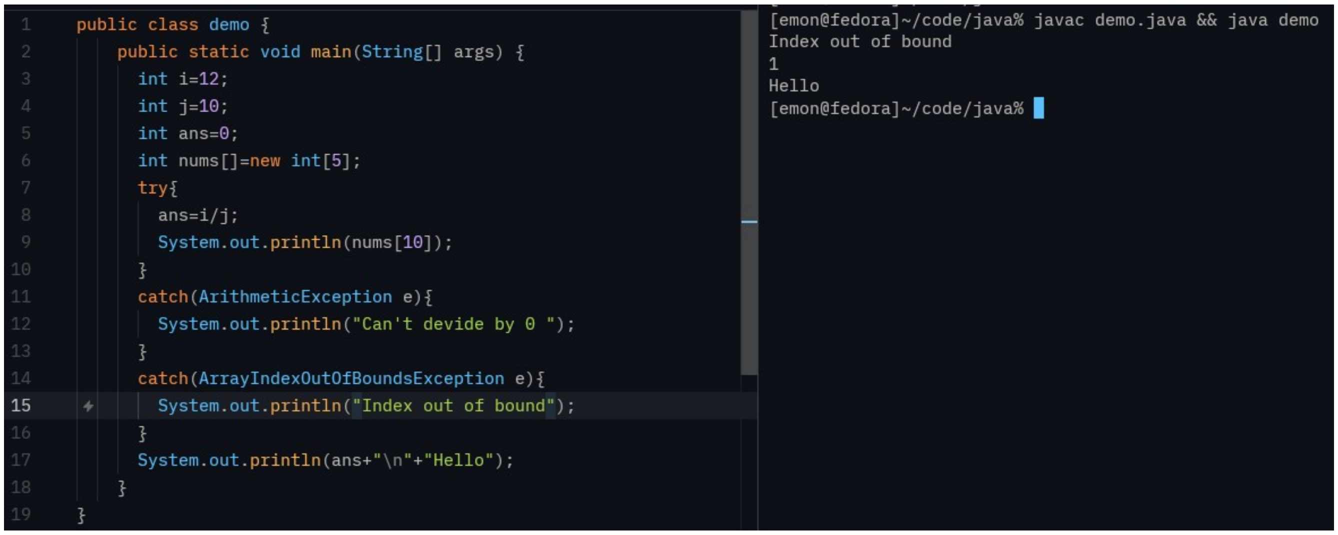
These runtime error is called exception. We need to handle them.



```java
public class demo {
    public static void main(String[] args) {
        int i=12;
        int j=0;
        int ans=0;
        try{
            ans=i/j;
        }
        catch(Exception e){
            System.out.println("Something wrong. " +e);
            System.out.println("Something wrong. " +e.getMessage());
        }
        System.out.println(ans+"\n"+"Hello");
    }
}
```

```
[emon@fedora]~/code/java% javac demo.java && java demo
Something wrong. java.lang.ArithmeticException: / by zero
Something wrong. / by zero
0
Hello
[emon@fedora]~/code/java%
```

*Catch block will be execute only while exception occur. Here "Exception" is a class. Ans 'e' is object/ reference variable of that class. Like → **Exception e=new Exception();***
When an exception happens, the JVM creates an object of Exception (or its subclass) and passes it to `e`*. don't need to create.*

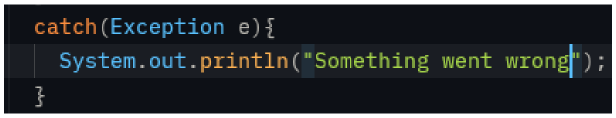


```java
1   public class demo {
2       public static void main(String[] args) {
3           int i=12;
4           int j=10;
5           int ans=0;
6           int nums[]=new int[5];
7           try{
8               ans=i/j;
9               System.out.println(nums[10]);
10          }
11          catch(ArithmeticException e){
12              System.out.println("Can't devide by 0 ");
13          }
14          catch(ArrayIndexOutOfBoundsException e){
15              System.out.println("Index out of bound");
16          }
17          System.out.println(ans+"\n"+"Hello");
18      }
19  }
```

```
[emon@fedora]~/code/java% javac demo.java && java demo
Index out of bound
1
Hello
[emon@fedora]~/code/java%
```

Here let "i=0" then in try block in line:8 there will be an exception, so index out of bound check will not occur. Direct jump to catch block. Arithmetic exception clock will receive the exception. If i!=0 line:9; exception will occur and Array index out of bound will occur and Array index out of bound catch block will receive the exception.
Only one exception occurs at a time because as soon as an error happens inside `try`*,*
execution stops at that line, and the next catch is checked.

Like ArrayOutOfBoundException, ArithmeticException there are many exception classes.
But "Exception" is parent class. "Exception" handles all exceptions. When we don't know about exception specifically, can use only Exception. Below



```java
catch(Exception e){
    System.out.println("Something went wrong");
}
```