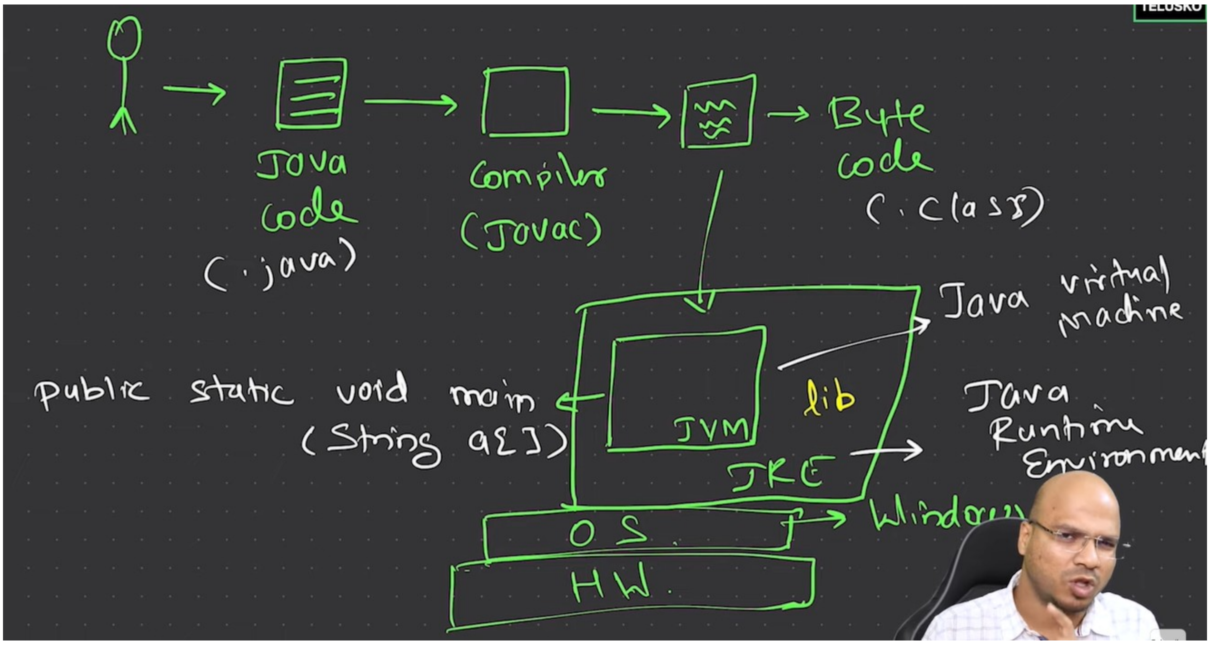
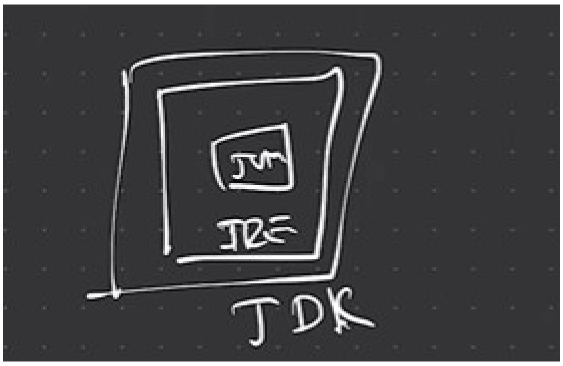
Java Notes





>> JVM does not look for just main method, it looks for entire method signature “public static void main(String a[])”

>> Java is called WORA -> Write Once Run Anywhere

>> JIT: Just In Time Compiler: The Just-In-Time (JIT) compiler is a component of the runtime environment that improves the performance of Java™ applications by compiling bytecodes to native machine code at run time

>> AOT: Ahead Of Time Compiler: It used to compile byte code to native code prior to the execution by the JVM.

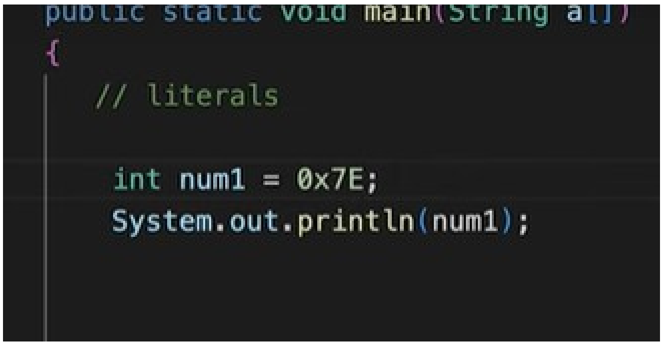
>> Java Features:

Simple, Secure, Portable, Object-Oriented, Robust, Multithreaded, Architechture-neutral, Interpreted, High performance, Distributed, Dynamic.

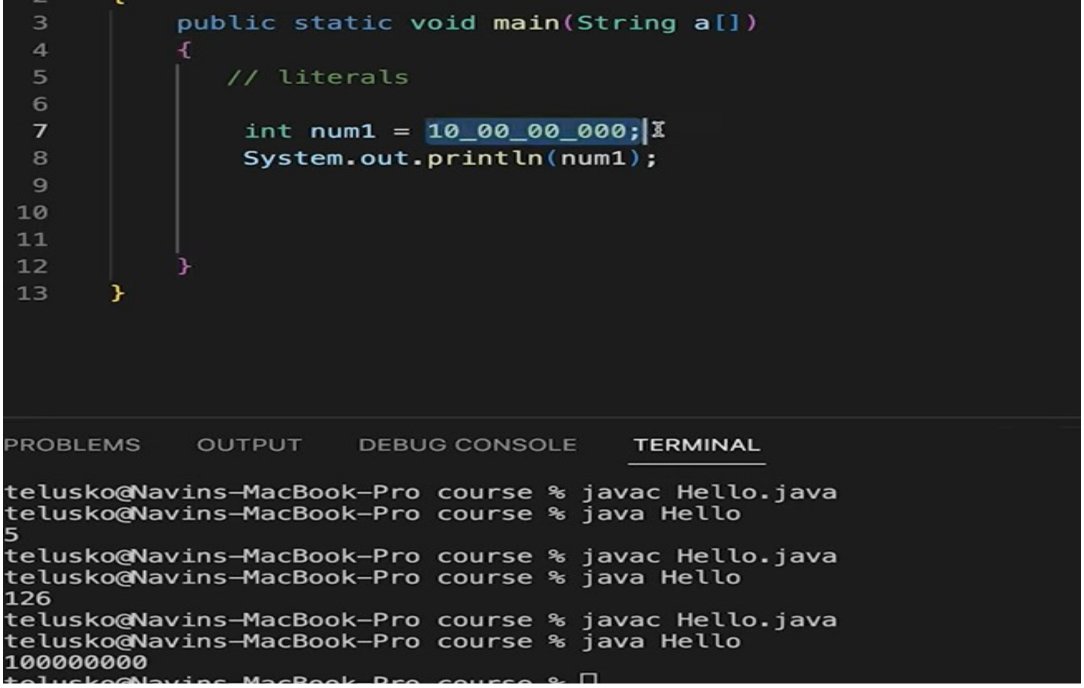
Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so. That's why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os environment which makes java programs more secure.

Robust: The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

>> beginning with JDK 8 it is now possible to define a default implementation for a method specified by an interface. I f no implementation for a default method is created the the default defined by the interface is used.

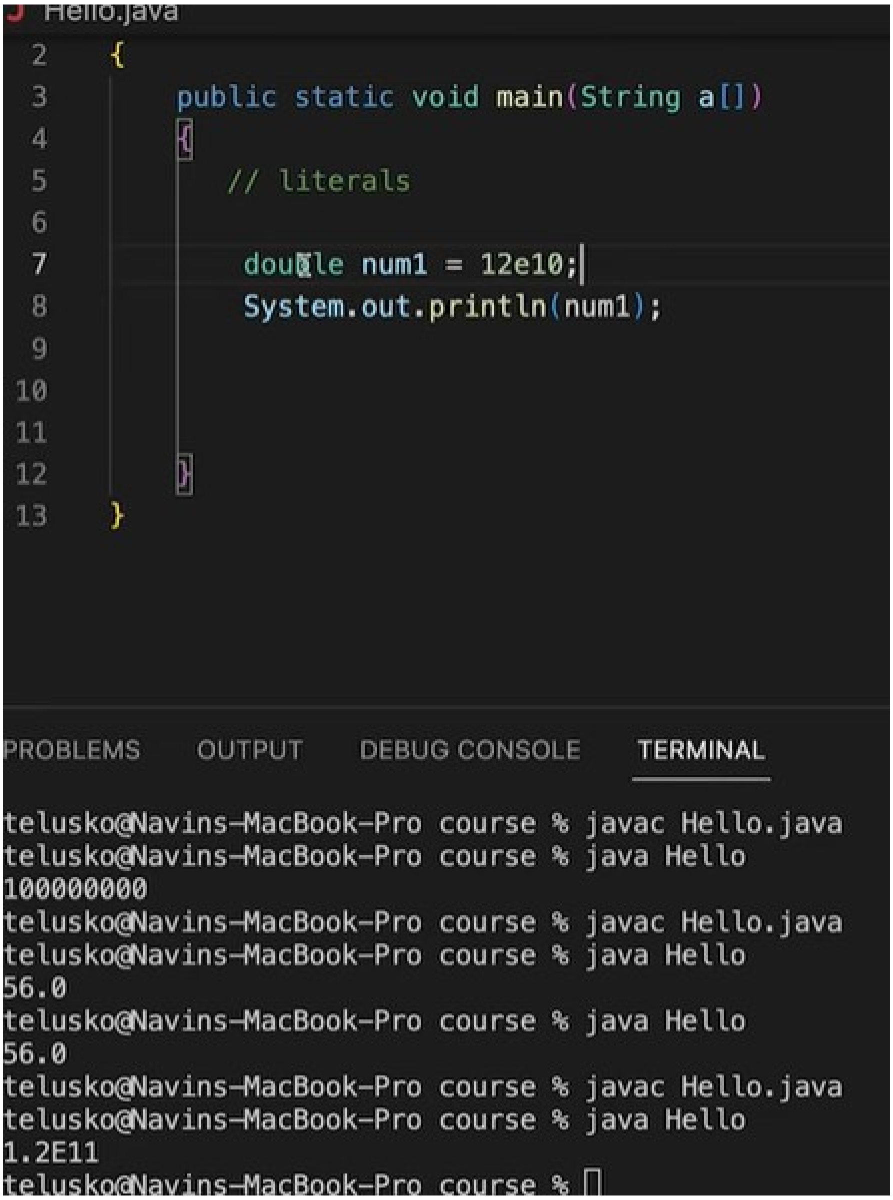


```
public static void main(String a[])
{
    // literals

     int num1 = 0x7E;
     System.out.println(num1);
```



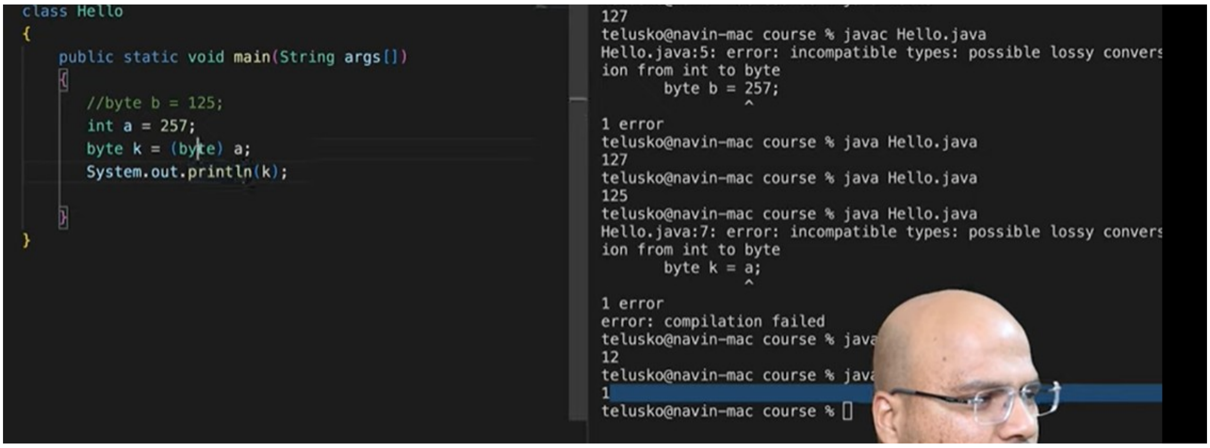
```
    public static void main(String a[])
    {
        // literals

        int num1 = 10_00_00_000;
        System.out.println(num1);



    }
}
```

```
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
5
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
126
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
100000000
telusko@Navins-MacBook-Pro course %
```



```
{
    public static void main(String a[])
    {
        // literals

        double num1 = 12e10;
        System.out.println(num1);



    }
}
```

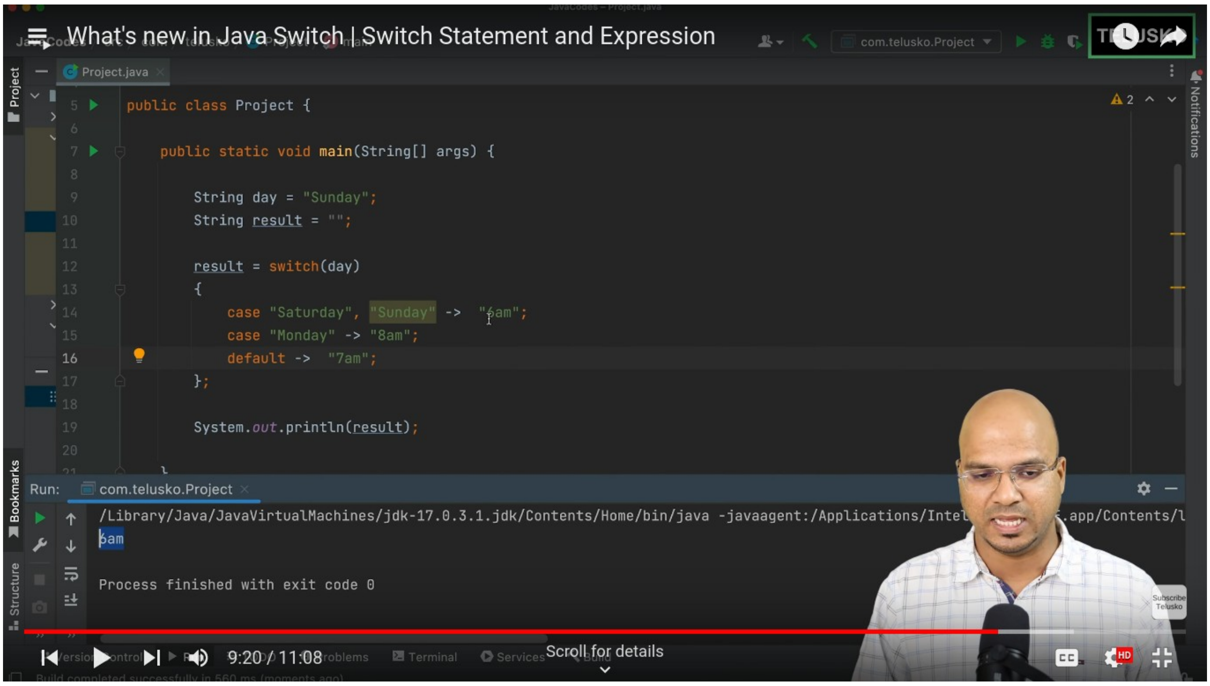
```
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
100000000
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
56.0
telusko@Navins-MacBook-Pro course % java Hello
56.0
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
1.2E11
telusko@Navins-MacBook-Pro course %
```



```java
class Hello
{
    public static void main(String args[])
    {
        //byte b = 125;
        int a = 257;
        byte k = (byte) a;
        System.out.println(k);
    }
}
```

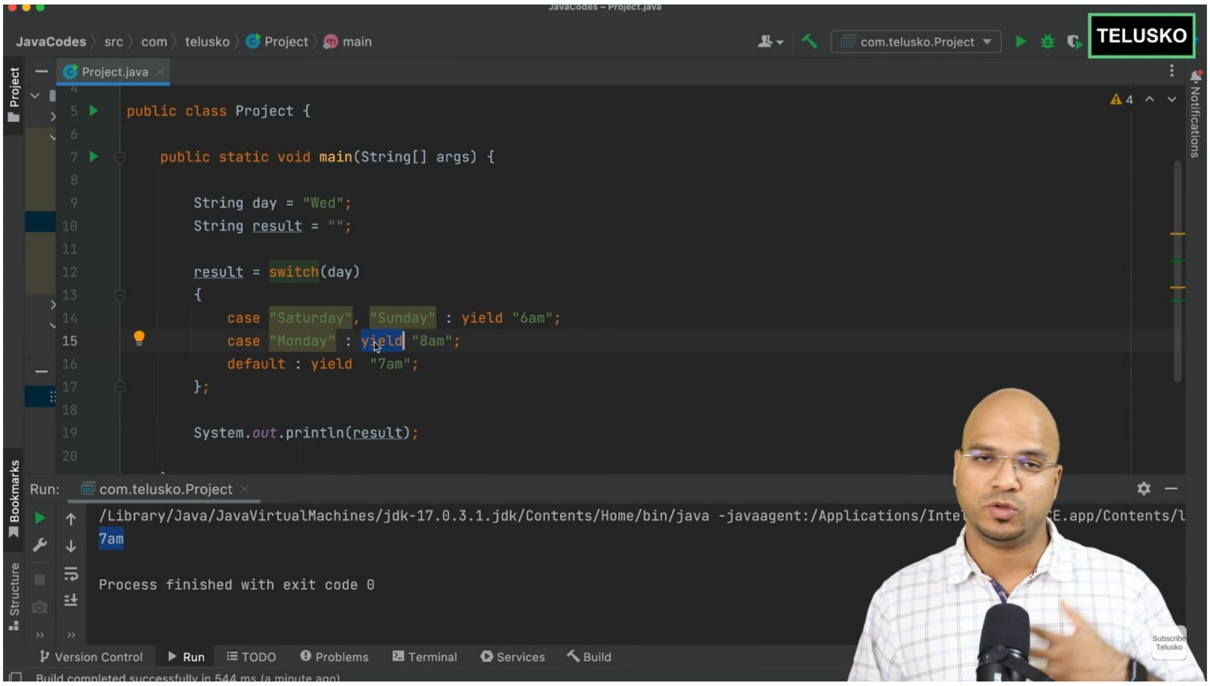
```
127
telusko@navin-mac course % javac Hello.java
Hello.java:5: error: incompatible types: possible lossy convers
ion from int to byte
        byte b = 257;
                 ^
1 error
telusko@navin-mac course % java Hello.java
127
telusko@navin-mac course % java Hello.java
125
telusko@navin-mac course % java Hello.java
Hello.java:7: error: incompatible types: possible lossy convers
ion from int to byte
        byte k = a;
                 ^
1 error
error: compilation failed
telusko@navin-mac course % java
12
telusko@navin-mac course % java
1
telusko@navin-mac course % 
```

Here 257 is out of range so printing 257%256



```java
public class Project {

    public static void main(String[] args) {

        String day = "Sunday";
        String result = "";

        result = switch(day)
        {
            case "Saturday", "Sunday" -> "6am";
            case "Monday" -> "8am";
            default -> "7am";
        };

        System.out.println(result);

```

```
/Library/Java/JavaVirtualMachines/jdk-17.0.3.1.jdk/Contents/Home/bin/java -javaagent:/Applications/Inte .app/Contents/l
6am

Process finished with exit code 0
```



```java
public class Project {

    public static void main(String[] args) {

        String day = "Wed";
        String result = "";

        result = switch(day)
        {
            case "Saturday", "Sunday" : yield "6am";
            case "Monday" : yield "8am";
            default : yield  "7am";
        };

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

```
/Library/Java/JavaVirtualMachines/jdk-17.0.3.1.jdk/Contents/Home/bin/java -javaagent:/Applications/Inte E.app/Contents/l
7am

Process finished with exit code 0
```

Build completed successfully in 544 ms (a minute ago)

>>

Car c = new Car(); Memory management in sequential order

1. **Class Loading**:

- **Check if Class is Loaded**: The JVM checks if the `Car` class is already loaded. If not, the class loader loads it into memory.

- **Class Verification**: The JVM verifies the bytecode of the `Car` class for correctness and security.

- **Class Preparation**: The JVM allocates memory for static variables of the `Car` class and initializes them to default values.

- **Class Resolution**: The JVM resolves symbolic references in the `Car` class to actual references.

2. **Reference Variable Creation**:

- **Reference Variable Creation**: A reference variable `c` of type `Car` is declared. If `c` is a local variable, it is created on the stack. If it is an instance variable, it is part of the memory layout of the containing object, which resides on the heap.

3. **Object Creation**:

- **Memory Allocation**: The JVM allocates memory for a new `Car` object on the heap. The size of the allocated memory depends on the fields defined in the `Car` class.

- **Default Initialization**: The memory allocated for the `Car` object is initialized to default values (e.g., `0` for numeric types, `null` for reference types, `false` for boolean).

- **Constructor Call**: The constructor of the `Car` class is called, which may further initialize the fields of the `Car` object with specific values.

4. **Reference Assignment**:

- **Assign Memory Address**: The memory address of the newly created `Car` object is assigned to the reference variable `c`.

5. **Garbage Collection**:

- **Reachability Check**: The `Car` object remains in memory as long as it is reachable through the reference variable `c` or any other references.

- **Object Eligibility**: When the reference variable `c` goes out of scope or is assigned to another object or `null`, the `Car` object becomes eligible for garbage collection.

- **Garbage Collection Process**: The garbage collector eventually reclaims the memory used by the `Car` object, making it available for future allocations.

So the correct order is:

1. Class loading (if needed)

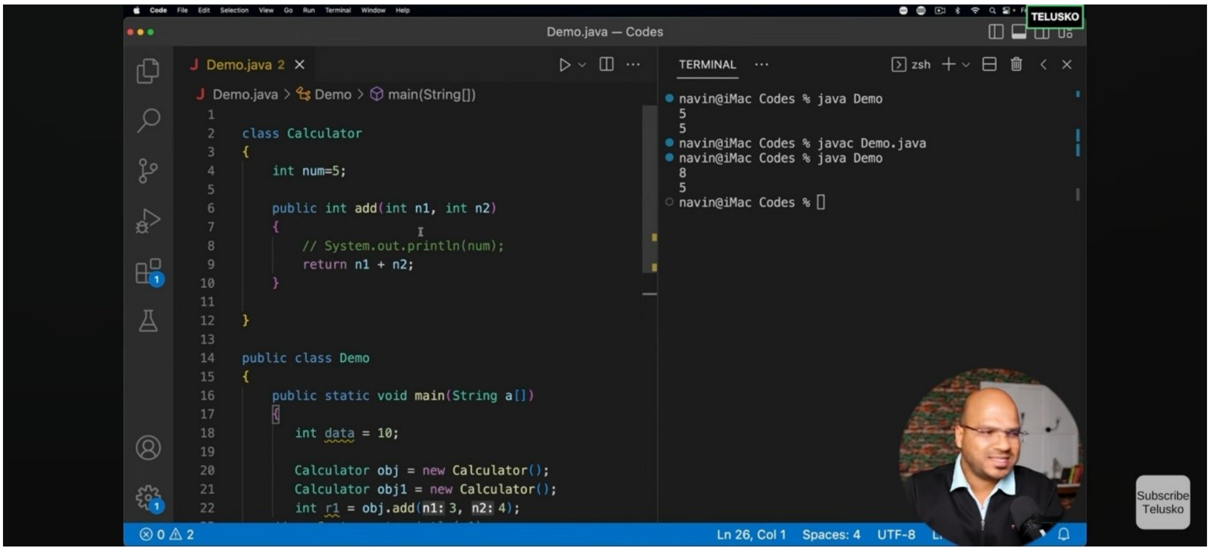
2. Reference variable creation (`c` on the stack)

3. Object creation (memory allocation on the heap)

4. Reference assignment (heap address assigned to `c`)

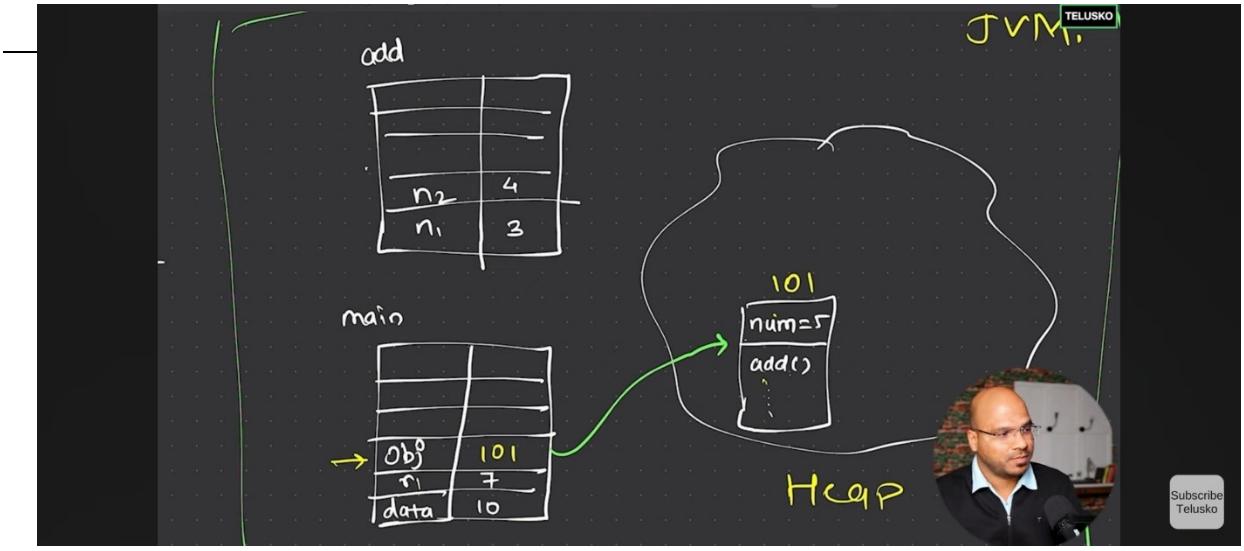
5. Garbage collection (if applicable)

This revised sequence correctly reflects how the JVM manages memory when the statement `Car c = new Car();` is executed.



```

class Calculator
{
    int num=5;

    public int add(int n1, int n2)
    {
        // System.out.println(num);
        return n1 + n2;
    }

}

public class Demo
{
    public static void main(String a[])
    {
        int data = 10;

        Calculator obj = new Calculator();
        Calculator obj1 = new Calculator();
        int r1 = obj.add(n1: 3, n2: 4);
```

```
navin@iMac Codes % java Demo
5
5
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
8
5
navin@iMac Codes % 
```



>>How Main method works: In Java, the main class (the class containing the `main` method) serves as the entry point for the application. When you run a Java application, the Java Virtual Machine (JVM) starts executing the `main` method without explicitly creating an instance of the main class. Here's how memory management works in this context:

1. **Static Context**: The `main` method is `static`, meaning it belongs to the class itself rather than an instance of the class. Static methods and variables are stored in a special area of memory known as the **method area** (part of the heap in many JVM implementations).

2. **Method Area**: The method area stores class-level data, including static variables, method definitions (including the `main` method), and other class-level structures. When the JVM loads the main class, it allocates memory for these static members in the method area.

3. **Stack Memory**: When the JVM starts executing the `main` method, it creates a stack frame for it in the **stack memory**. The stack is used for method execution, including local variables and method call management. Each thread has its own stack.

4. **Heap Memory**: If the `main` method or any other part of the program creates objects using the `new` keyword, these objects are allocated memory in the **heap**. The heap is shared among all threads and is managed by the JVM's garbage collector, which reclaims memory from objects that are no longer reachable.

To summarize, the `main` method and any other static members of the main class reside in the method area, while objects created during the execution of the program are stored in the heap. Local variables within the `main` method (or any other method) are stored in the stack. This division of memory areas helps in efficient memory management and execution of Java programs.

>>

Instances Variables vs Other Variables:

Instance variables in Java differ from other types of variables in several key ways:

1. **Scope and Lifetime**:

- **Instance Variables**: These are defined within a class but outside any method, constructor, or block. They are created when an object of the class is instantiated and destroyed when the object is destroyed. Each object has its own copy of instance variables.

- **Local Variables**: These are defined within a method, constructor, or block. They are created when the method, constructor, or block is entered and destroyed once it exits. They are not accessible outside their defining method, constructor, or block.

- **Static Variables**: Also known as class variables, these are defined within a class using the `static` keyword but outside any method, constructor, or block. There is only one copy of a static variable, regardless of how many objects of the class are created. They are associated with the class itself rather than any specific instance.

2. **Memory Location**:

- **Instance Variables**: Stored in the heap, as part of the object.

- **Local Variables**: Stored in the stack, as part of the stack frame for the method, constructor, or block in which they are defined.

   - **Static Variables**: Stored in the method area (part of the heap in many JVM implementations), shared among all instances of the class.

3. **Default Values**:

   - **Instance Variables**: Automatically initialized to default values (e.g., `0` for integers, `null` for object references) if not explicitly initialized.

   - **Local Variables**: Must be explicitly initialized before use; they do not get default values.

   - **Static Variables**: Automatically initialized to default values if not explicitly initialized.

4. **Access Modifiers**:

   - **Instance Variables**: Can be marked with access modifiers (`private`, `protected`, `public`) to control their visibility.

   - **Local Variables**: Cannot have access modifiers; their scope is limited to the method, constructor, or block in which they are declared.

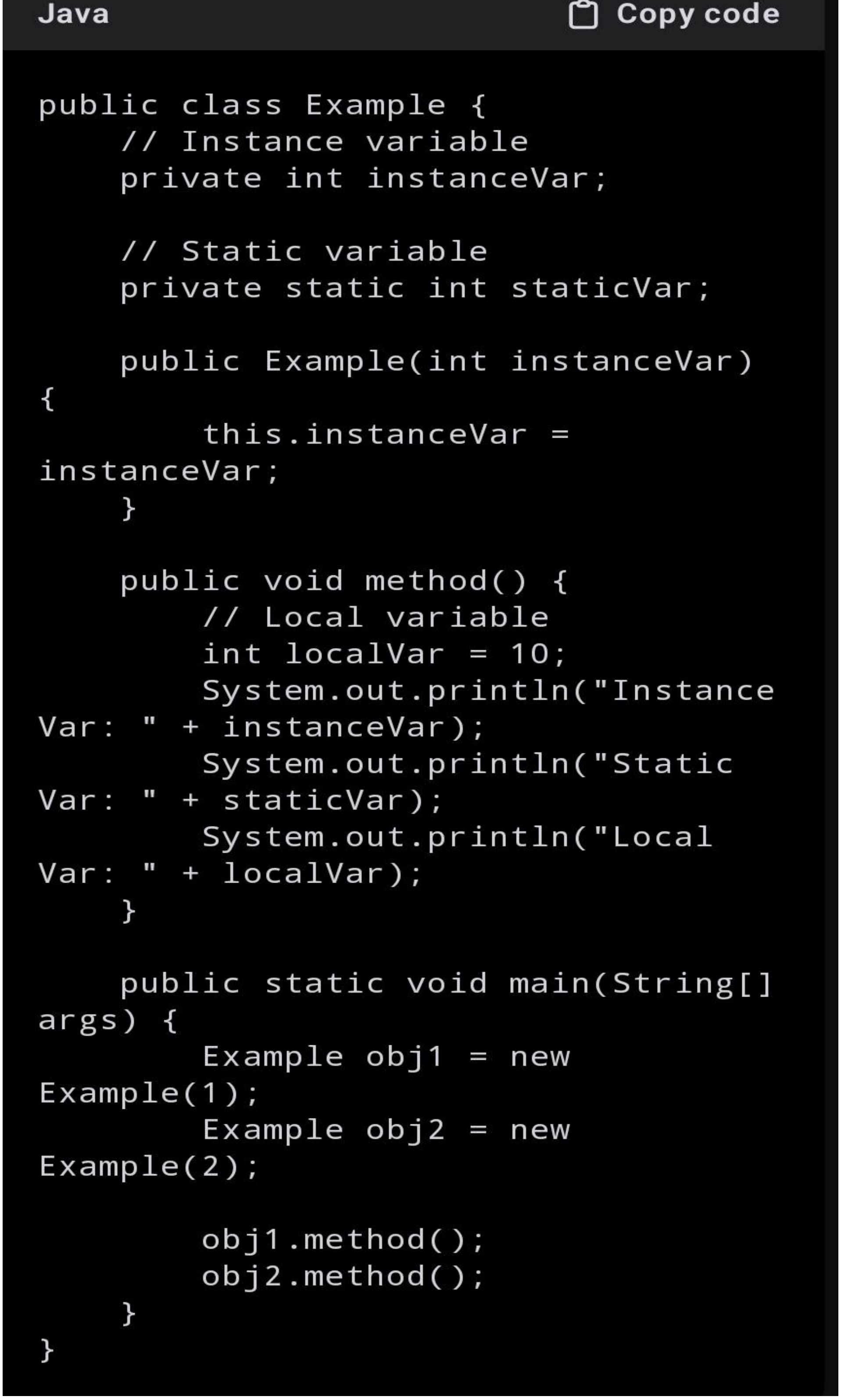
   - **Static Variables**: Can also be marked with access modifiers and are accessed using the class name.

5. **Use Case**:

   - **Instance Variables**: Hold the state of an object. Different objects can have different values for their instance variables.

   - **Local Variables**: Used for temporary storage of data within a method, constructor, or block.

   - **Static Variables**: Used for values shared among all instances of a class, such as constants or counters. Here is a quick example to illustrate the differences:



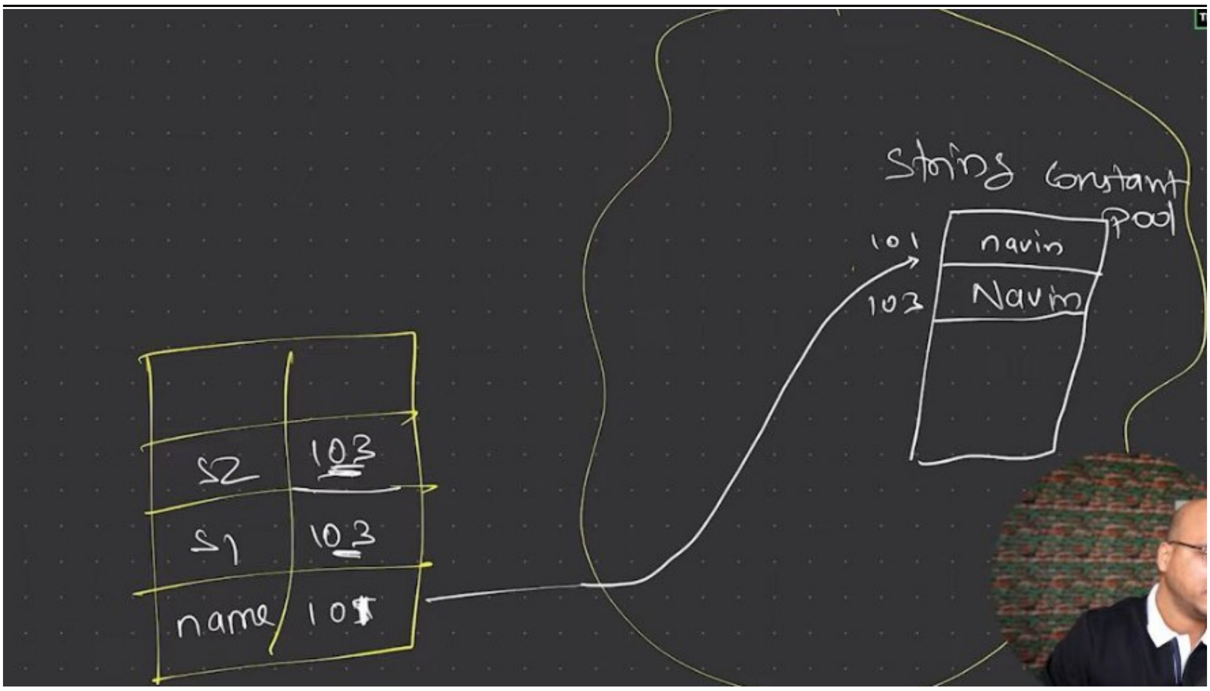
```
public class Example {
    // Instance variable
    private int instanceVar;

    // Static variable
    private static int staticVar;

    public Example(int instanceVar)
{
        this.instanceVar =
instanceVar;
    }

    public void method() {
        // Local variable
        int localVar = 10;
        System.out.println("Instance
Var: " + instanceVar);
        System.out.println("Static
Var: " + staticVar);
        System.out.println("Local
Var: " + localVar);
    }

    public static void main(String[]
args) {
        Example obj1 = new
Example(1);
        Example obj2 = new
Example(2);

        obj1.method();
        obj2.method();
    }
}
```

In this example:

- `instanceVar` is an instance variable, so each `Example` object has its own copy.

- `staticVar` is a static variable, shared among all `Example` objects.

- `localVar` is a local variable, accessible only within the `method`.



>>Class.forName("<classname>") is to load the class

>>The static keyword in Java is used to define class-level variables and methods, meaning they belong to the class itself rather than any specific instance of the class. This has multiple uses and benefits. First, static variables are shared among all instances of a class, ensuring there is only one copy of the variable, which helps in saving memory and maintaining consistency across all instances. For example, a static counter variable can track the number of objects created, accessible and modifiable by all instances. Second, static methods can be called directly using the class name without needing to create an object. This is particularly useful for utility or helper methods that perform tasks not dependent on instance-specific data, such as mathematical calculations or configuration settings. Additionally, static blocks allow for static initialization, which runs once when the class is loaded, enabling the setup of static variables. Overall, the static keyword enhances memory efficiency, simplifies access to common methods and variables, and ensures consistent state management across all instances of a class.

>>The heap is divided into generations (Young Generation, Old Generation) to optimize garbage collection.The Young Generation is collected frequently because it contains many short-lived objects.The Old Generation is collected less frequently but can still undergo garbage collection when it fills up.

>>The method area in the Java Virtual Machine (JVM) is a part of the JVM's memory where class-level structures are stored. It serves several key purposes in managing Java programs:

1. Class Metadata:
   - When a class is loaded by the JVM, its metadata (information about the class such as its name, modifiers, superclass, interfaces, etc.) is stored in the method area.

2. Static Fields:
   - Static variables declared within a class are stored in the method area. These variables are shared among all instances of the class.

3. Method Data:
   - The method area contains information about methods and constructors of classes, including bytecode (compiled code of methods).

4. Constant Pool:
   - Each class loaded by the JVM has a runtime constant pool, which is part of the method area. This pool contains constants, literals, and symbolic references used within the class.

5. Memory Management:
   - The method area is managed by the JVM's memory manager, and while it supports garbage collection, it typically collects less frequently compared to the heap.

6. Shared among Threads:
   - The method area is shared among all threads running within the JVM. This shared access ensures consistency in class data and method execution across the application.

Overall, the method area plays a crucial role in storing and managing class-related information and static variables, supporting the execution and management of Java programs within the JVM.

>>this keyword is a reference to the current instance of the class in which it is used. this reference is not created in memory in the same way that objects and variables are. Instead, it is a logical construct provided by the Java compiler that exists as part of the method or constructor's context. When an object is created using the new keyword, the object itself is allocated in the heap memory. The **this** reference is implicitly passed to instance methods and constructors as a hidden parameter.

>>If you call a method from a constructor that is overridden in a subclass, the method from the subclass will be called.

Reason:

Dynamic Method Dispatch:

In Java, non-static methods are by default virtual, meaning the method that gets executed is determined at runtime based on the actual object's type, not the reference type. This is called dynamic method dispatch or late binding.When an object is being created, the constructor of the class and its parent classes are executed in a specific order. If a method call occurs in a constructor, the virtual method mechanism looks up the method in the runtime type of the object being created, not the type where the constructor is defined.

Object Creation Process:

When a subclass object is created, Java first initializes the superclass part of the object by calling the superclass constructor. If the superclass constructor calls an overridden method, Java calls the overridden method in the subclass

---

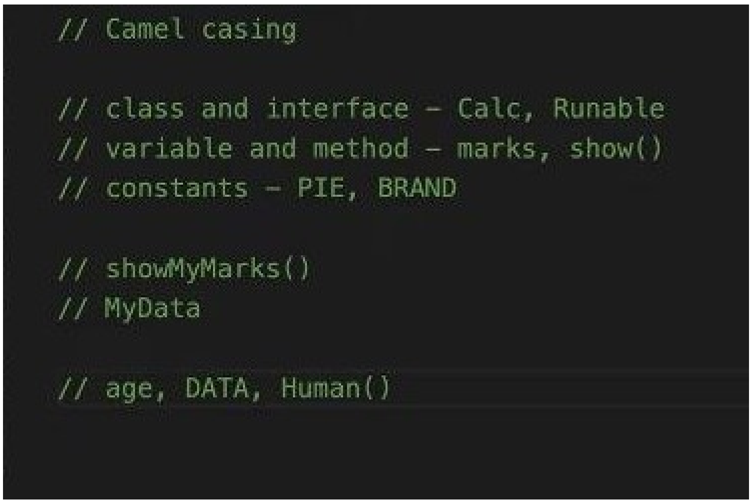
>>In Java, when you create an object of a subclass, the constructor of the superclass is automatically called. This happens because of the way object initialization works in Java, ensuring that the entire object is fully initialized, including the parts inherited from any superclasses. Here’s a detailed explanation:

Implicit Constructor Call:

When you instantiate a subclass, Java implicitly calls the constructor of its superclass before executing the subclass constructor.This ensures that the inherited fields and methods from the superclass are properly initialized before the subclass's own initialization code runs.

Constructor Chaining:

Constructor chaining refers to the process where a constructor calls another constructor, either in the same class or in the superclass. In Java, if a constructor does not explicitly call another constructor using this() (for the same class) or super() (for the superclass), the compiler automatically inserts a call to the no-argument constructor of the superclass as the first line of the constructor.



```
// Camel casing

// class and interface - Calc, Runable
// variable and method - marks, show()
// constants - PIE, BRAND

// showMyMarks()
// MyData

// age, DATA, Human()
```

>>

Advantages of Using List Reference while creating a ArrayList object:

Flexibility:

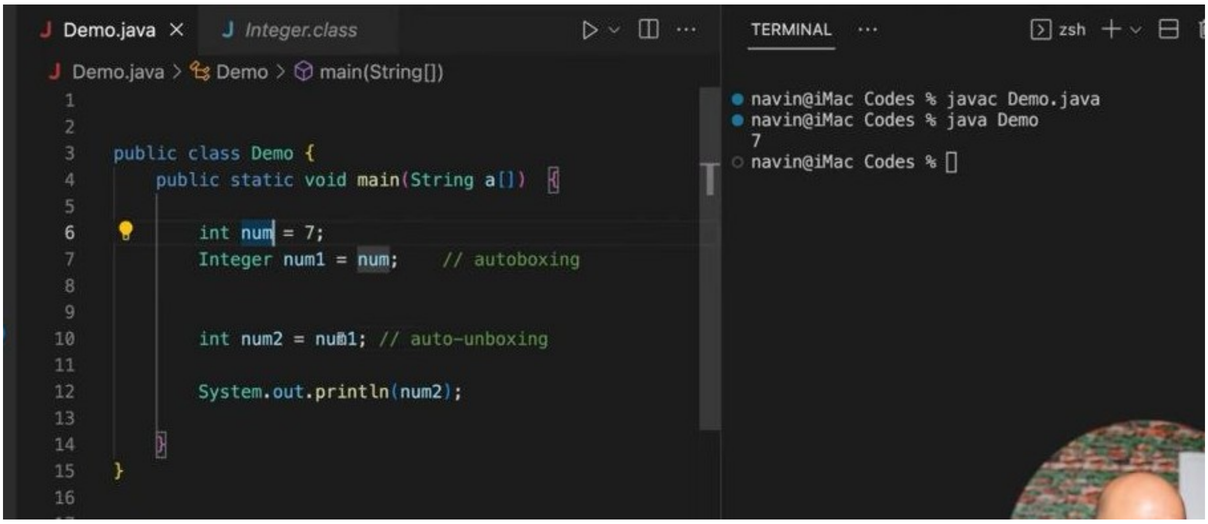
If you use a List reference, you can easily change the implementation to another List type (e.g., LinkedList) without changing the code that uses the List.List<String> list = new LinkedList<>();

Abstraction:

Using interfaces promotes abstraction, making your code more modular and easier to maintain. You can work with higher-level concepts (like a list) rather than specific implementations.

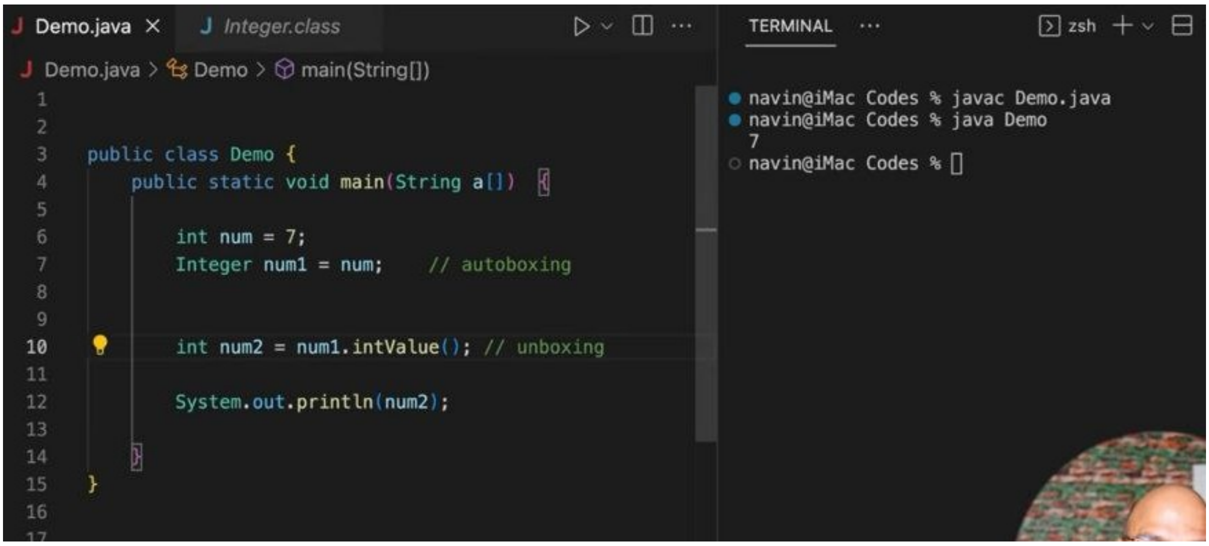
Code Reusability:Code that uses the List interface can work with any implementation of List, increasing code reusability and reducing coupling.

Easier Testing: Using interfaces makes it easier to substitute different implementations during testing, which can be useful for unit tests.



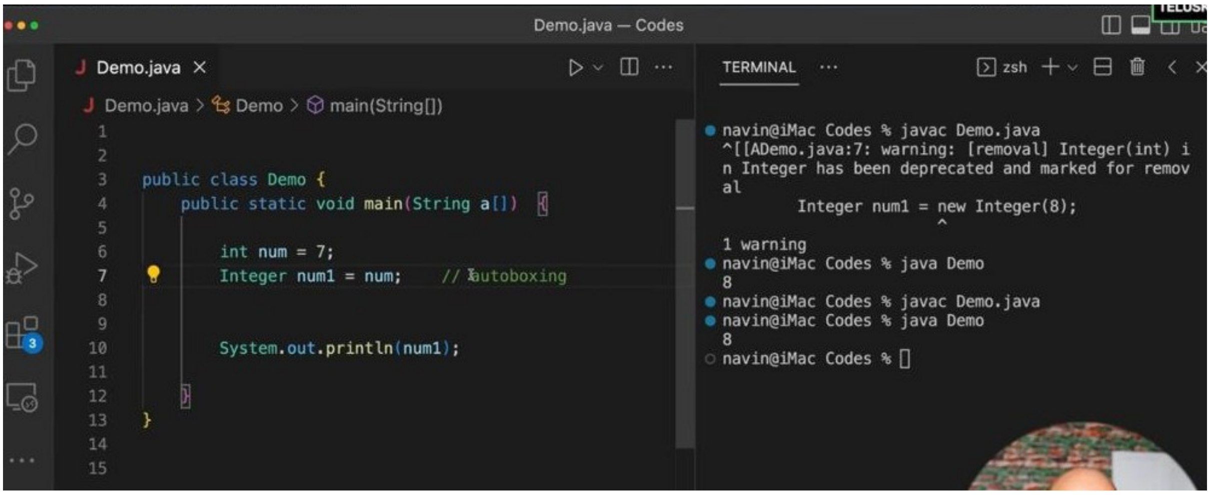
```java


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

        int num = 7;
        Integer num1 = num;     // autoboxing


        int num2 = num1; // auto-unboxing

        System.out.println(num2);

    }
}
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
7
navin@iMac Codes %
```



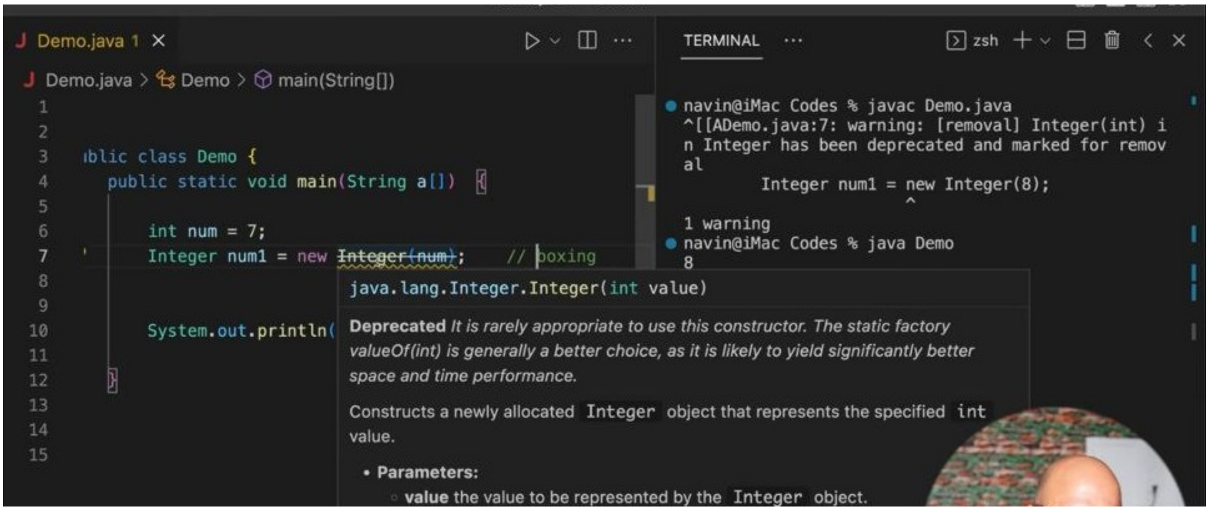
```java
public class Demo {
    public static void main(String a[]) {

        int num = 7;
        Integer num1 = num;    // autoboxing


        int num2 = num1.intValue(); // unboxing

        System.out.println(num2);

    }
}
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
7
navin@iMac Codes % 
```



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

        int num = 7;
        Integer num1 = num;    // autoboxing


        System.out.println(num1);

    }
}
```

```
navin@iMac Codes % javac Demo.java
^[[ADemo.java:7: warning: [removal] Integer(int) i
n Integer has been deprecated and marked for remov
al
        Integer num1 = new Integer(8);
                       ^
1 warning
navin@iMac Codes % java Demo
8
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
8
navin@iMac Codes % 
```



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

        int num = 7;
        Integer num1 = new ~~Integer(num)~~;    // boxing


        System.out.println(

    }
```

`java.lang.Integer.Integer(int value)`

Deprecated *It is rarely appropriate to use this constructor. The static factory valueOf(int) is generally a better choice, as it is likely to yield significantly better space and time performance.*

Constructs a newly allocated `Integer` object that represents the specified `int` value.

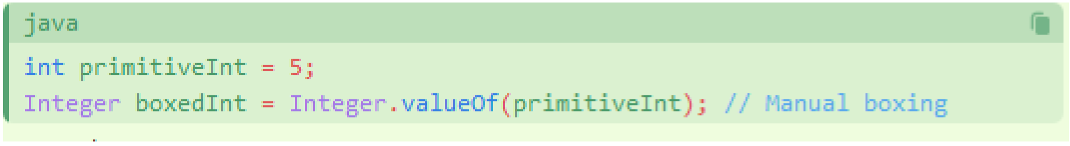
- **Parameters:**
  - **value** the value to be represented by the `Integer` object.

```
navin@iMac Codes % javac Demo.java
^[[ADemo.java:7: warning: [removal] Integer(int) i
n Integer has been deprecated and marked for remov
al
        Integer num1 = new Integer(8);
                       ^
1 warning
navin@iMac Codes % java Demo
8
```

Boxing

Boxing is the process of converting a primitive type into its corresponding wrapper class object. This is done manually before Java 5.

Example:

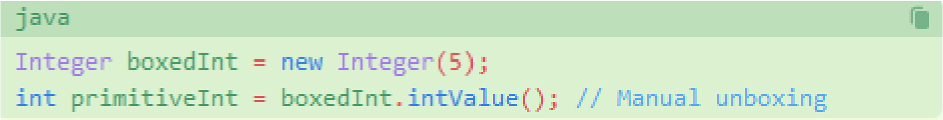


```java
int primitiveInt = 5;
Integer boxedInt = Integer.valueOf(primitiveInt); // Manual boxing
```

Unboxing

Unboxing is the reverse process, where a wrapper class object is converted back to its corresponding primitive type.

Example:

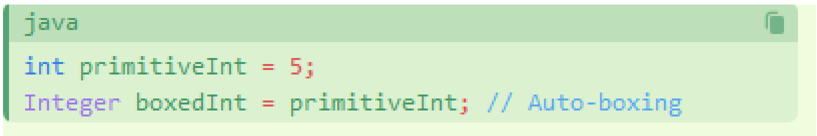


```java
Integer boxedInt = new Integer(5);
int primitiveInt = boxedInt.intValue(); // Manual unboxing
```

Auto-boxing

Auto-boxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. This feature was introduced in Java 5.

Example:

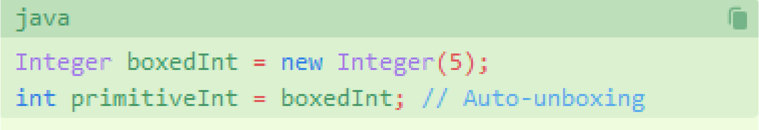


```java
int primitiveInt = 5;
Integer boxedInt = primitiveInt; // Auto-boxing
```

Auto-unboxing

Auto-unboxing is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type.

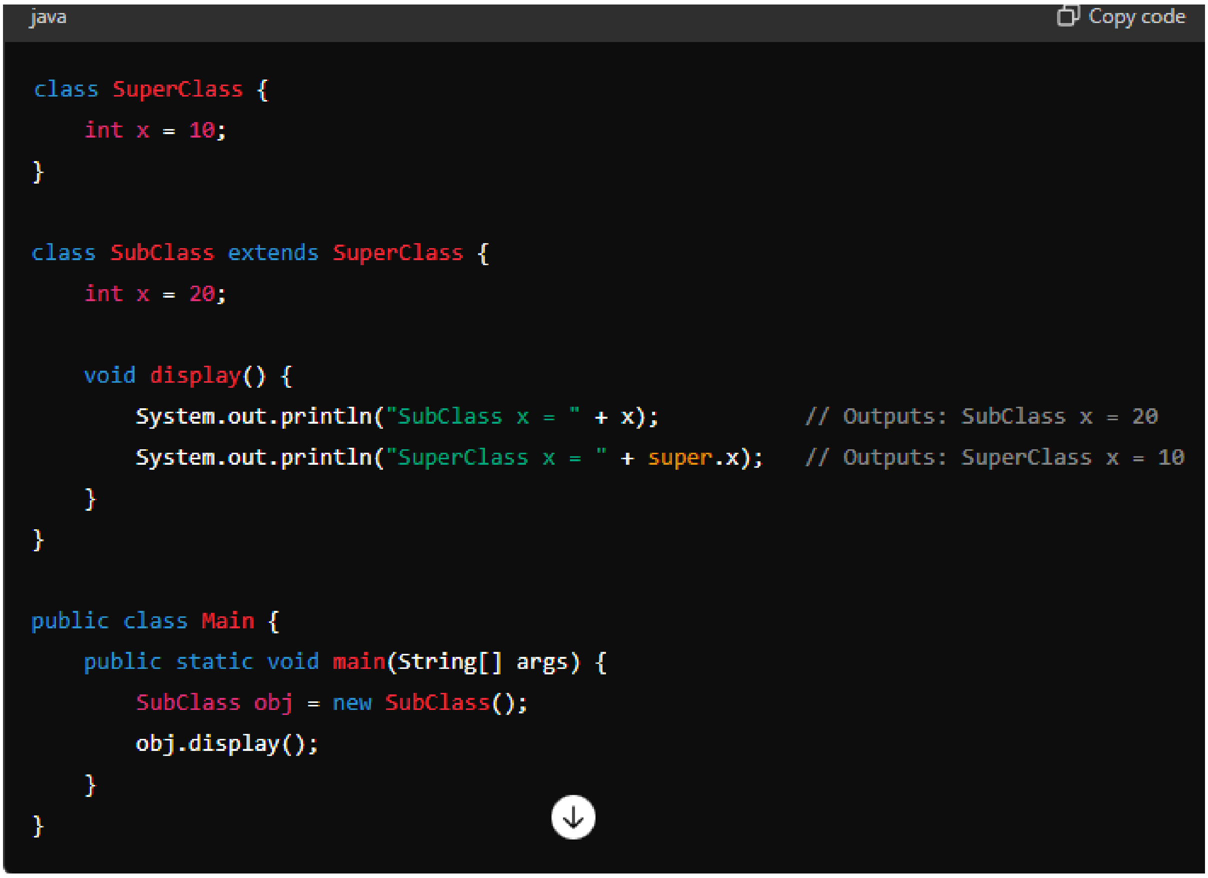
Example:



```java
Integer boxedInt = new Integer(5);
int primitiveInt = boxedInt; // Auto-unboxing
```

>>

In Java, you cannot override a variable, but you can hide it. Variable hiding occurs when a subclass declares a variable with the same name as a variable in its superclass. The variable in the superclass is hidden, not overridden, by the variable in the subclass

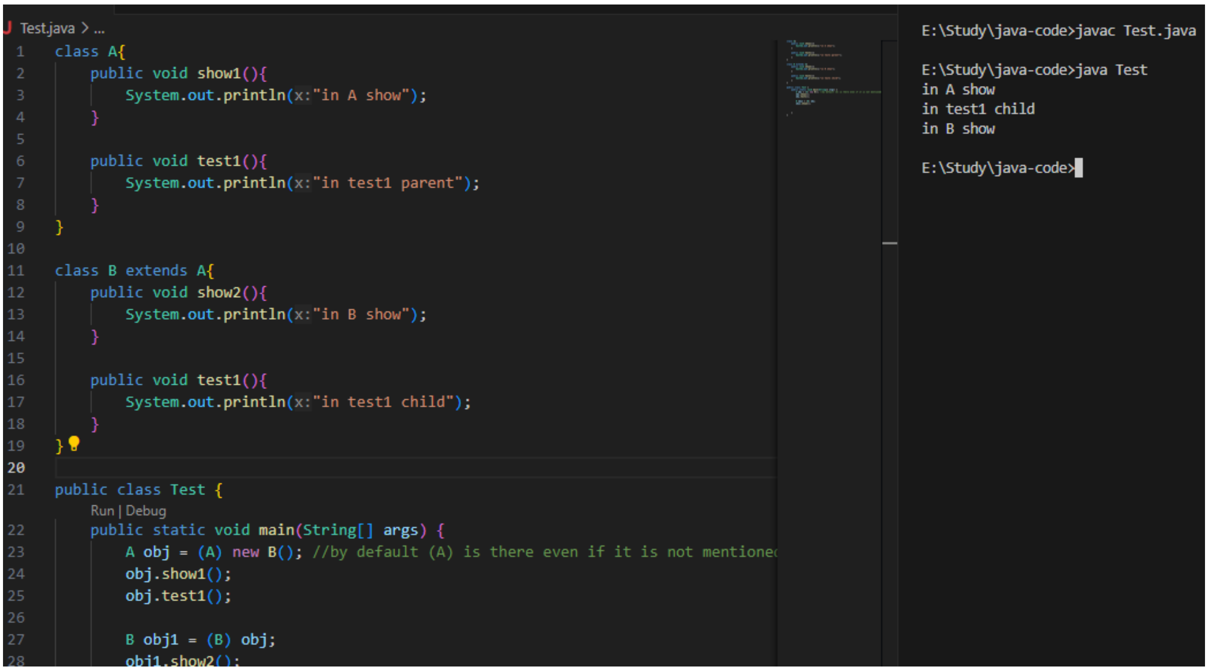


```java
class SuperClass {
    int x = 10;
}

class SubClass extends SuperClass {
    int x = 20;

    void display() {
        System.out.println("SubClass x = " + x);           // Outputs: SubClass x = 20
        System.out.println("SuperClass x = " + super.x);   // Outputs: SuperClass x = 10
    }
}

public class Main {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        obj.display();
    }
}
```

>>concrete class is the child class of abstract class which have all the implementation of abstract methods of abstract class

---

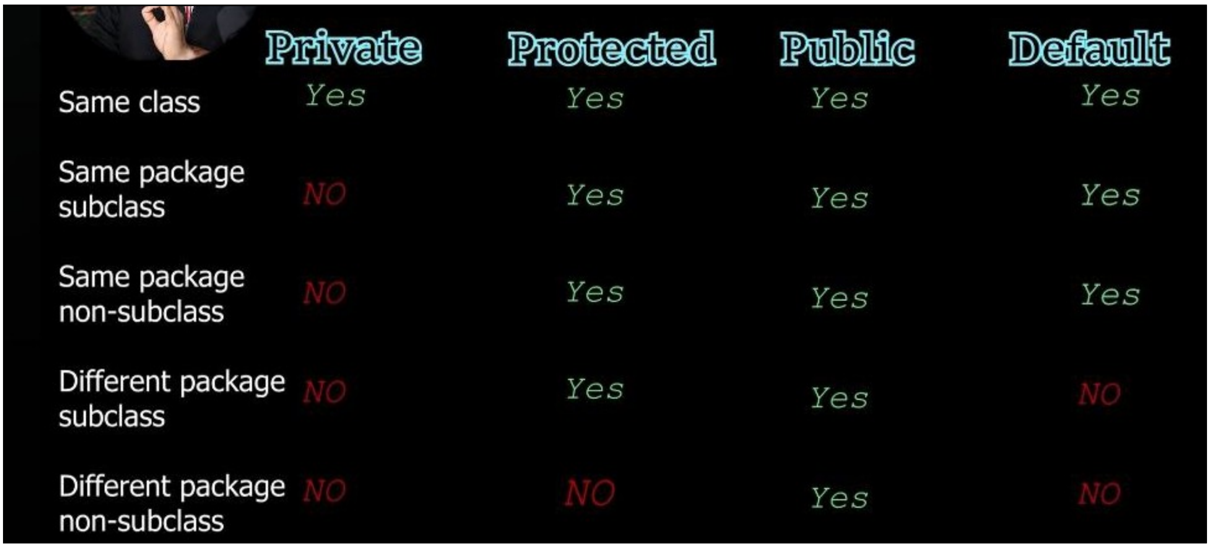
>> more on casting



Test.java > ...

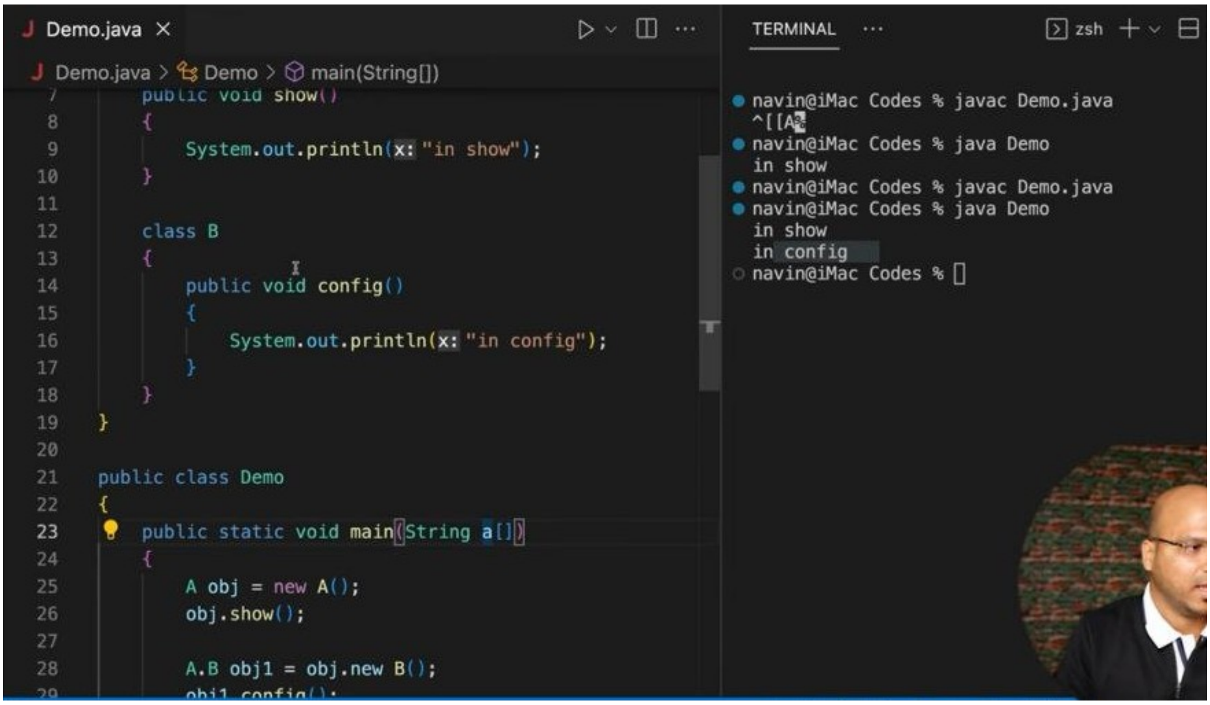
```java
class A{
    public void show1(){
        System.out.println(x:"in A show");
    }

    public void test1(){
        System.out.println(x:"in test1 parent");
    }
}

class B extends A{
    public void show2(){
        System.out.println(x:"in B show");
    }

    public void test1(){
        System.out.println(x:"in test1 child");
    }
}

public class Test {
    Run | Debug
    public static void main(String[] args) {
        A obj = (A) new B(); //by default (A) is there even if it is not mentione
        obj.show1();
        obj.test1();

        B obj1 = (B) obj;
        obj1.show2();
```

```
E:\Study\java-code>javac Test.java

E:\Study\java-code>java Test
in A show
in test1 child
in B show

E:\Study\java-code>
```



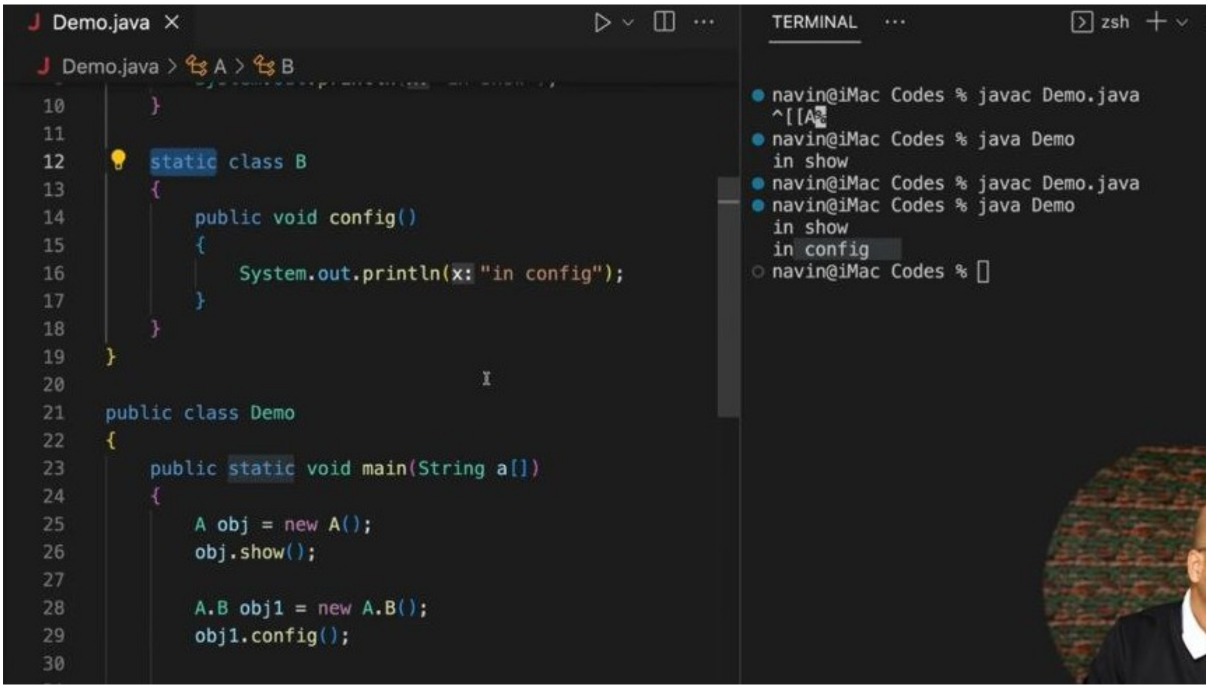
	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package non-subclass	NO	NO	Yes	NO

>>Inner Class



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

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

public class Demo
{
    public static void main(String a[])
    {
        A obj = new A();
        obj.show();

        A.B obj1 = obj.new B();
        obj1.config();
```

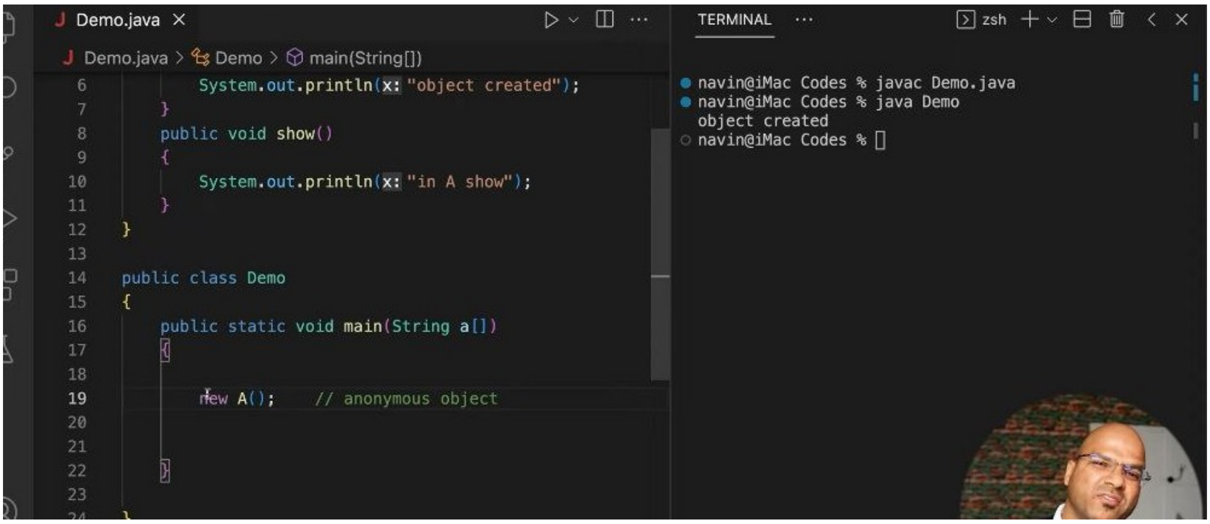
```
navin@iMac Codes % javac Demo.java
^[[A
navin@iMac Codes % java Demo
in show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in show
in config
navin@iMac Codes %
```



```
    }

    static class B
    {
        public void config()
        {
            System.out.println(x: "in config");
        }
    }
}

public class Demo
{
    public static void main(String a[])
    {
        A obj = new A();
        obj.show();

        A.B obj1 = new A.B();
        obj1.config();

```

```
navin@iMac Codes % javac Demo.java
^[[A
navin@iMac Codes % java Demo
in show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in show
in config
navin@iMac Codes %
```

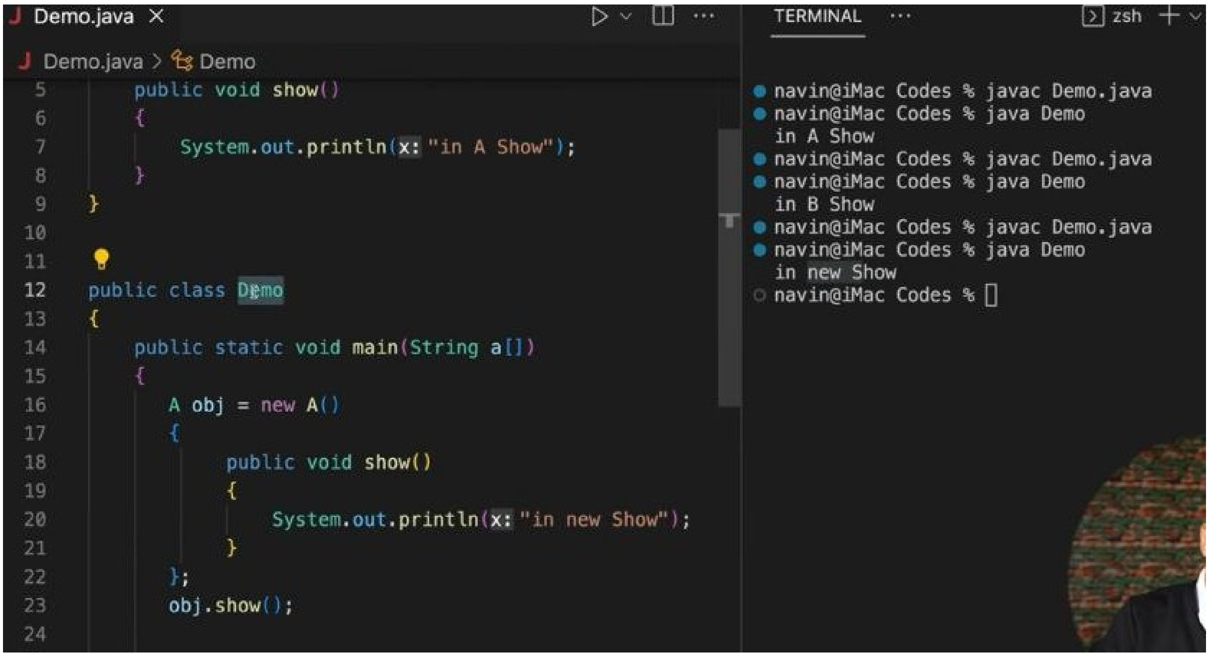
>>**Anonymous** object



```
        System.out.println(x: "object created");
    }
    public void show()
    {
        System.out.println(x: "in A show");
    }
}

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

        new A();     // anonymous object


    }

}
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
object created
navin@iMac Codes % 
```

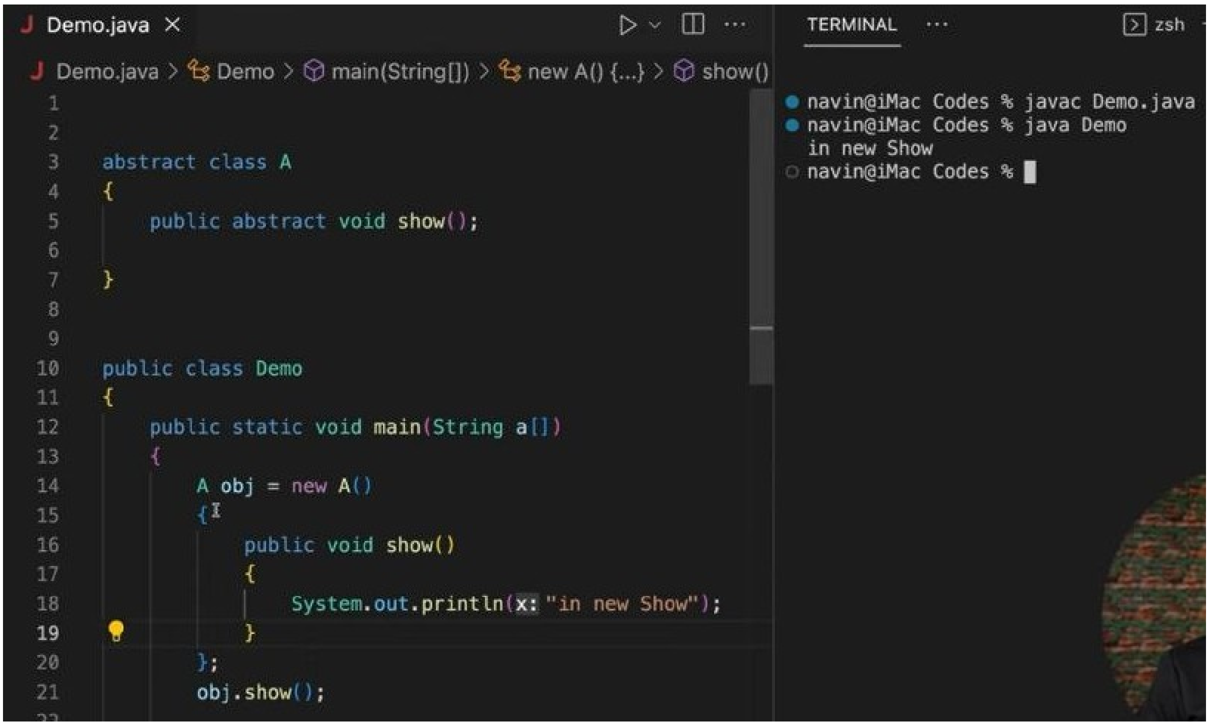
>>anonymous class/anonymous inner class



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


public class Demo
{
    public static void main(String a[])
    {
        A obj = new A()
        {
            public void show()
            {
                System.out.println(x: "in new Show");
            }
        };
        obj.show();

```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in A Show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in B Show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in new Show
navin@iMac Codes % 
```

>>Anonymous inner class for an abstract class



```java


abstract class A
{
    public abstract void show();

}


public class Demo
{
    public static void main(String a[])
    {
        A obj = new A()
        {
            public void show()
            {
                System.out.println(x: "in new Show");
            }
        };
        obj.show();
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in new Show
navin@iMac Codes % 
```

>>interface can have variables. those are by default static and final

---

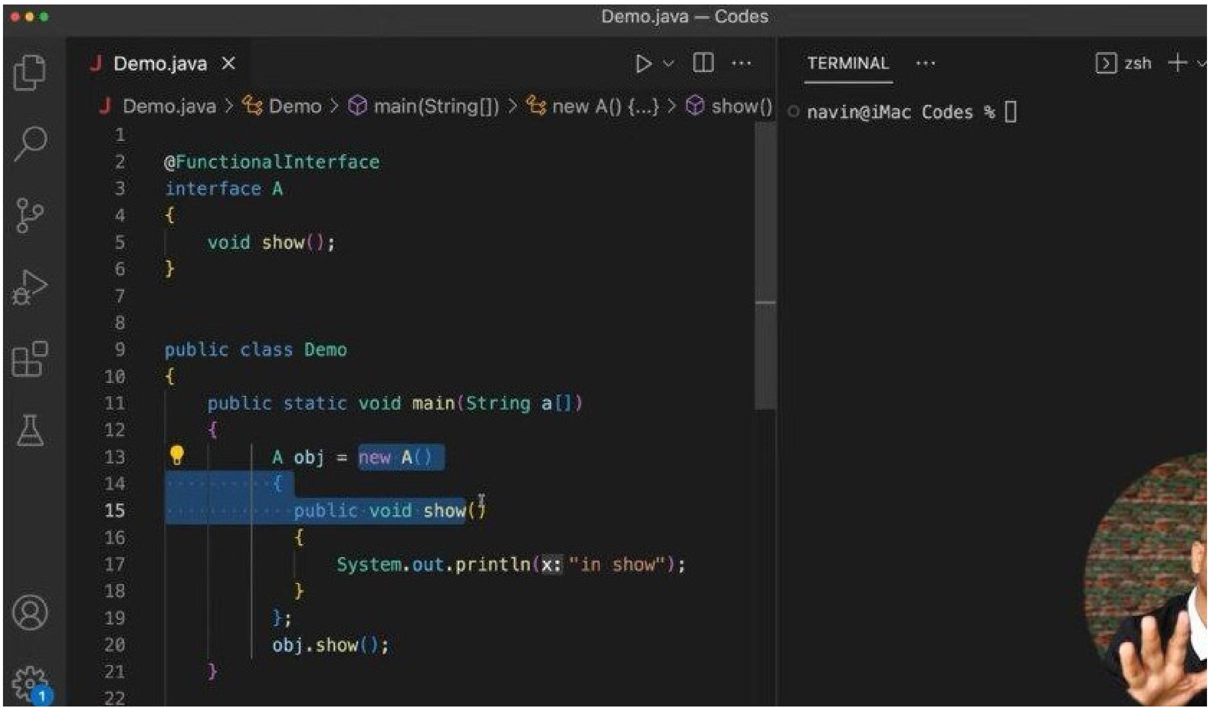
>> If you want to prove that the default constructor is getting invoked without explicitly adding any print statements in the constructor, you can demonstrate it indirectly by observing the initialization behavior of instance variables. Constructor is responsible for initialization of the instance variables

---

>> A functional interface in Java is an interface that contains exactly one abstract method. These interfaces can have any number of default or static methods. Functional interfaces are used as the basis for lambda expressions and method references, which provide a clear and concise way to represent single-method interfaces using an expression

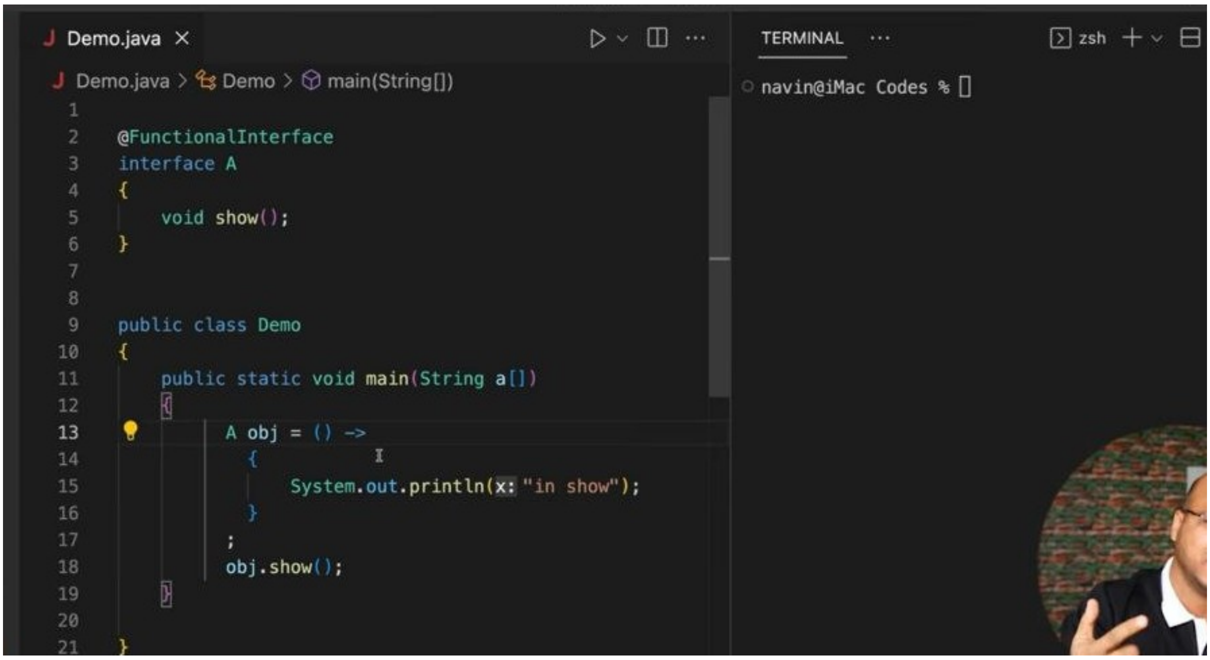
---

>>lambda expression



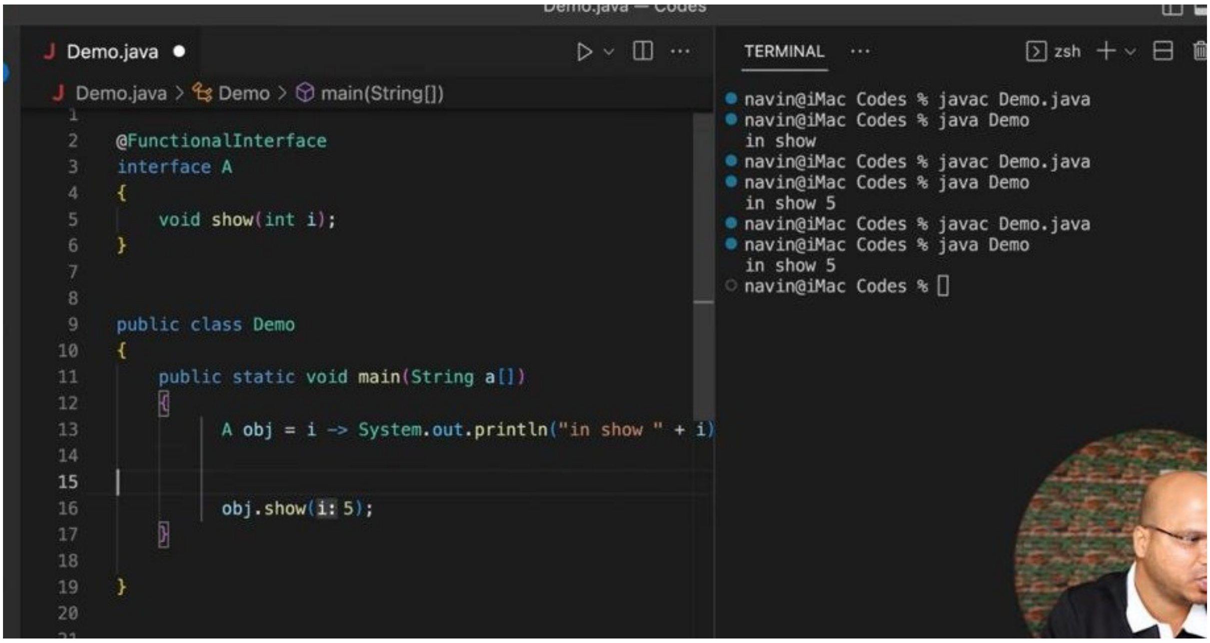
```
@FunctionalInterface
interface A
{
    void show();
}


public class Demo
{
    public static void main(String a[])
    {
        A obj = new A()
        {
            public void show()
            {
                System.out.println(x: "in show");
            }
        };
        obj.show();
    }
```

```
navin@iMac Codes % 
```



```
@FunctionalInterface
interface A
{
    void show();
}


public class Demo
{
    public static void main(String a[])
    {
        A obj = () ->
          {
              System.out.println(x: "in show");
          }
        ;
        obj.show();
    }

}
```

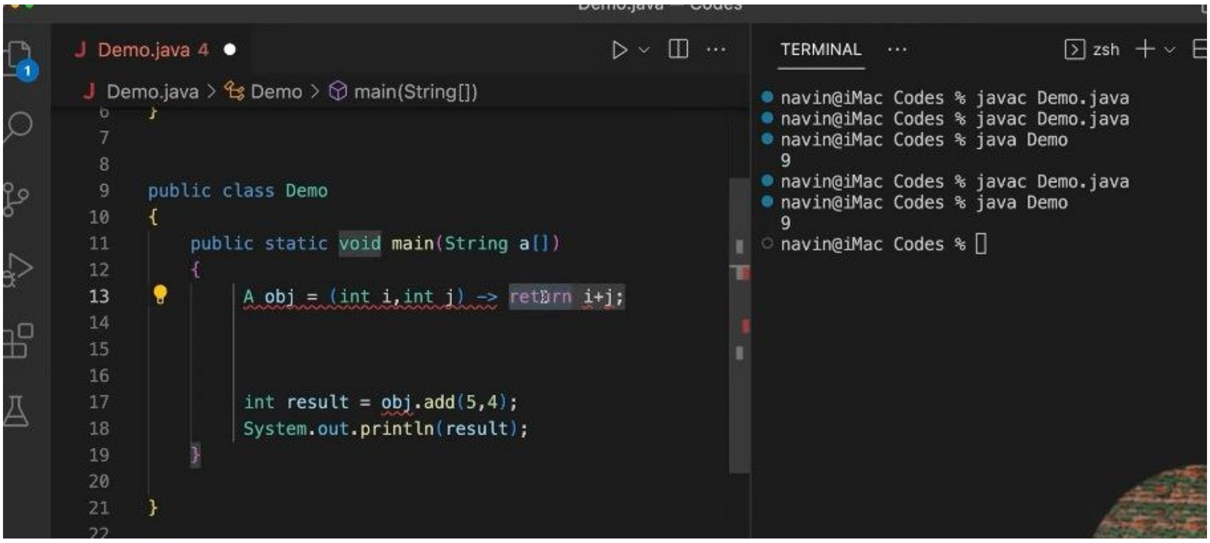
```
navin@iMac Codes % 
```



```java
@FunctionalInterface
interface A
{
    void show(int i);
}


public class Demo
{
    public static void main(String a[])
    {
        A obj = i -> System.out.println("in show " + i)

        obj.show(i: 5);
    }

}
```

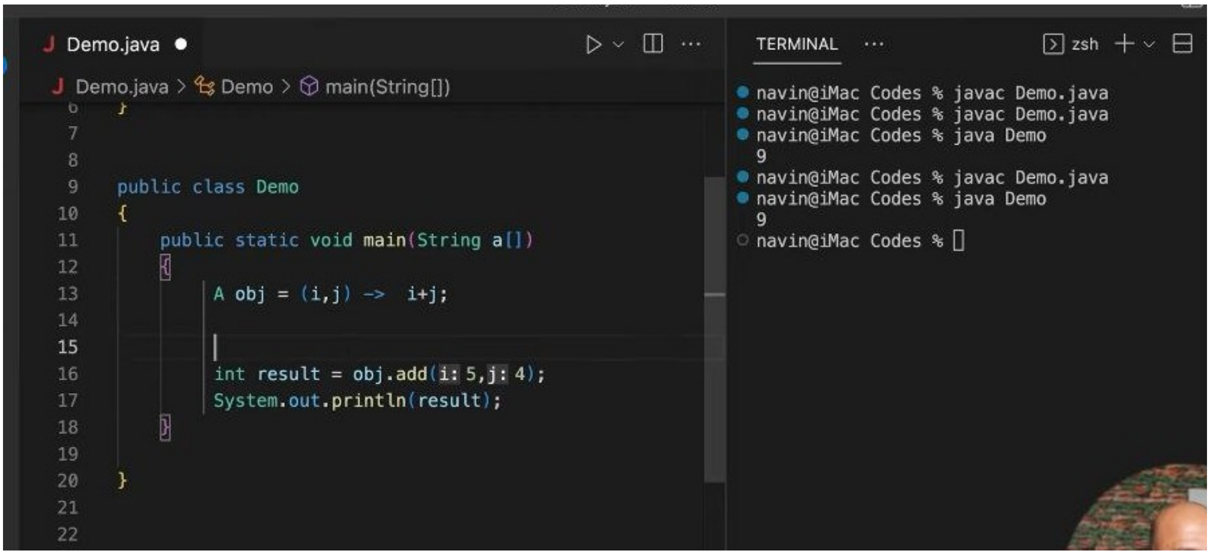
```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in show 5
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in show 5
navin@iMac Codes %
```

only one statement so we don't need to write return



```java
}


public class Demo
{
    public static void main(String a[])
    {
        A obj = (int i,int j) -> return i+j;



        int result = obj.add(5,4);
        System.out.println(result);
    }

}
```

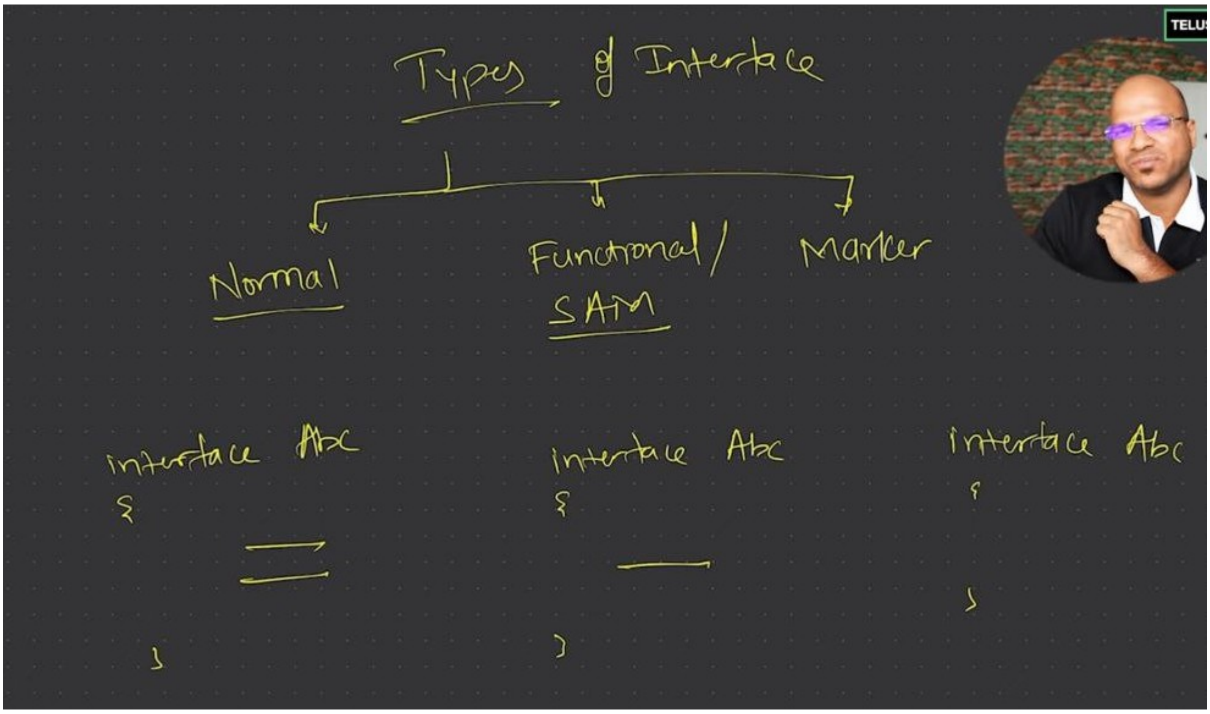
```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
9
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
9
navin@iMac Codes %
```



```java
    }


public class Demo
{
    public static void main(String a[])
    {
        A obj = (i,j) ->  i+j;


        int result = obj.add(i: 5,j: 4);
        System.out.println(result);
    }

}
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
9
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
9
navin@iMac Codes %
```

>>Types of interface



A marker interface in Java is an interface that does not contain any methods or fields. Its primary purpose is to mark or tag a class with a specific capability or property. This tagging can be used by the Java runtime or other frameworks to provide special behavior to the marked classes.

Common examples of marker interfaces in Java include:

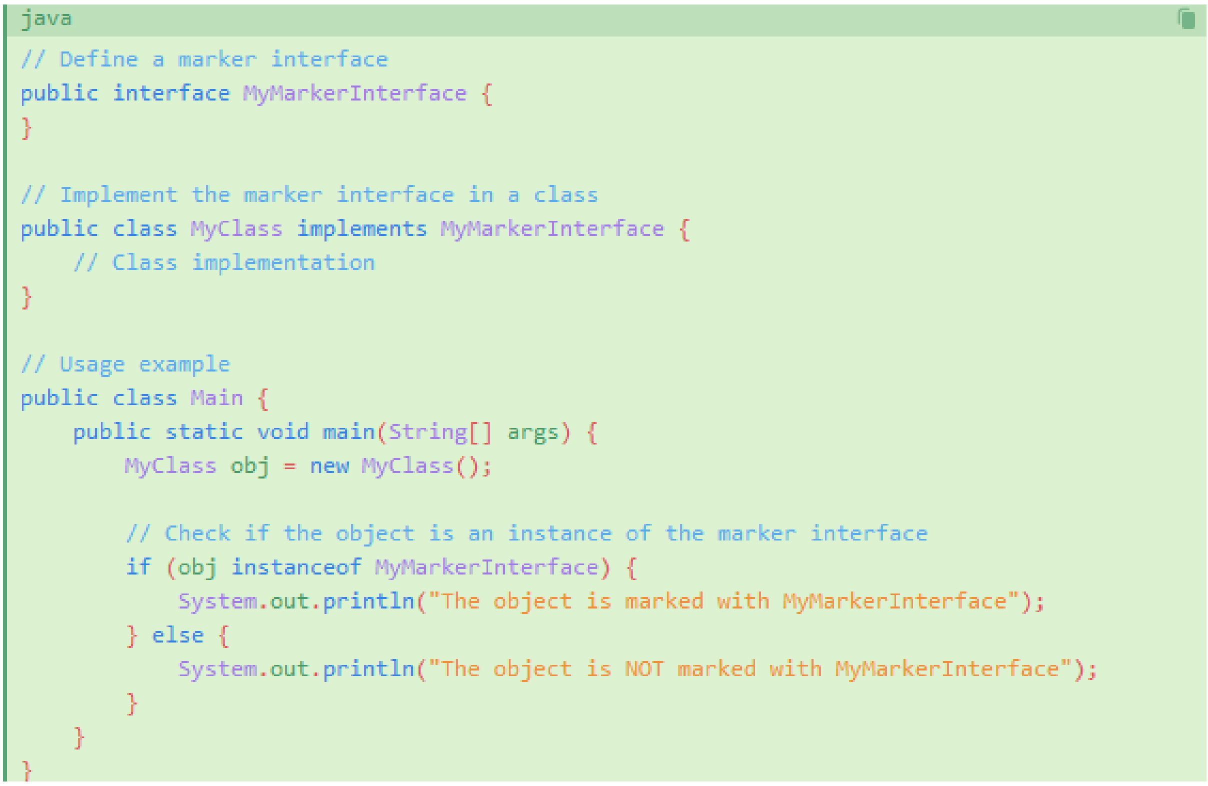
1. Serializable: Used to indicate that a class can be serialized (converted into a byte stream) so that its objects can be saved to a file or transmitted over a network.

2. Cloneable: Indicates that a class allows for a field-for-field copy of instances using the clone method.

3. Remote: Marks a class as a remote object for Java RMI (Remote Method Invocation).

How Marker Interfaces Work

Marker interfaces work by relying on runtime checks or framework-specific logic that looks for the presence of the marker interface to determine if special handling is required. For example, the Java serialization mechanism checks if a class implements Serializable to decide whether it can serialize the object's state.



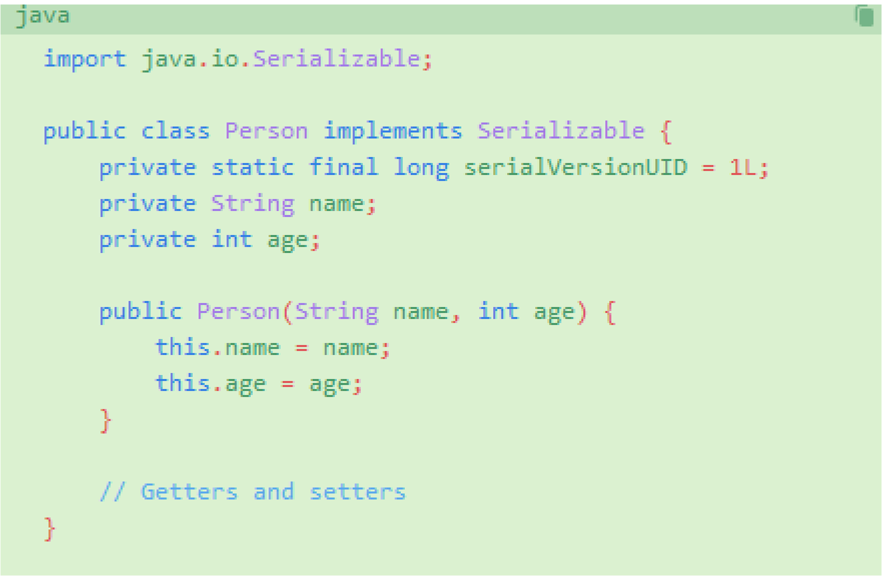
```java
// Define a marker interface
public interface MyMarkerInterface {
}

// Implement the marker interface in a class
public class MyClass implements MyMarkerInterface {
    // Class implementation
}

// Usage example
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Check if the object is an instance of the marker interface
        if (obj instanceof MyMarkerInterface) {
            System.out.println("The object is marked with MyMarkerInterface");
        } else {
            System.out.println("The object is NOT marked with MyMarkerInterface");
        }
    }
}
```

Serializable

- Purpose: Indicates that a class can be serialized, meaning its instances can be converted into a byte stream and later restored.

- Use Case: Object serialization for saving object state to a file, sending objects over a network, or storing them in a database.

- Example:



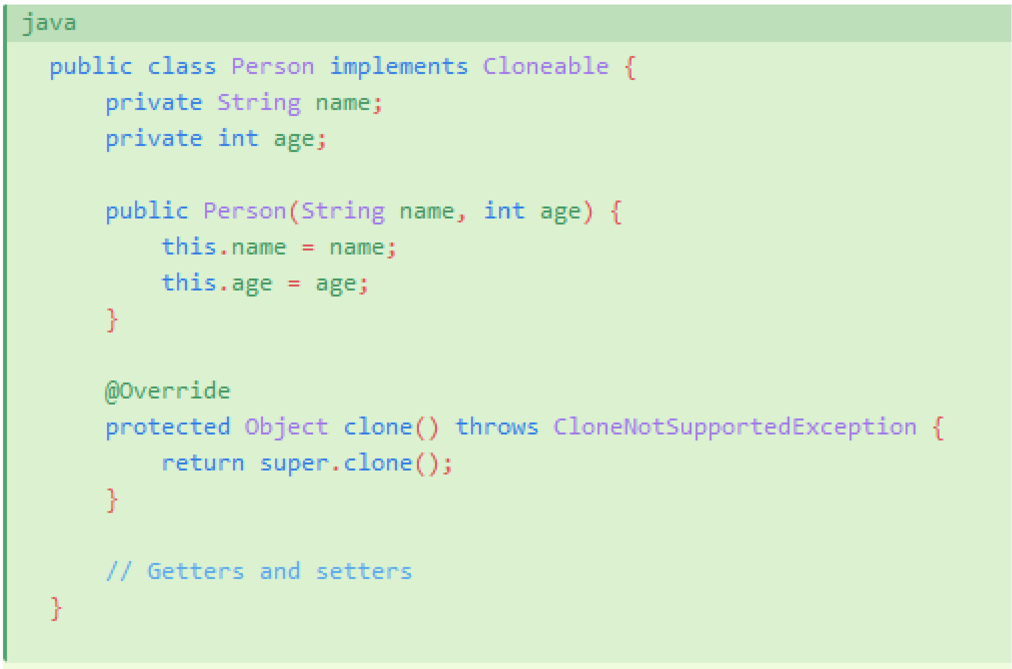
```java
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

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

    // Getters and setters
}
```

Cloneable

- Purpose: Indicates that a class allows for a field-for-field copy of its instances through the clone() method.

- Use Case: Creating a duplicate object with the same state.

- Example:



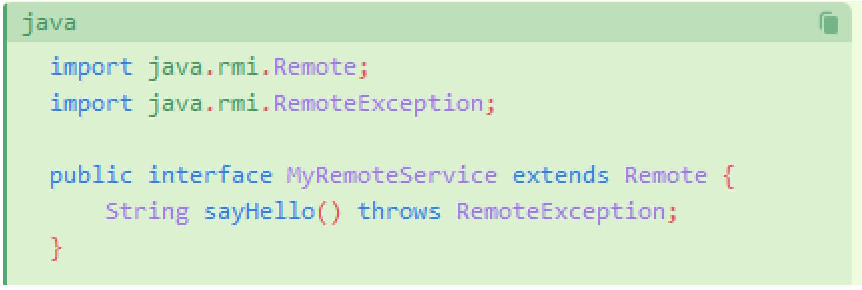
```java
public class Person implements Cloneable {
    private String name;
    private int age;

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

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    // Getters and setters
}
```

Remote

- Purpose: Indicates that a class can be used to create remote objects in Java RMI (Remote Method Invocation).

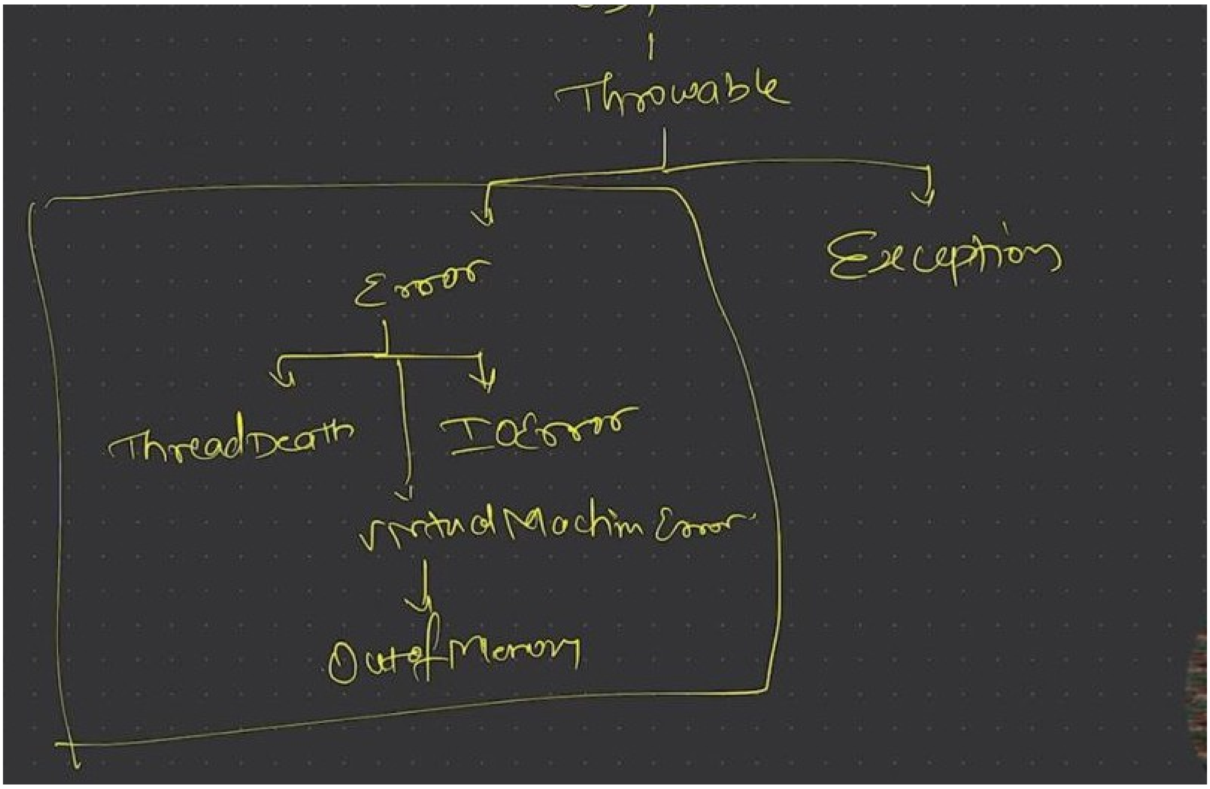
- Use Case: Defining objects whose methods can be called from another JVM.

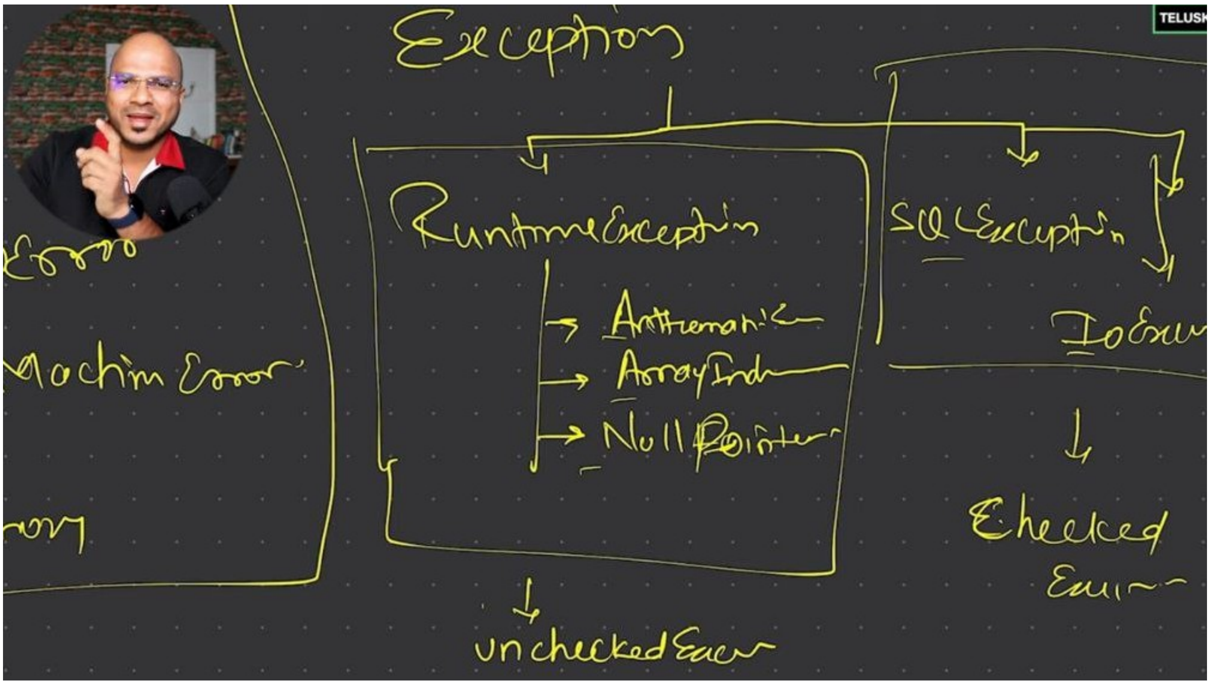
- Example:

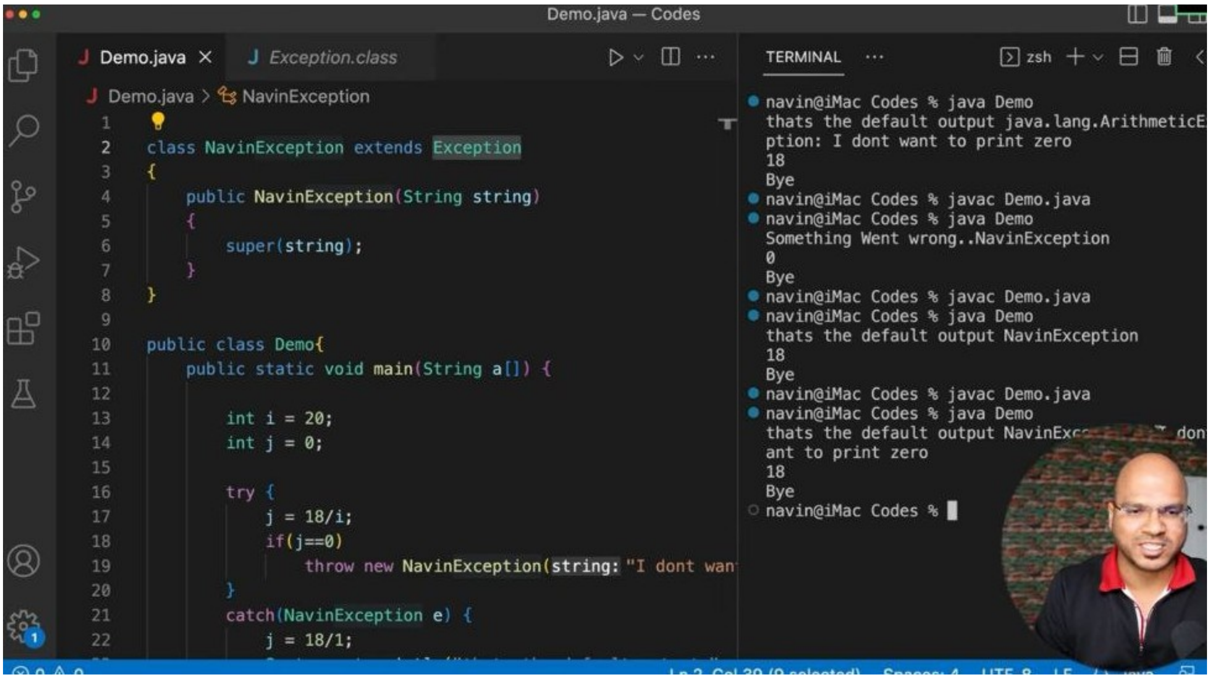


```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemoteService extends Remote {
    String sayHello() throws RemoteException;
}
```

>>Exception

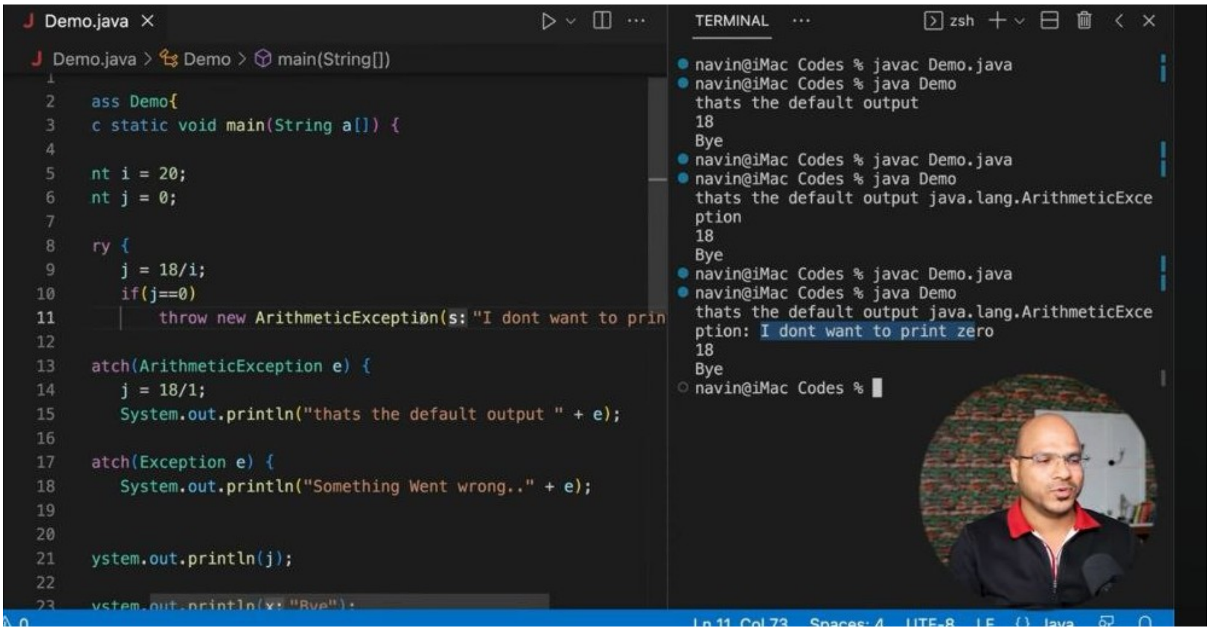

Throwable
Error
Exception
ThreadDeath
IOError
VirtualMachineError
OutofMemory


TELUSK
Exception
Error
Machin Error
RuntimeException
Arithematic
ArrayInd
NullPointer
SQLException
IOExcep
Checked Excep
unchecked Excep



```java
class NavinException extends Exception
{
    public NavinException(String string)
    {
        super(string);
    }
}

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

        int i = 20;
        int j = 0;

        try {
            j = 18/i;
            if(j==0)
                throw new NavinException(string: "I dont wan
        }
        catch(NavinException e) {
            j = 18/1;
```

```
navin@iMac Codes % java Demo
thats the default output java.lang.ArithmeticE
ption: I dont want to print zero
18
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Something Went wrong..NavinException
0
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
thats the default output NavinException
18
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
thats the default output NavinExc          dont
ant to print zero
18
Bye
navin@iMac Codes %
```

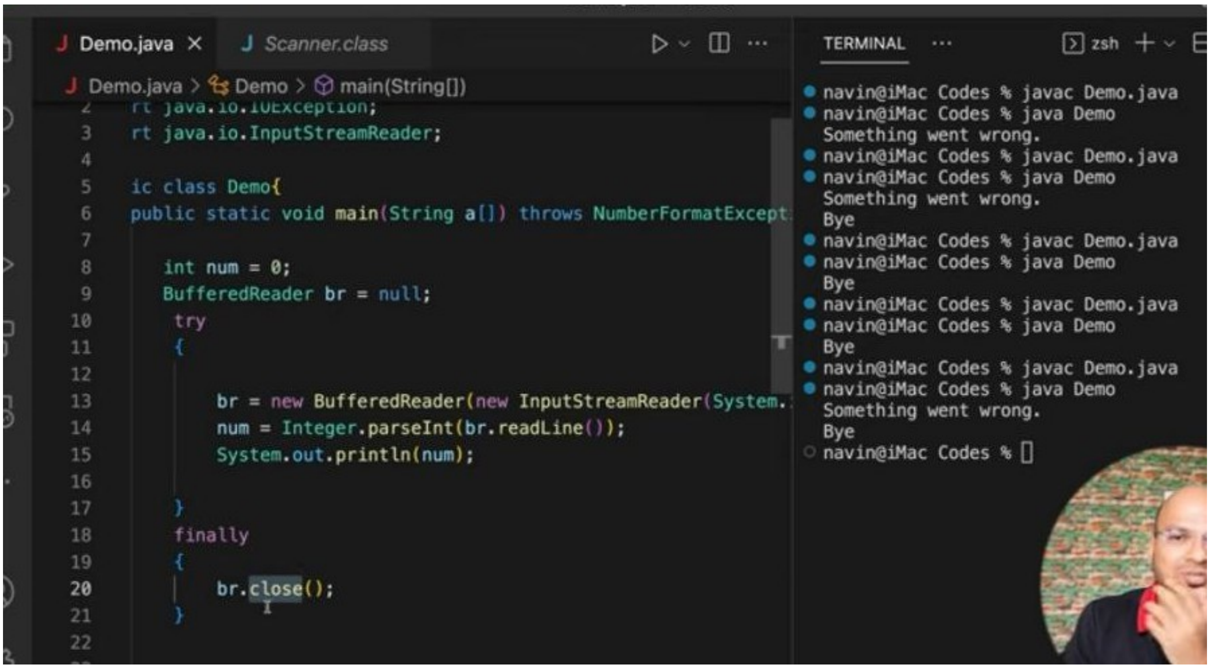


```java
ass Demo{
c static void main(String a[]) {

nt i = 20;
nt j = 0;

ry {
    j = 18/i;
    if(j==0)
        throw new ArithmeticException(s: "I dont want to prin

atch(ArithmeticException e) {
    j = 18/1;
    System.out.println("thats the default output " + e);

atch(Exception e) {
    System.out.println("Something Went wrong.." + e);


ystem.out.println(j);

ystem.out.println(x: "Bye");
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
thats the default output
18
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
thats the default output java.lang.ArithmeticExce
ption
18
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
thats the default output java.lang.ArithmeticExce
ption: I dont want to print zero
18
Bye
navin@iMac Codes %
```

>>Finally keyword

The `finally` keyword in Java is used in conjunction with a `try` block to ensure that a block of code is always executed, regardless of whether an exception is thrown or caught. This is particularly useful for cleanup activities like closing files, releasing resources, or other important finalization tasks.



```java
rt java.io.IOException;
rt java.io.InputStreamReader;

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

    int num = 0;
    BufferedReader br = null;
     try
     {

         br = new BufferedReader(new InputStreamReader(System.
         num = Integer.parseInt(br.readLine());
         System.out.println(num);

     }
     finally
     {
         br.close();
     }

```

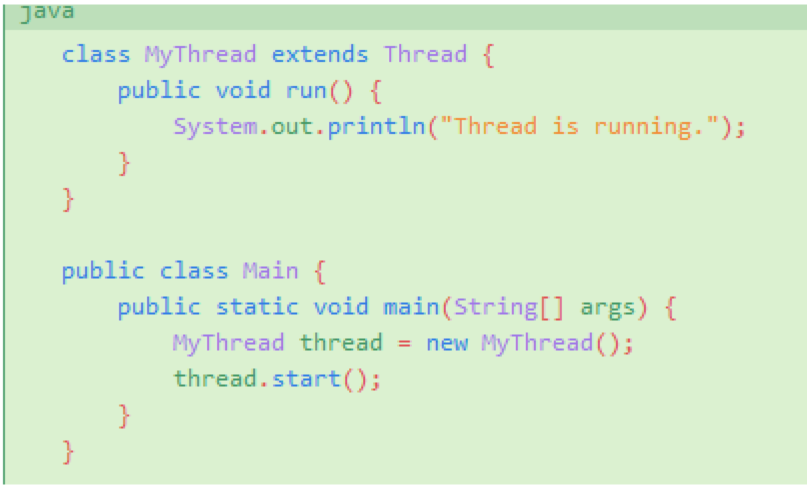
```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Something went wrong.
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Something went wrong.
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Bye
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Something went wrong.
Bye
navin@iMac Codes % 
```

>>Tread

In Java, a thread is a lightweight process that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Threads can be used to perform complicated tasks in the background without interrupting the main program.

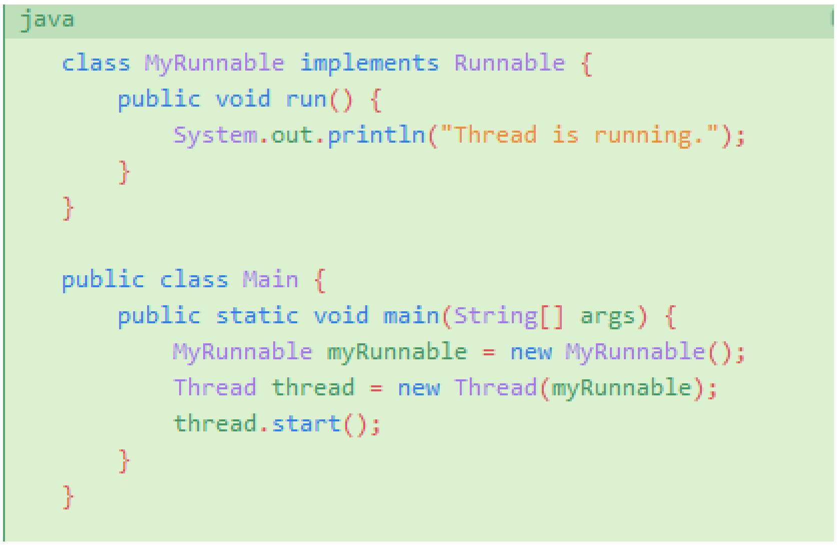
Java provides two ways to create a thread:

1. **Extending the Thread class**: You can create a new class that extends Thread and override its run method. Then, create an instance of this class and call its start method to begin execution



```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

2. **Implementing the Runnable interface**: You can create a class that implements the Runnable interface and pass an instance of this class to a Thread object, then call the start method.



java

```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
```

Key methods associated with threads include:

- start(): Starts the execution of the thread.

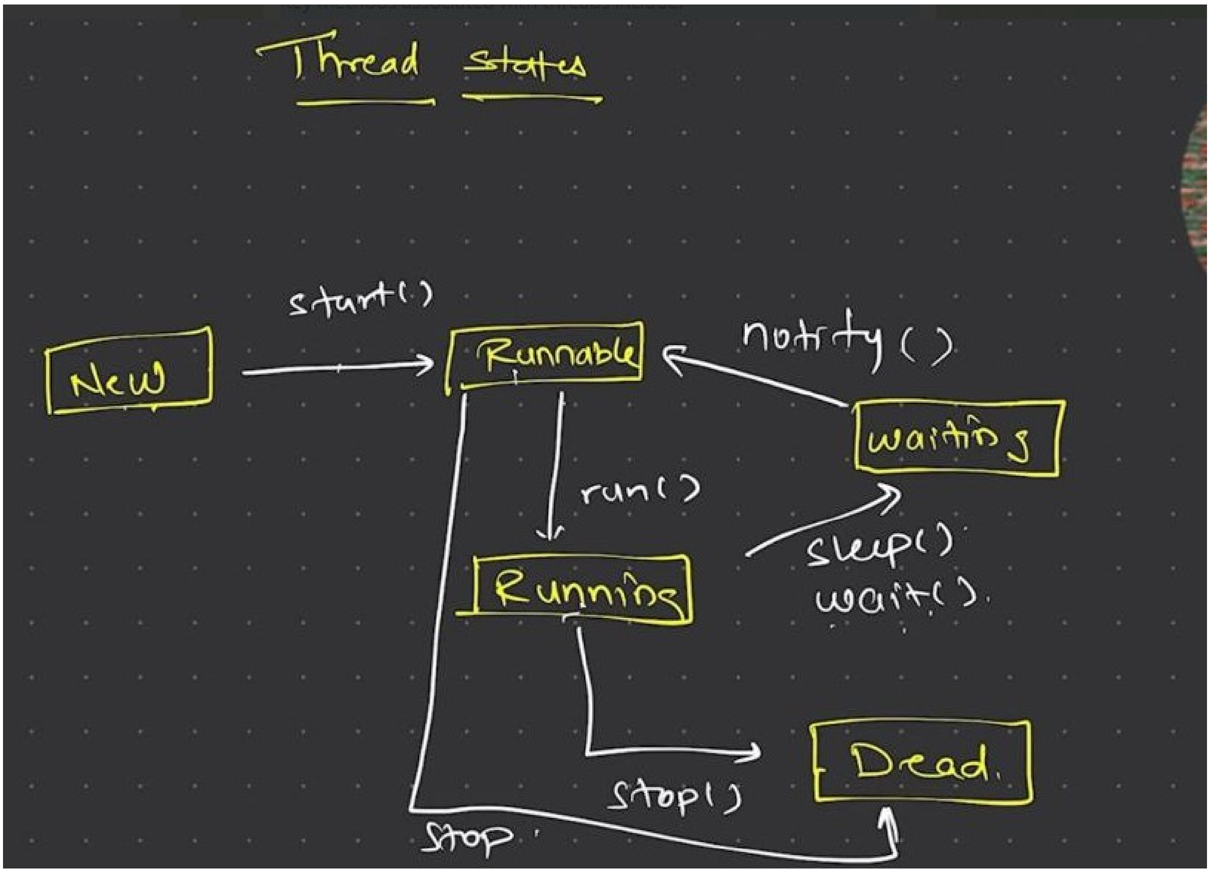
- run(): Contains the code that constitutes the new thread.

- sleep(long millis): Causes the thread to sleep for a specified number of milliseconds.

- join(): Waits for the thread to die.

- interrupt(): Interrupts the thread.

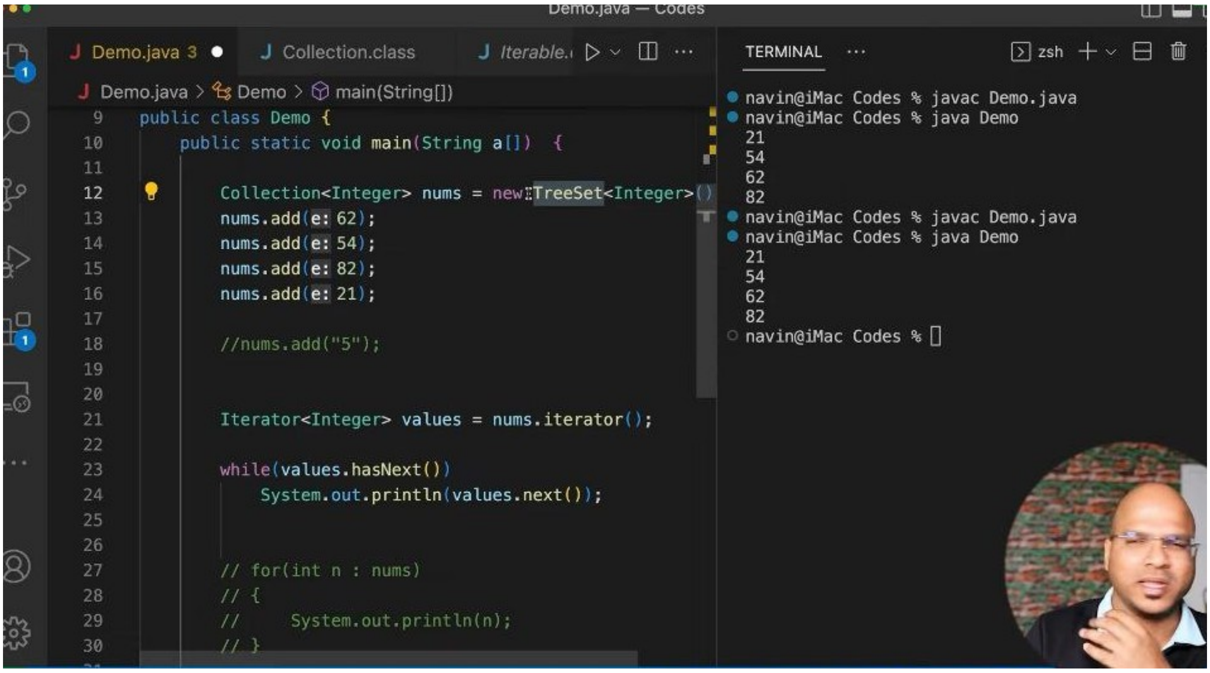
Using threads can improve the performance of a Java application by allowing multiple operations to run concurrently. However, it also introduces complexity, particularly with issues related to synchronization and shared resources



>>

`TreeSet` is a class in Java that implements the `Set` interface and uses a `TreeMap` to store its elements. It provides an ordered set backed by a balanced tree, meaning the elements are sorted according to their natural ordering or by a specified comparator. Here are some key characteristics and features of `TreeSet`:

1. **Sorted Order**: Elements are sorted in ascending order by default. You can also provide a custom comparator to define the order.
2. **No Duplicates**: As with any `Set`, `TreeSet` does not allow duplicate elements.
3. **NavigableSet Implementation**: `TreeSet` implements the `NavigableSet` interface, which provides methods for navigation (e.g., `lower()`, `floor()`, `ceiling()`, `higher()`).



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

        Collection<Integer> nums = new TreeSet<Integer>()
        nums.add(e: 62);
        nums.add(e: 54);
        nums.add(e: 82);
        nums.add(e: 21);

        //nums.add("5");


        Iterator<Integer> values = nums.iterator();

        while(values.hasNext())
            System.out.println(values.next());


        // for(int n : nums)
        // {
        //     System.out.println(n);
        // }
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
21
54
62
82
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
21
54
62
82
navin@iMac Codes % 
```

>> HaspSet and HashMap

In Java, both HashSet and HashMap do not allow duplicate values, but they handle it in different ways due to their internal structure and design.

HashSet:

- Internal Structure: HashSet is backed by a HashMap instance. Essentially, it uses a HashMap to store its elements.

- Storage Mechanism: Each element in a HashSet is stored as a key in the underlying HashMap, with a constant dummy value (like Boolean.TRUE).

- Memory Management: When you add an element to a HashSet, it checks if the HashMap already contains the key. If the key exists, it does not add the duplicate key. This lookup operation is generally O(1) due to hashing. This is efficient in terms of memory management because it prevents the storage of duplicate values by leveraging the HashMap's key uniqueness.

HashMap:

- Internal Structure: HashMap consists of an array of buckets. Each bucket is essentially a linked list (or a balanced tree if the list becomes too long) of Entry objects. Each Entry object contains a key, a value, a hash value, and a pointer to the next Entry.

- Storage Mechanism: When you add a key-value pair to a HashMap, the hash value of the key is calculated and used to determine the appropriate bucket. If the bucket is empty, the Entry is placed there. If not, it checks the linked list for the presence of the key.

- Memory Management: For a key-value pair, the key is stored uniquely. If you attempt to add a key that already exists, the HashMap replaces the old value with the new value but does not store the key again. This ensures no duplicate keys exist, optimizing memory usage.

>>Memories

In Java, memory is primarily divided into two main areas: the heap and the stack. Here's a breakdown of what is stored in each:

1. Heap Memory:

   - Objects: All objects in Java are stored in the heap.

   - Instance Variables: The fields defined inside a class and stored within the objects.

   - JVM Runtime Constant Pool: Part of the method area that stores constants.

2. Stack Memory:

   - Primitive Local Variables: Variables of primitive data types (e.g., int, char) declared inside methods.

   - Reference Variables: Variables that hold references to objects in the heap.

   - Method Calls: Each method call creates a new frame in the stack, storing local variables, method arguments, and the return address.

3. Other Memory Areas:

   - Method Area (or Permanent Generation / Metaspace): Stores class-level data, such as class definitions, static variables, and method bytecode. In Java 8 and later, the Metaspace replaces the Permanent Generation.

   - PC Registers: Each thread has its own PC (Program Counter) register, which keeps track of the JVM instruction being executed.

   - Native Method Stacks: Used for native (non-Java) method calls.

Understanding the distinction between these memory areas is crucial for optimizing memory usage and performance in Java applications.

>>HashMap and HashTable

HashMap and Hashtable are both implementations of the Map interface in Java, but they have several key differences:

HashMap

1. Synchronization: HashMap is not synchronized, meaning it is not thread-safe. If multiple threads access a HashMap concurrently and at least one of the threads modifies it, external synchronization is necessary.

2. Performance: Because it is not synchronized, HashMap is faster than Hashtable for non-thread-safe operations.

3. Null Keys and Values: HashMap allows one null key and multiple null values.

4. Iterator: The iterator of HashMap is fail-fast, which means that if the map is structurally modified after the iterator is created, it will throw a ConcurrentModificationException.

Hashtable

1. Synchronization: Hashtable is synchronized, making it thread-safe. All methods in Hashtable are synchronized, which ensures thread safety.

2. Performance: Because of its synchronized methods, Hashtable is generally slower than HashMap.

3. Null Keys and Values: Hashtable does not allow any null key or value. If a null key or value is used, it throws a NullPointerException.

4. Iterator: The enumerator for Hashtable is not fail-fast.

>> Collection vs Collections vs the Collection API

In Java, Collection, Collections, and the Collection API are related but distinct concepts. Here's an overview of each:

Collection API

- Definition: The Collection API is a framework provided by Java that defines several classes and interfaces for managing groups of objects as a single unit. This includes various interfaces, implementations, and utility methods for data structures like lists, sets, and maps.

- Components:

  - Interfaces: Collection, List, Set, Queue, Map, etc.

  - Implementations: ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, etc.

  - Utility Classes: Collections, Arrays, etc.

Collection

- Definition: Collection is the root interface in the Collection hierarchy. It represents a group of objects, known as elements. The Collection interface is part of the Java Collection API.

- Key Methods:

  - add()

  - remove()

  - size()

  - iterator()

  - clear()

- Subinterfaces: List, Set, Queue (which all extend Collection).

Collections

- Definition: Collections is a utility class in the Java Collection API. It consists exclusively of static methods that operate on or return collections. It cannot be instantiated.

- Purpose: Provides methods for:

  - Sorting (sort())

  - Searching (binarySearch())

  - Thread-safety (synchronizedCollection(), synchronizedList())

  - Immutability (unmodifiableCollection(), unmodifiableList())

  - Other utility operations (reverse(), shuffle(), min(), max())

Summary

- Collection API: The overall framework that includes interfaces, implementations, and utility methods for working with collections of objects.

- Collection: The root interface that represents a group of objects. It is extended by more specific subinterfaces like List, Set, and Queue.

- Collections: A utility class with static methods to manipulate and return collections. It provides algorithms and methods for common operations like sorting, searching, and making collections thread-safe or immutable.

>> Comparable and Comparator

Comparable

1. Interface Definition: Comparable<T>

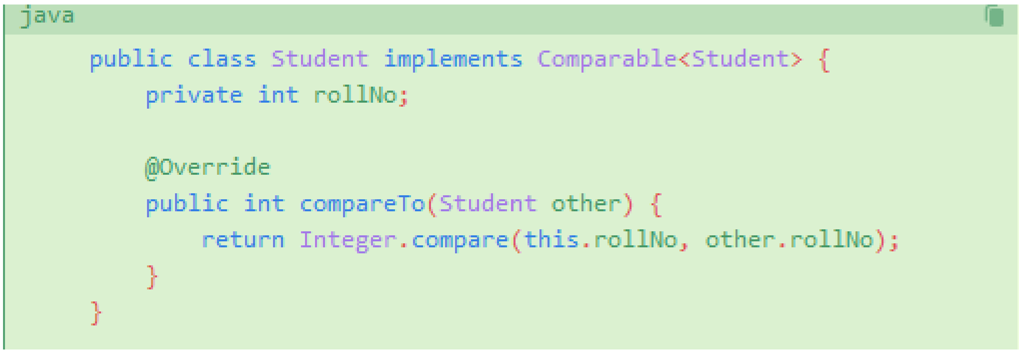
2. Purpose: Defines the natural ordering of objects. It allows an object to compare itself with another object of the same type.

3. Method: compareTo(T o)

- Signature: int compareTo(T o)

- Description: Compares the current object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

4. Usage: Implemented by the class whose instances need to be compared.



```java
public class Student implements Comparable<Student> {
    private int rollNo;

    @Override
    public int compareTo(Student other) {
        return Integer.compare(this.rollNo, other.rollNo);
    }
}
```

5. Default Sorting: When using sorting methods (like Collections.sort()), if a class implements Comparable, its compareTo() method is used for ordering

Comparator

1. Interface Definition: Comparator<T>

2. Purpose: Defines a custom ordering of objects. It allows a class to compare two other objects.

3. Methods:

- compare(T o1, T o2)

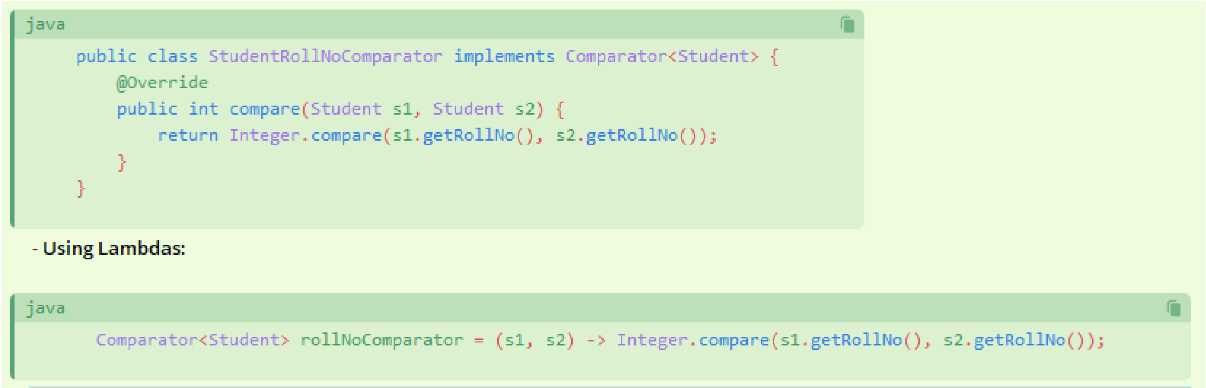
  - Signature: int compare(T o1, T o2)

  - Description: Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

- reversed(), thenComparing(), and other default and static methods for chaining and composing comparators.

4. Usage: Implemented by a separate class or used as an anonymous class or lambda expression.

Example:



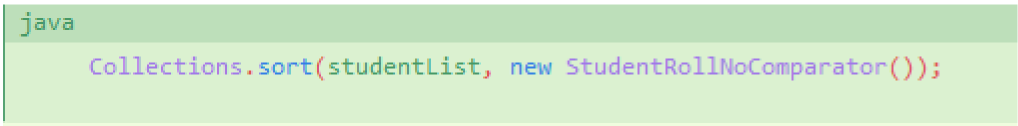
```java
    public class StudentRollNoComparator implements Comparator<Student> {
        @Override
        public int compare(Student s1, Student s2) {
            return Integer.compare(s1.getRollNo(), s2.getRollNo());
        }
    }
```

- Using Lambdas:

```java
    Comparator<Student> rollNoComparator = (s1, s2) -> Integer.compare(s1.getRollNo(), s2.getRollNo());
```

Custom Sorting: When using sorting methods (like Collections.sort()), you can pass an instance of Comparator to define the custom order.

- Example:



```java
    Collections.sort(studentList, new StudentRollNoComparator());
```

Summary

- Comparable:
  - Used for natural ordering.
  - Implemented by the class itself.
  - Has a single method compareTo().

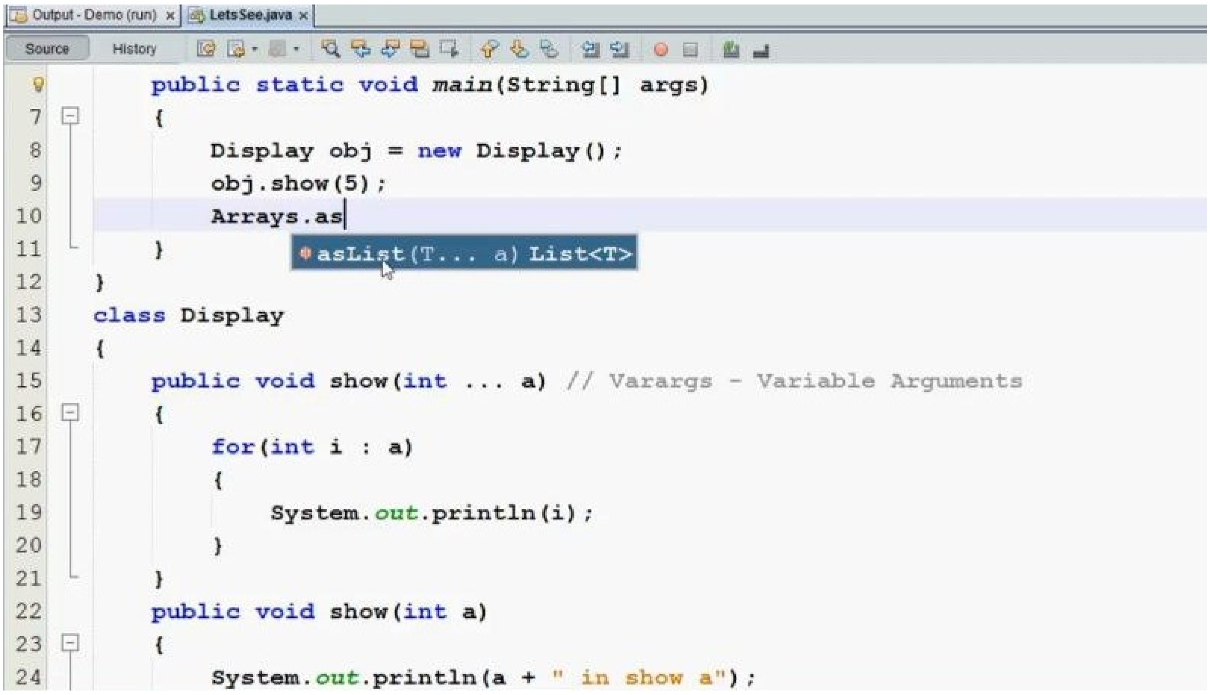
- Comparator:
  - Used for custom ordering.
  - Implemented by a separate class or using lambdas.
  - Has a method compare().
  - More flexible and can define multiple sorting sequences.

In practice, use Comparable when objects have a natural order and Comparator when you need multiple ways to compare objects or do not control the source code of the class being compared.

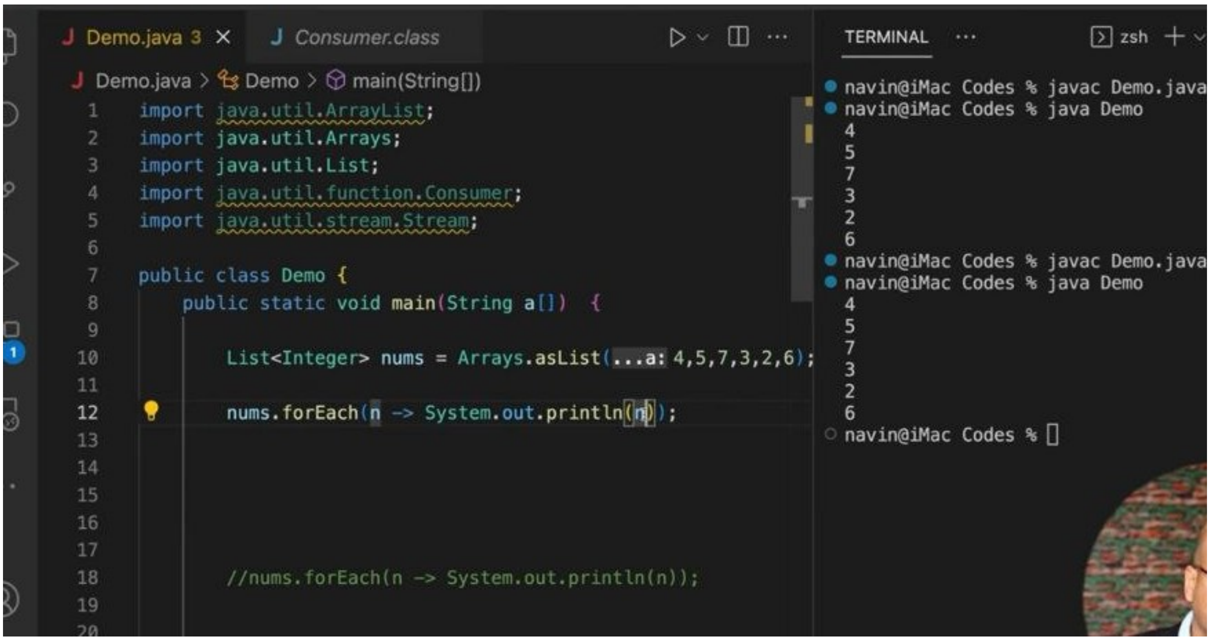
>>varargs

In Java, the term `varargs` refers to a feature that allows a method to accept a variable number of arguments. This is achieved using the ellipsis (`...`) syntax. The method can then handle these arguments as an array. Varargs provide a convenient way to pass a variable number of arguments of the same type to a method without needing to define multiple overloaded methods.

** Within the method, the varargs parameter is treated as an array of the specified type.



```
    public static void main(String[] args)
    {
        Display obj = new Display();
        obj.show(5);
        Arrays.as
    }
}
class Display
{
    public void show(int ... a) // Varargs - Variable Arguments
    {
        for(int i : a)
        {
            System.out.println(i);
        }
    }
    public void show(int a)
    {
        System.out.println(a + " in show a");
```



```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.stream.Stream;

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

        List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);

        nums.forEach(n -> System.out.println(n));






        //nums.forEach(n -> System.out.println(n));
```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
4
5
7
3
2
6
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
4
5
7
3
2
6
navin@iMac Codes %
```

>>Stream

A stream represents an ordered sequence of data that can be read from or written to. Streams are a fundamental concept for performing input and output (I/O) operations in Java, enabling the reading and writing of data from different sources like files, network connections, and memory.

Types of Streams in Java

Byte Streams

Byte streams handle I/O of raw binary data. They read and write data in bytes and are suitable for handling binary files like images and audio files.

- InputStream: An abstract class representing an input stream of bytes. Common subclasses include:
  - FileInputStream
  - BufferedInputStream
  - DataInputStream

- OutputStream: An abstract class representing an output stream of bytes. Common subclasses include:
  - FileOutputStream
  - BufferedOutputStream
  - DataOutputStream

Character Streams

Character streams handle I/O of character data. They read and write data in characters (16-bit Unicode). These streams are suitable for handling text files.

- Reader: An abstract class representing a character input stream. Common subclasses include:
  - FileReader
  - BufferedReader
  - InputStreamReader

- Writer: An abstract class representing a character output stream. Common subclasses include:

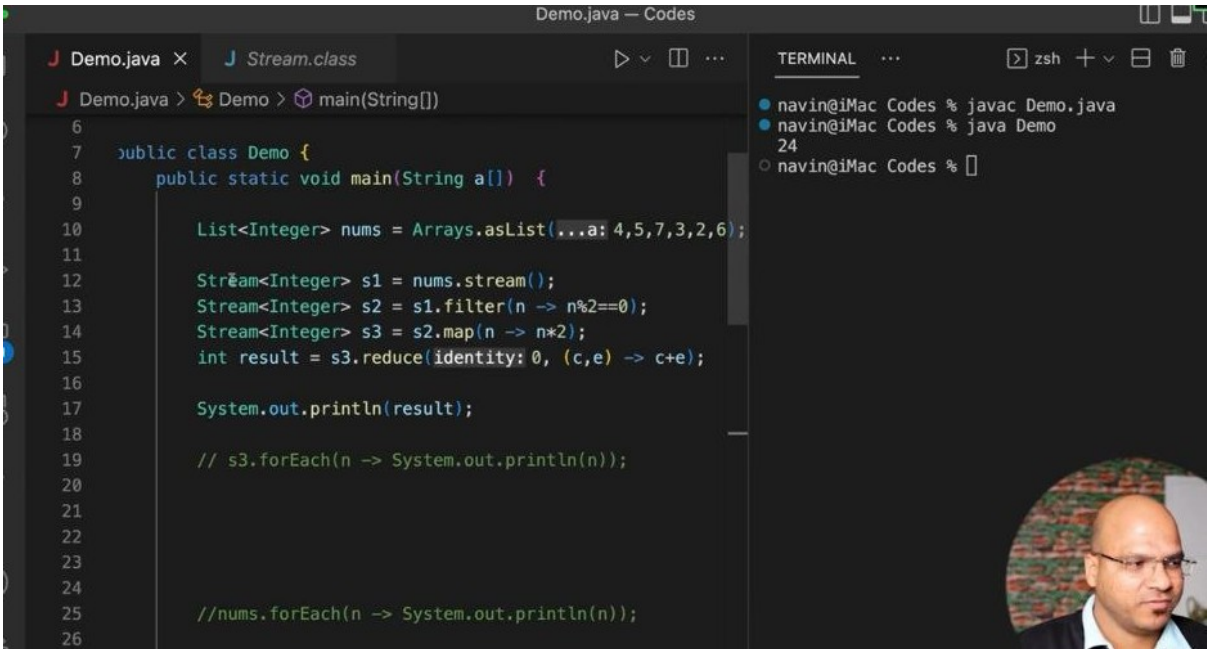
  - FileWriter

  - BufferedWriter

  - OutputStreamWriter

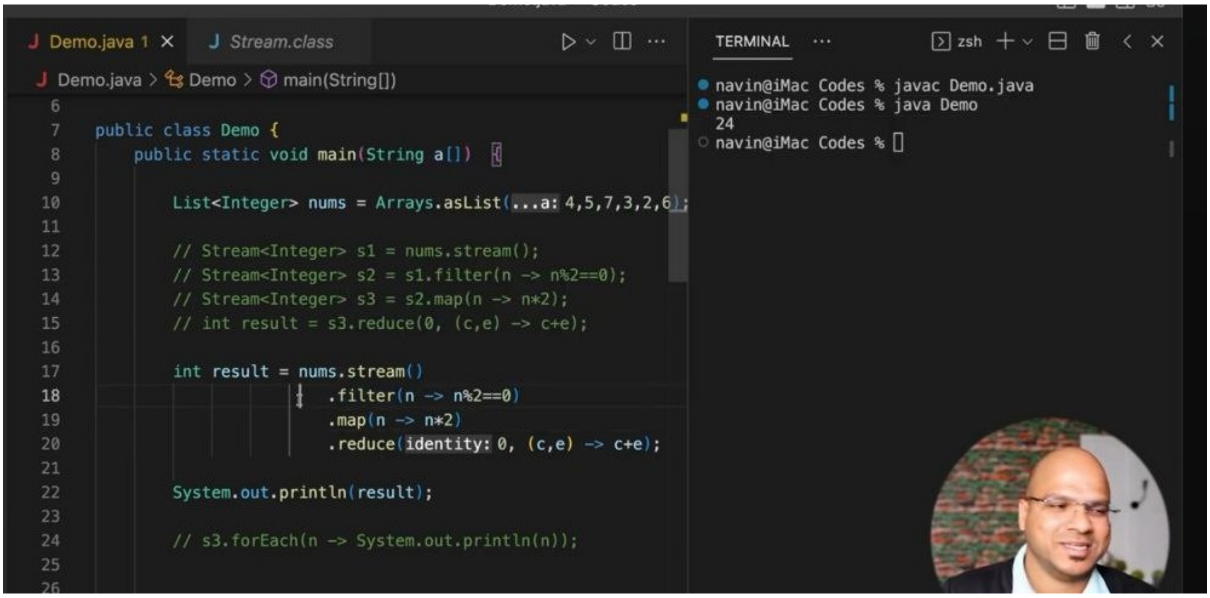
Buffered Streams

Buffered streams use an internal buffer to read or write data in larger chunks, which can improve I/O performance by reducing the number of I/OBufferedInputStreamredInBufferedOutputStreamedOutputStream** for BufferedReaderBuffeBufferedWriterBufferedWriter** for character streams



```java

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

        List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);

        Stream<Integer> s1 = nums.stream();
        Stream<Integer> s2 = s1.filter(n -> n%2==0);
        Stream<Integer> s3 = s2.map(n -> n*2);
        int result = s3.reduce(identity: 0, (c,e) -> c+e);

        System.out.println(result);

        // s3.forEach(n -> System.out.println(n));






        //nums.forEach(n -> System.out.println(n));

```

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
24
navin@iMac Codes %
```



```java

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

        List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);

        // Stream<Integer> s1 = nums.stream();
        // Stream<Integer> s2 = s1.filter(n -> n%2==0);
        // Stream<Integer> s3 = s2.map(n -> n*2);
        // int result = s3.reduce(0, (c,e) -> c+e);

        int result = nums.stream()
                        .filter(n -> n%2==0)
                        .map(n -> n*2)
                        .reduce(identity: 0, (c,e) -> c+e);

        System.out.println(result);

        // s3.forEach(n -> System.out.println(n));

```

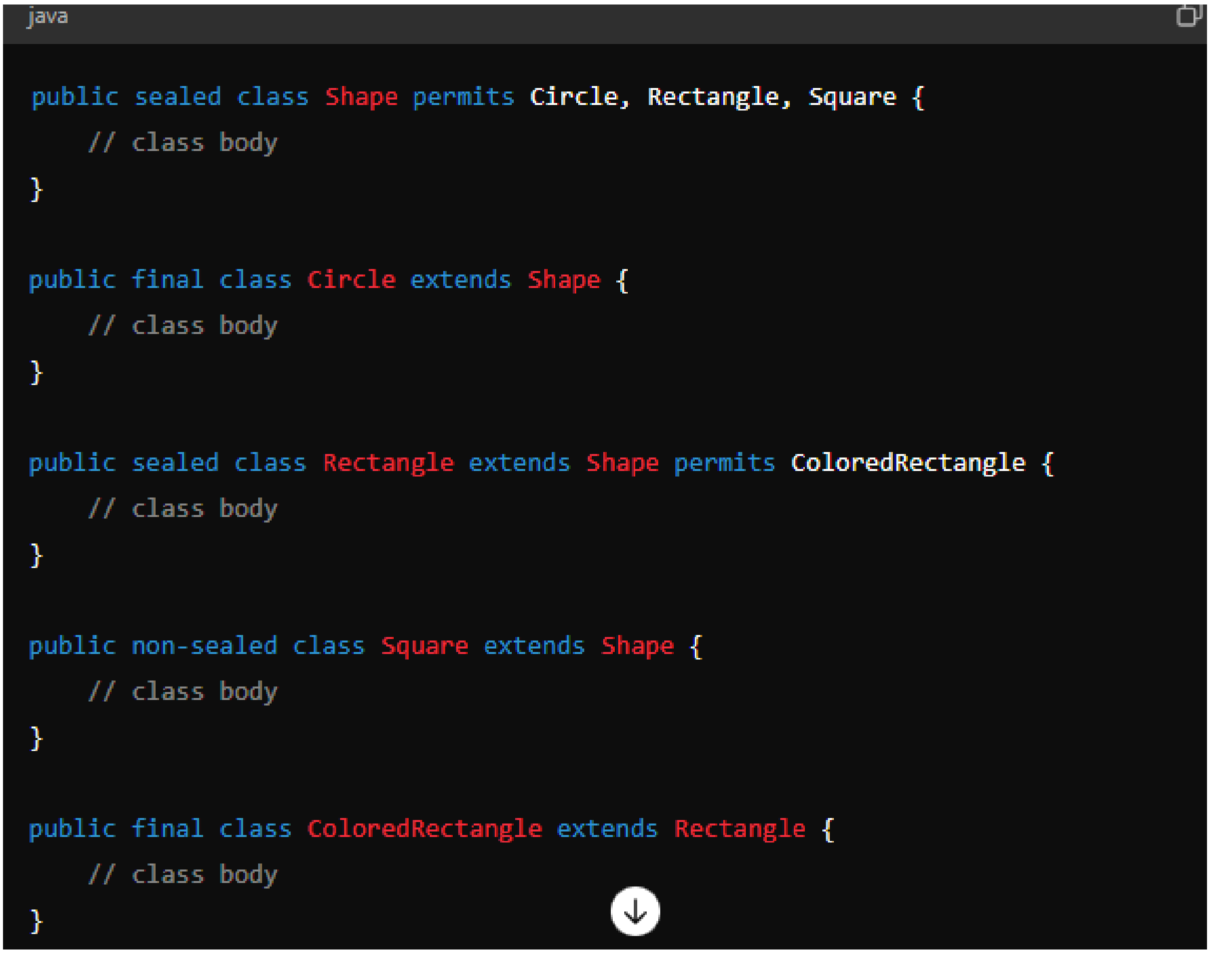
```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
24
navin@iMac Codes % 
```

>>sealed class

In Java, the `sealed` keyword is used to define a class or interface that restricts which other classes or interfaces can extend or implement it. This feature, introduced in Java 15 as a preview and finalized in Java 17, helps to provide more control over the inheritance hierarchy, allowing for better maintainability and security.

Key Concepts of Sealed Classes

1. **Sealed Class**: A class that restricts which other classes can extend it.
2. **Permitted Subclasses**: Only classes explicitly specified in the sealed class declaration can extend the sealed class.
3. **Non-sealed Classes**: A subclass that is allowed to be extended by other classes without restriction.
4. **Final Classes**: A subclass that cannot be further extended.
5. **Abstract Classes**: A subclass that can be extended but must be specified as permitted.



```java
public sealed class Shape permits Circle, Rectangle, Square {
    // class body
}

public final class Circle extends Shape {
    // class body
}

public sealed class Rectangle extends Shape permits ColoredRectangle {
    // class body
}

public non-sealed class Square extends Shape {
    // class body
}

public final class ColoredRectangle extends Rectangle {
    // class body
}
```

>> Non-static variables (instance variables) can't be accessed inside static methods because static methods belong to the class itself, not to any instance of the class. Instance variables are tied to specific objects of the class, and since static methods don't have a reference to any specific object, they can't directly access these variables. To access instance variables within a static method, you must pass an instance of the class to the method and use that instance to access the variables.

>>Object cloning

Object cloning in Java refers to creating a duplicate or exact copy of an existing object. This process allows you to replicate an object's state and behavior without requiring the same initial setup or data retrieval operations that were used to create the original object. Cloning can be achieved in Java using either the Cloneable interface or by implementing the Cloneable interface and overriding the clone() method.

Using Cloneable Interface

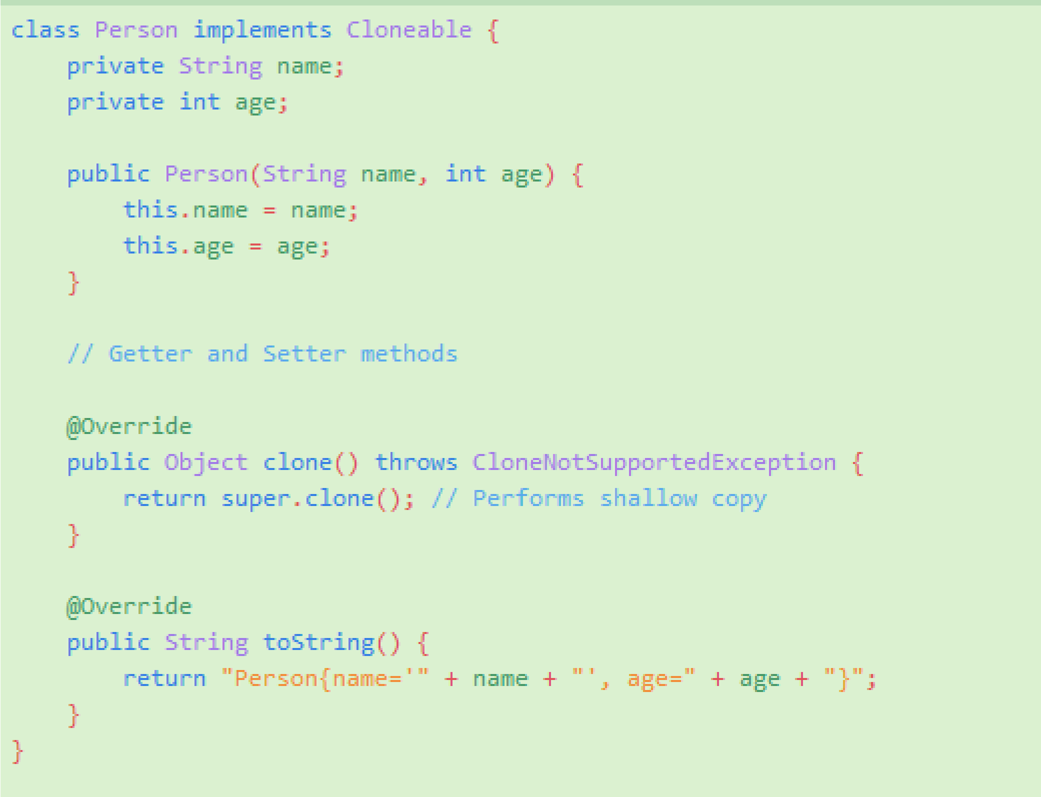
1. **Implementing Cloneable**:

  - Cloneable is a marker interface (contains no methods) that indicates the class allows its instances to be cloned.

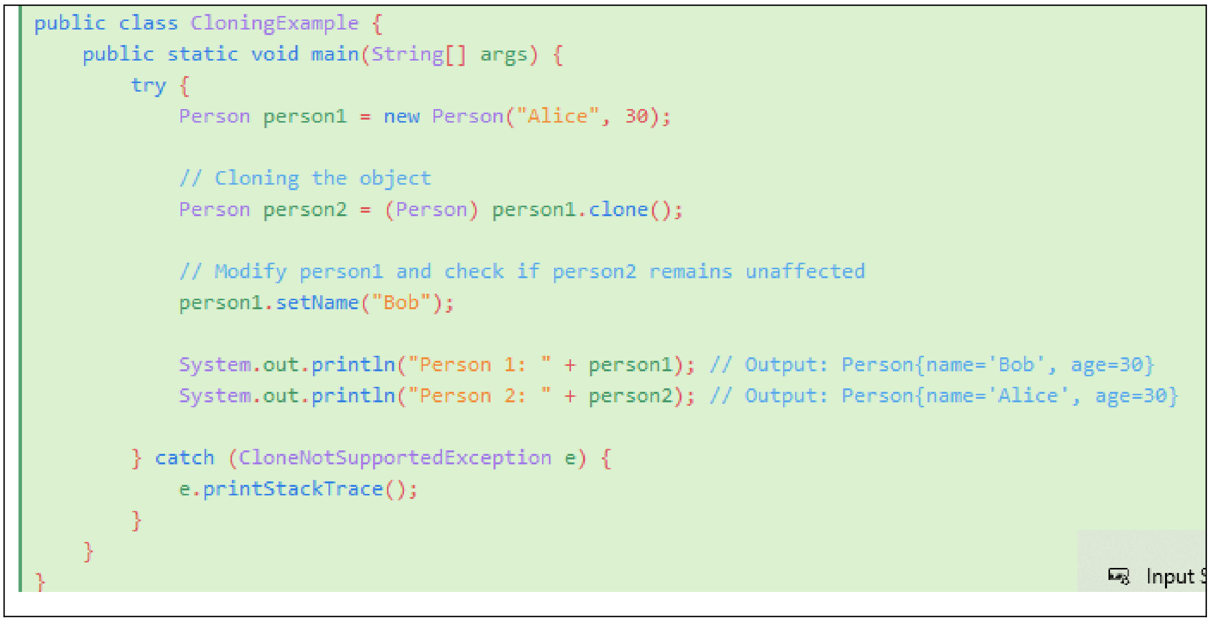
2. **Overriding clone() Method**:

  - The clone() method is defined in the Object class with the protected access level. To use it, you must override it with public access level in your claDeep vs Shallow Cloningloning*Shallow cloningloning**: By default, clone() performs a shallow copy, copying all fields of the object. If fields are primitive types, they are copied directly. If fields are reference types, the references are copied (not the objects themselvesDeep cloningloning**: If the object contains reference fields that need to be deeply copied (i.e., copies of referenced objects are also created), you need to override clone() to achieve this.

Example of Object Cloning

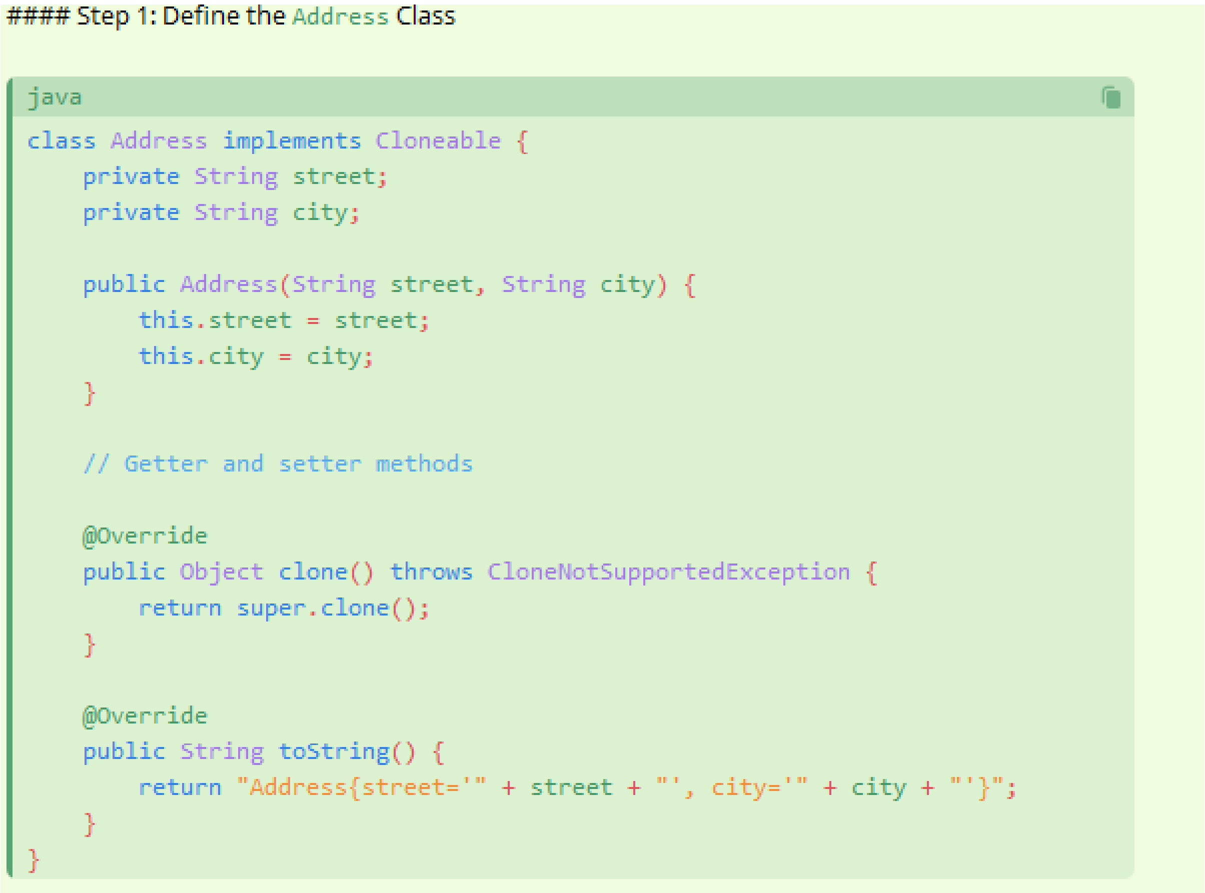


```java
class Person implements Cloneable {
    private String name;
    private int age;

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

    // Getter and Setter methods

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // Performs shallow copy
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```



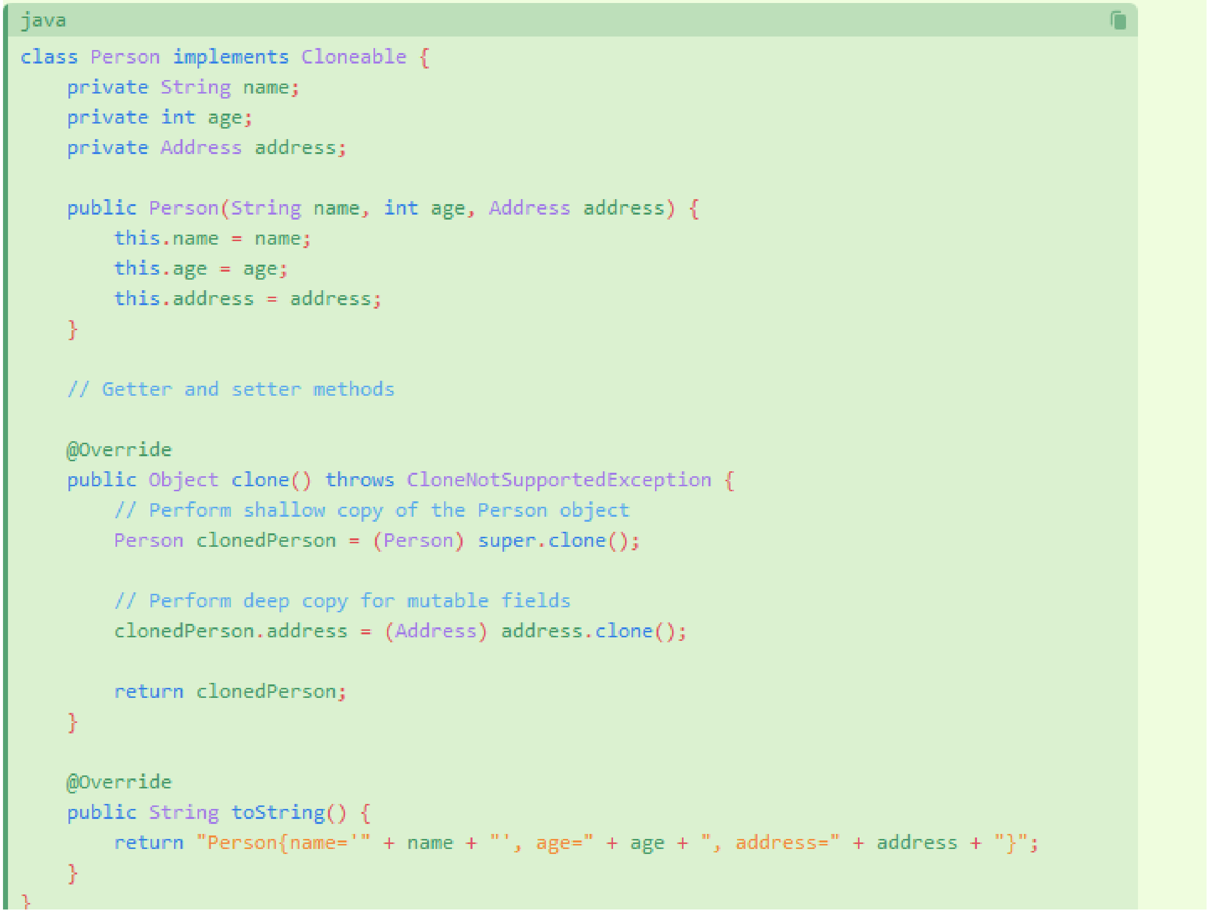
```
public class CloningExample {
    public static void main(String[] args) {
        try {
            Person person1 = new Person("Alice", 30);

            // Cloning the object
            Person person2 = (Person) person1.clone();

            // Modify person1 and check if person2 remains unaffected
            person1.setName("Bob");

            System.out.println("Person 1: " + person1); // Output: Person{name='Bob', age=30}
            System.out.println("Person 2: " + person2); // Output: Person{name='Alice', age=30}

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

>> Example of Deep copy using Clonable:

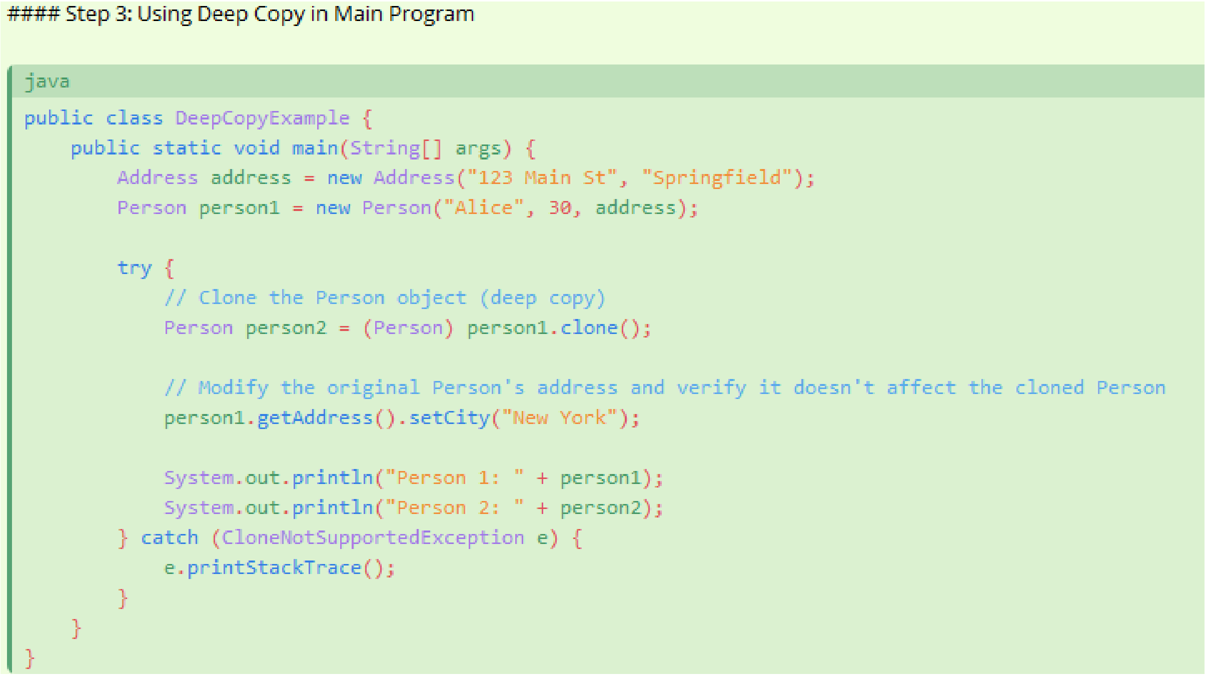


Step 1: Define the Address Class

```
class Address implements Cloneable {
    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    // Getter and setter methods

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public String toString() {
        return "Address{street='" + street + "', city='" + city + "'}";
    }
}
```



```java
class Person implements Cloneable {
    private String name;
    private int age;
    private Address address;

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    // Getter and setter methods

    @Override
    public Object clone() throws CloneNotSupportedException {
        // Perform shallow copy of the Person object
        Person clonedPerson = (Person) super.clone();

        // Perform deep copy for mutable fields
        clonedPerson.address = (Address) address.clone();

        return clonedPerson;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", address=" + address + "}";
    }
}
```



Step 3: Using Deep Copy in Main Program

```java
public class DeepCopyExample {
    public static void main(String[] args) {
        Address address = new Address("123 Main St", "Springfield");
        Person person1 = new Person("Alice", 30, address);

        try {
            // Clone the Person object (deep copy)
            Person person2 = (Person) person1.clone();

            // Modify the original Person's address and verify it doesn't affect the cloned Person
            person1.getAddress().setCity("New York");

            System.out.println("Person 1: " + person1);
            System.out.println("Person 2: " + person2);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

Core Classes

1. Object (Class)

   - Description: The root class of the Java class hierarchy. Every class has Object as a superclass. It provides methods such as equals(), hashCode(), toString(), clone(), and wait()/notify() for thread synchronization.

2. String (Class)

   - Description: Represents a sequence of characters. Strings are immutable in Java, meaning their values cannot be changed once created.

3. Math (Class)

   - Description: Provides methods for performing basic numeric operations such as exponentiation, logarithms, square roots, and trigonometric functions.

4. System (Class)

   - Description: Provides access to system-related functionality such as standard input/output streams, environment variables, and system properties.

5. Thread (Class)

   - Description: Represents a thread of execution in a Java program. Threads allow concurrent execution of code.

6. Exception (Class)

   - Description: The superclass of all exceptions that can be thrown by the Java Virtual Machine (JVM). Provides mechanisms for error handling in Java.

Collection Framework

1. List<E> (Interface)

   - Description: An ordered collection (also known as a sequence) that allows duplicate elements. Common implementations include ArrayList and LinkedList.

2. Set<E> (Interface)

- Description: A collection that does not allow duplicate elements. Common implementations include HashSet, LinkedHashSet, and TreeSet.

3. Map<K, V> (Interface)

- Description: An object that maps keys to values. A map cannot contain duplicate keys. Common implementations include HashMap, LinkedHashMap, and TreeMap.

4. Queue<E> (Interface)

- Description: A collection designed for holding elements prior to processing. Common implementations include LinkedList and PriorityQueue.

5. Iterator<E> (Interface)

- Description: An interface that provides methods to iterate over a collection.

I/O (Input/Output)

1. File (Class)

- Description: Represents a file or directory path in the filesystem.

2. InputStream / OutputStream (Class)

- Description: Abstract classes that represent input and output streams of bytes. Common subclasses include FileInputStream, FileOutputStream, and BufferedInputStream.

3. Reader / Writer (Class)

- Description: Abstract classes for reading and writing character streams. Common subclasses include FileReader, FileWriter, BufferedReader, and BufferedWriter.

Concurrency

1. Runnable (Interface)

- Description: Represents a task that can be executed concurrently by a thread. Implemented by classes that define a single method, run().

2. Callable<V> (Interface)

- Description: Similar to Runnable, but can return a result and throw a checked exception. Defines a single method, call().

3. ExecutorService (Interface)

- Description: Provides methods for managing the execution of asynchronous tasks.

4. CompletableFuture (Class)

- Description: Represents a future result of an asynchronous computation, and provides methods to handle the result once it becomes available.

Networking

1. Socket (Class)

- Description: Provides the means to establish a connection between a client and a server.

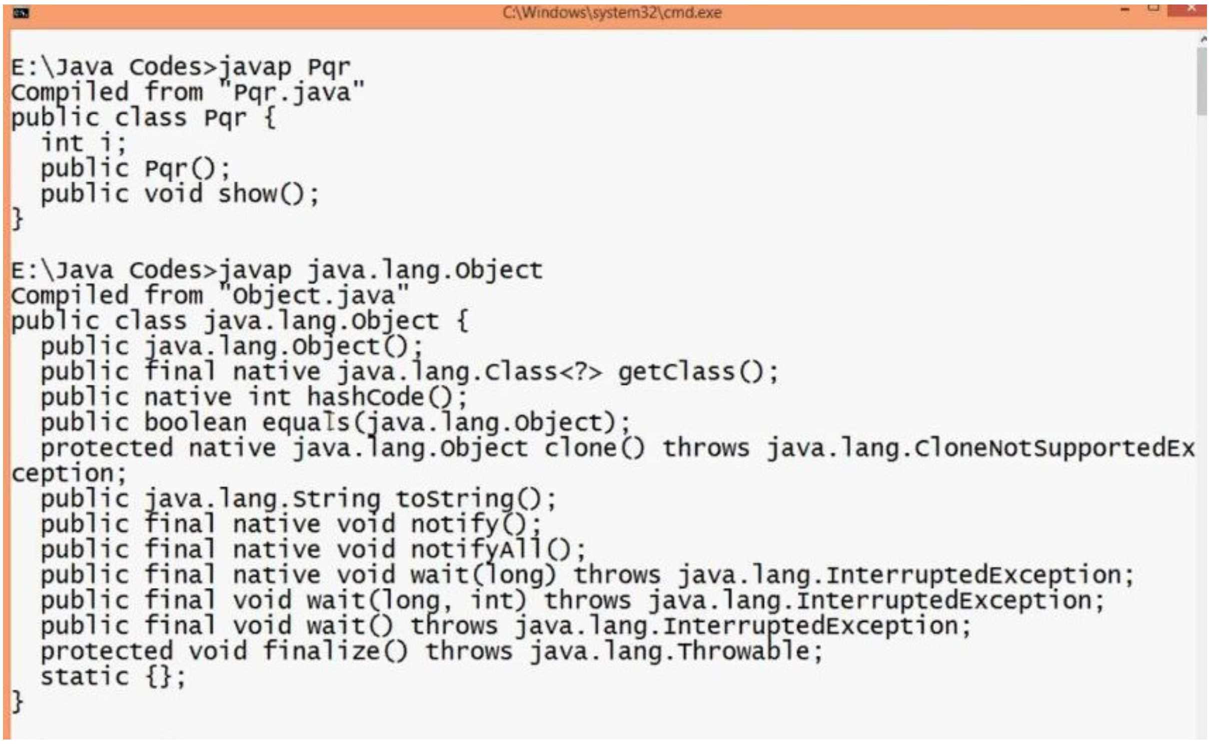
2. ServerSocket (Class)

- Description: Used by servers to listen for incoming connections.

3. URL (Class)

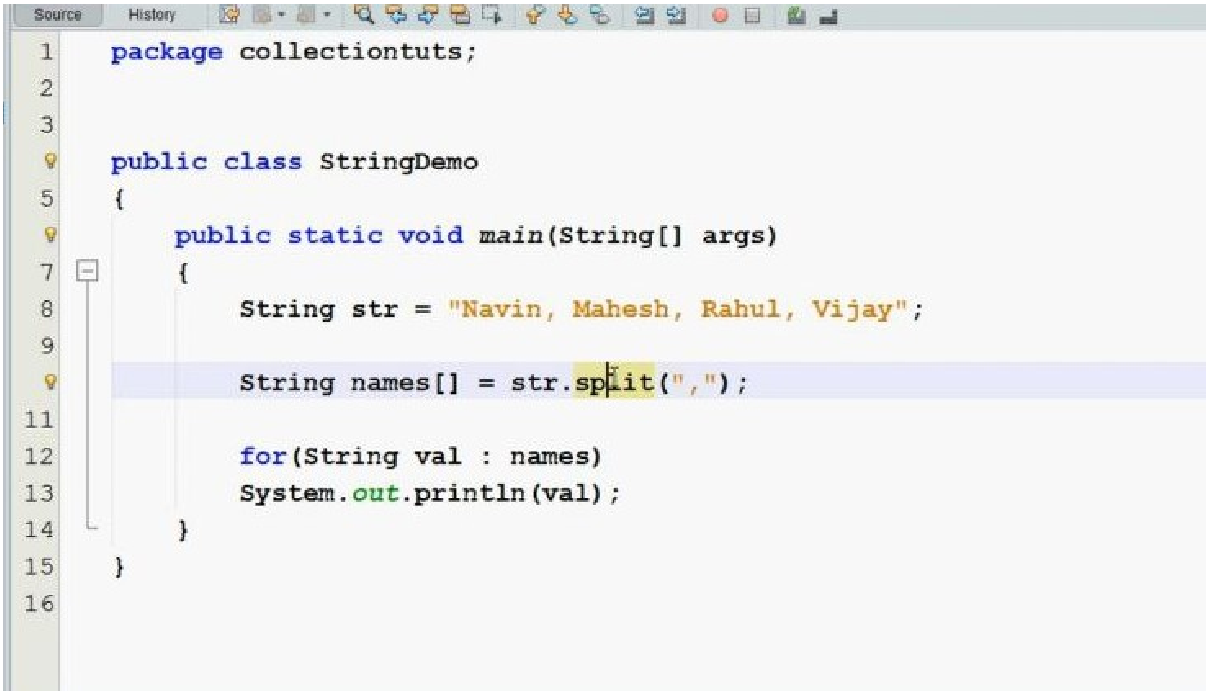
- Description: Represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.

>>Javap



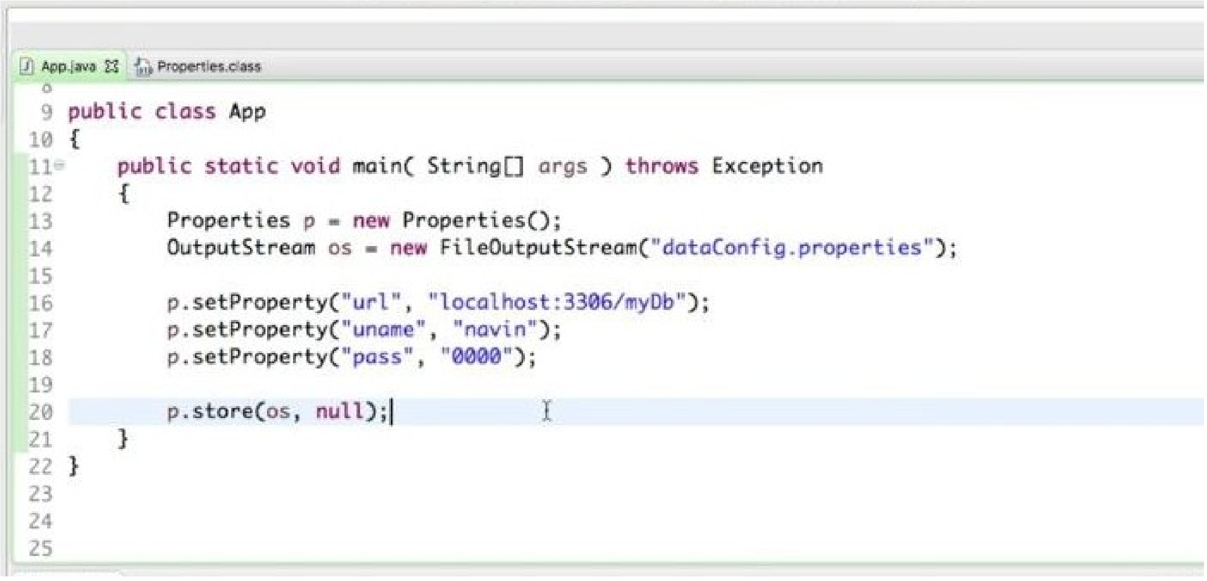
```
E:\Java Codes>javap Pqr
Compiled from "Pqr.java"
public class Pqr {
  int i;
  public Pqr();
  public void show();
}

E:\Java Codes>javap java.lang.Object
Compiled from "Object.java"
public class java.lang.Object {
  public java.lang.Object();
  public final native java.lang.Class<?> getClass();
  public native int hashCode();
  public boolean equals(java.lang.Object);
  protected native java.lang.Object clone() throws java.lang.CloneNotSupportedEx
ception;
  public java.lang.String toString();
  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long) throws java.lang.InterruptedException;
  public final void wait(long, int) throws java.lang.InterruptedException;
  public final void wait() throws java.lang.InterruptedException;
  protected void finalize() throws java.lang.Throwable;
  static {};
}
```

>>Split

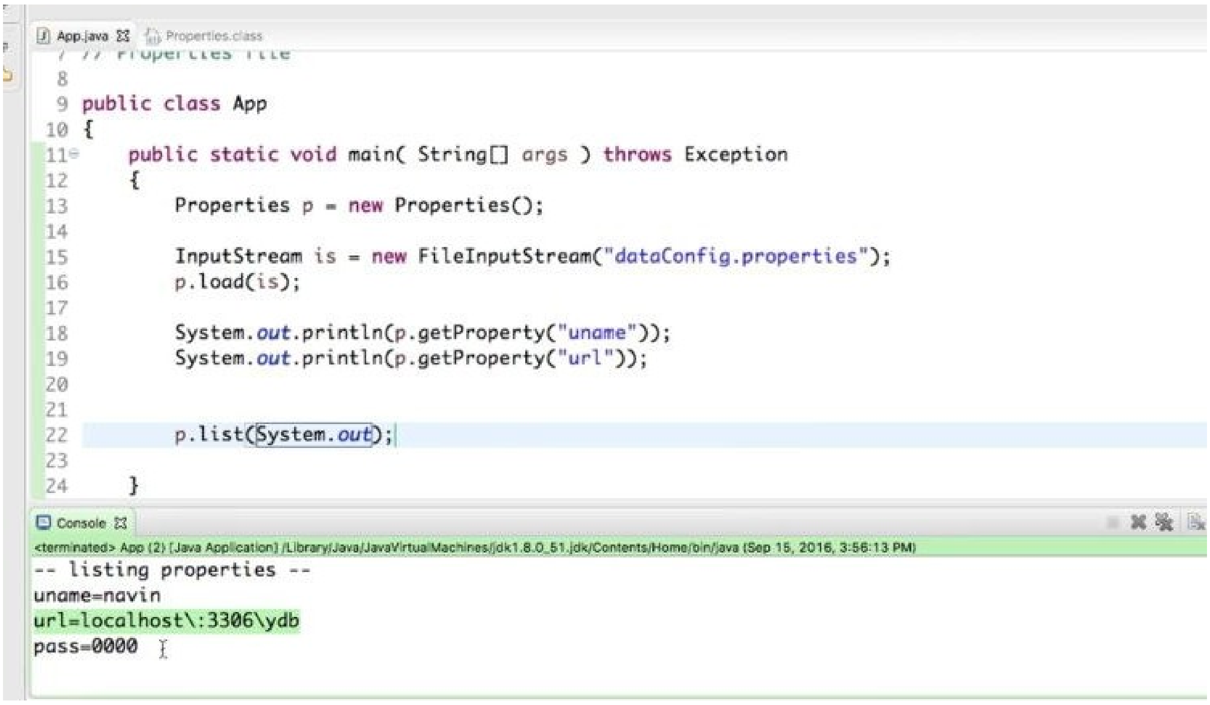


```java
package collectiontuts;


public class StringDemo
{
    public static void main(String[] args)
    {
        String str = "Navin, Mahesh, Rahul, Vijay";

        String names[] = str.split(",");

        for(String val : names)
        System.out.println(val);
    }
}
```

>>setProperties



```
public class App
{
    public static void main( String[] args ) throws Exception
    {
        Properties p = new Properties();
        OutputStream os = new FileOutputStream("dataConfig.properties");

        p.setProperty("url", "localhost:3306/myDb");
        p.setProperty("uname", "navin");
        p.setProperty("pass", "0000");

        p.store(os, null);
    }
}
```



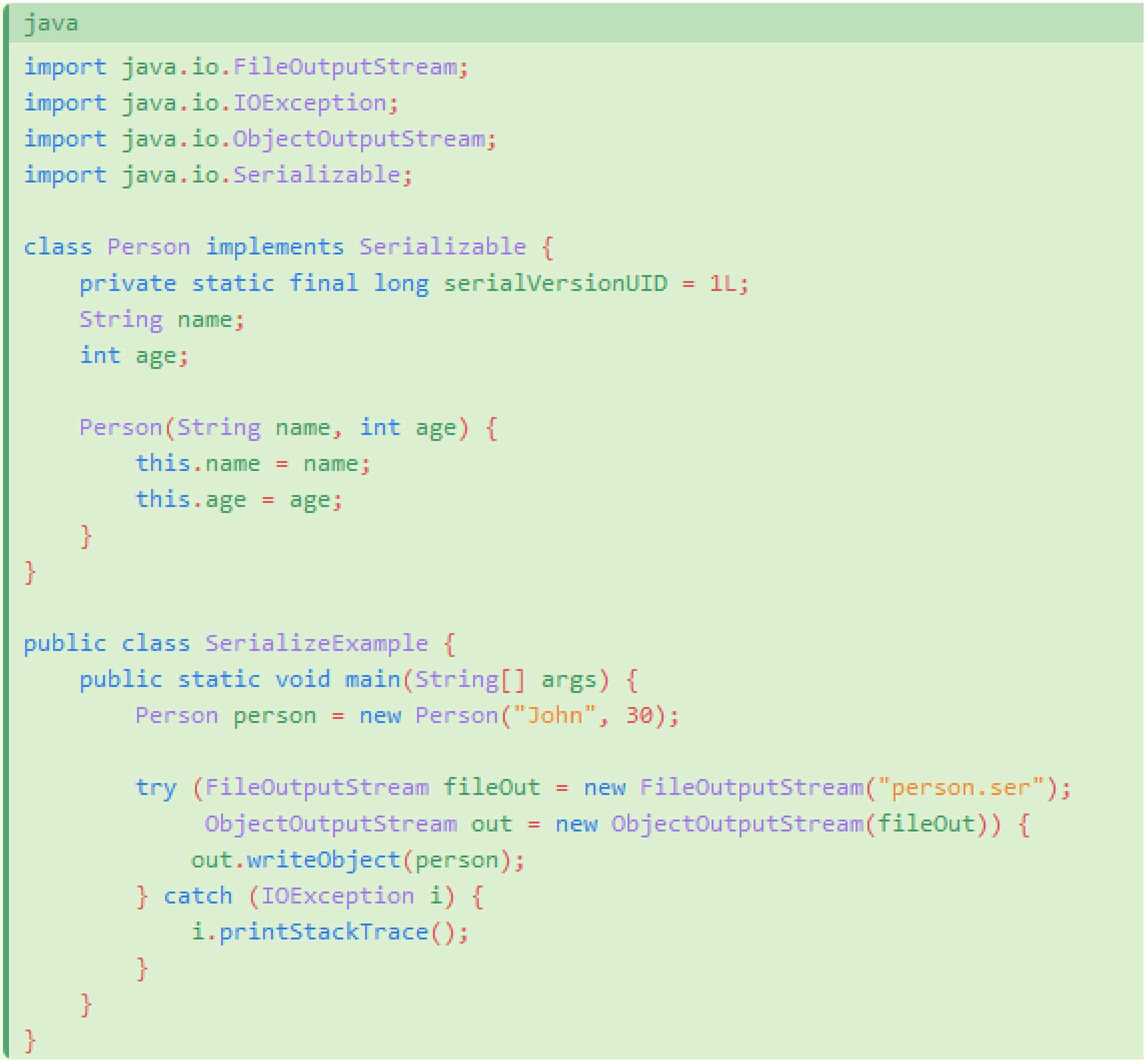
```
// Properties file

public class App
{
    public static void main( String[] args ) throws Exception
    {
        Properties p = new Properties();

        InputStream is = new FileInputStream("dataConfig.properties");
        p.load(is);

        System.out.println(p.getProperty("uname"));
        System.out.println(p.getProperty("url"));


        p.list(System.out);

    }
```

```
<terminated> App (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java (Sep 15, 2016, 3:56:13 PM)
-- listing properties --
uname=navin
url=localhost\:3306\ydb
pass=0000
```

>>

Serialization

Serialization is the process of converting a Java object into a byte stream. This byte stream can then be saved to a file, sent over a network, or stored in a database.

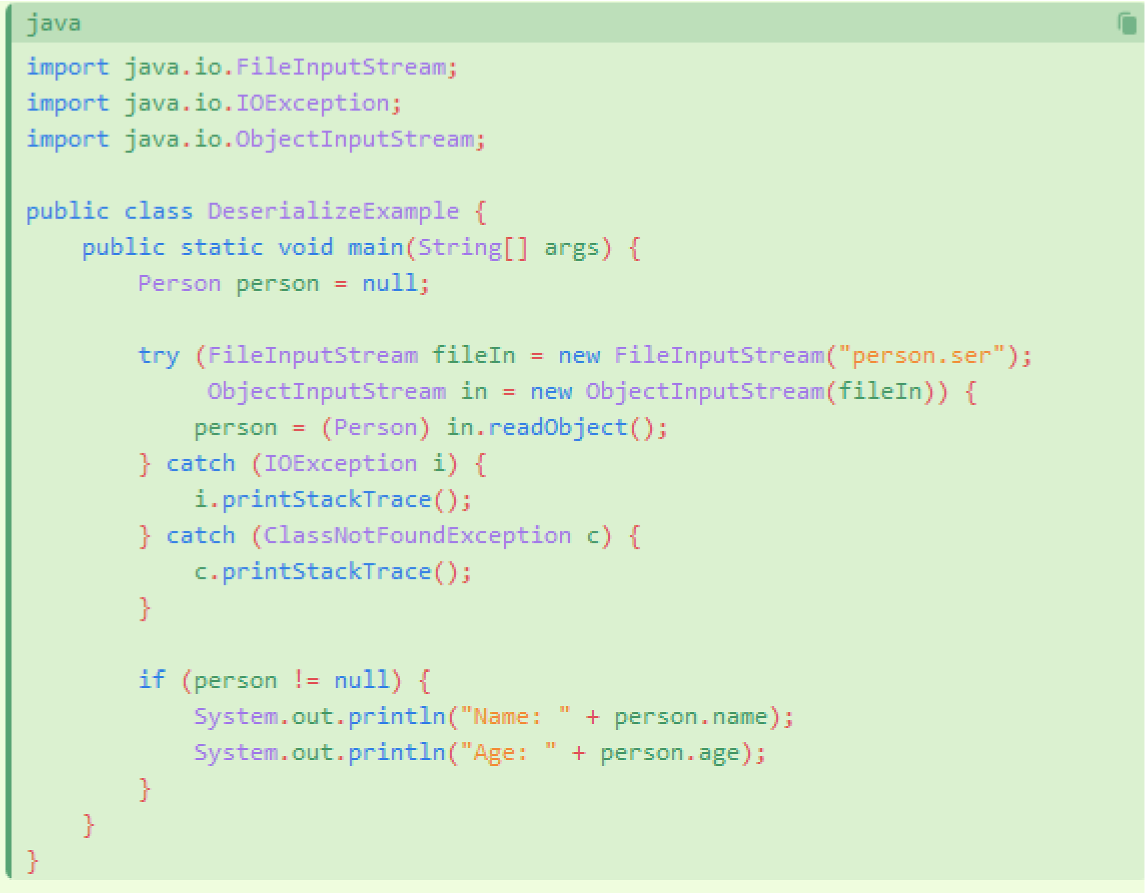


java

```java
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializeExample {
    public static void main(String[] args) {
        Person person = new Person("John", 30);

        try (FileOutputStream fileOut = new FileOutputStream("person.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
            out.writeObject(person);
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

Deserialization

Deserialization is the process of converting a byte stream back into a Java object. This is essentially the reverse process of serialization.



java

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializeExample {
    public static void main(String[] args) {
        Person person = null;

        try (FileInputStream fileIn = new FileInputStream("person.ser");
             ObjectInputStream in = new ObjectInputStream(fileIn)) {
            person = (Person) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
        } catch (ClassNotFoundException c) {
            c.printStackTrace();
        }

        if (person != null) {
            System.out.println("Name: " + person.name);
            System.out.println("Age: " + person.age);
        }
    }
}
```

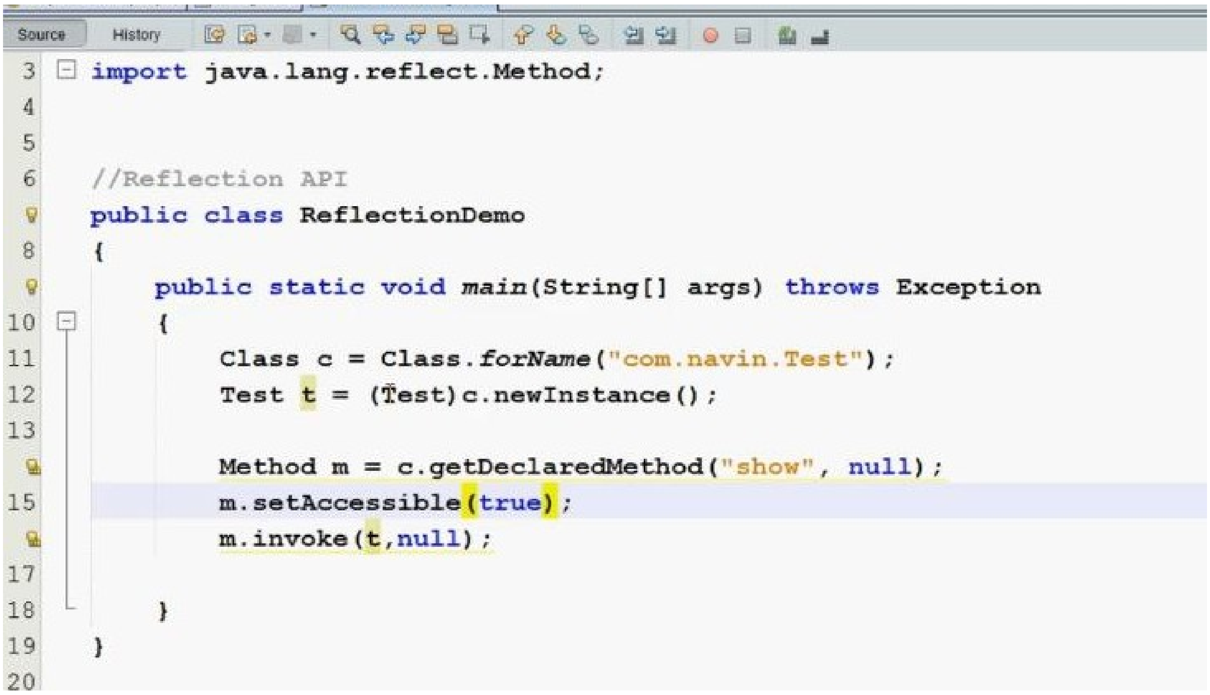
Important Points

1. serialVersionUID: It's good practice to include a serialVersionUID field in serializable classes. This field is used to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, an InvalidClassException is thrown.

2. Transient Fields: Fields marked with the transient keyword are not serialized. This is useful for fields that you don't want to be saved, like passwords or other sensitive data.

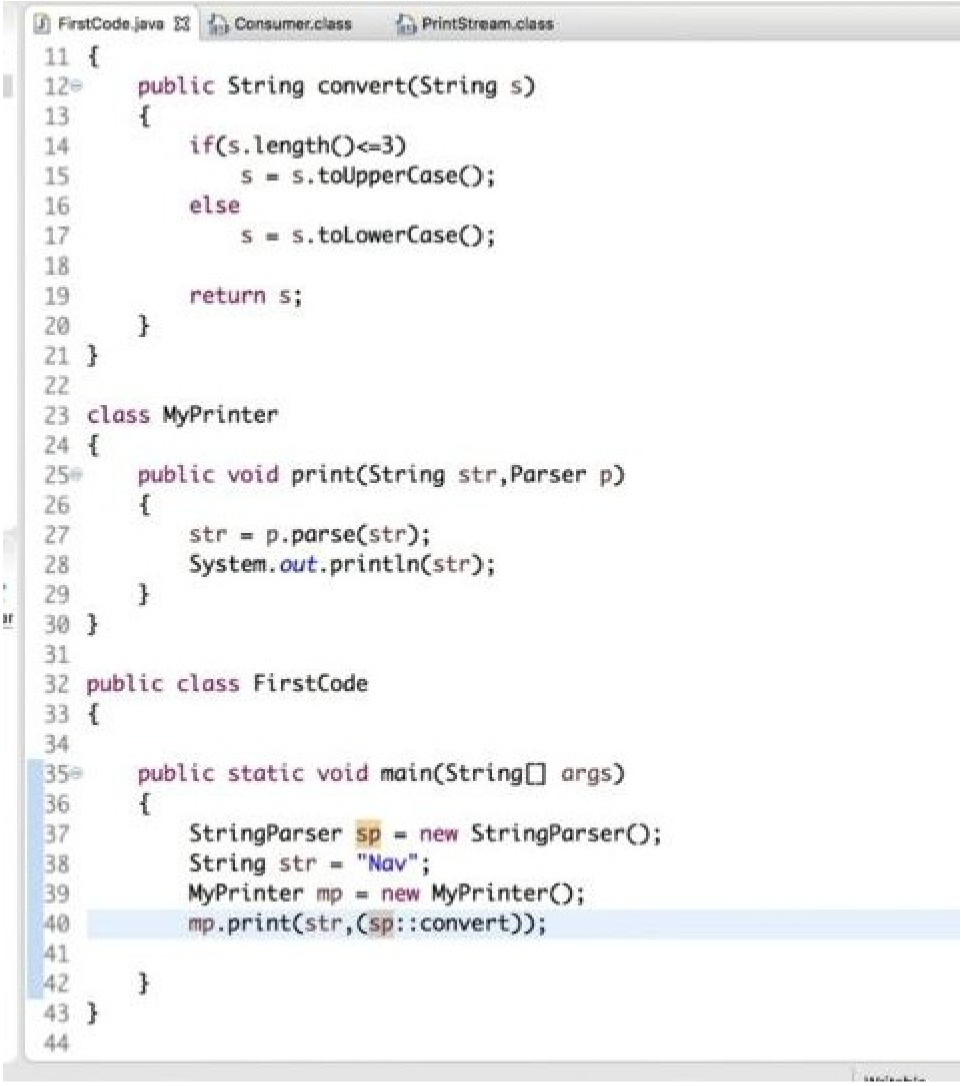
3. Custom Serialization: If you need more control over the serialization process, you can provide custom implementations of writeObject and readObject methods.

>>Reflection API



```java
import java.lang.reflect.Method;


//Reflection API
public class ReflectionDemo
{
    public static void main(String[] args) throws Exception
    {
        Class c = Class.forName("com.navin.Test");
        Test t = (Test)c.newInstance();

        Method m = c.getDeclaredMethod("show", null);
        m.setAccessible(true);
        m.invoke(t,null);

    }
}

```

>>Method reference



```java
{
    public String convert(String s)
    {
        if(s.length()<=3)
            s = s.toUpperCase();
        else
            s = s.toLowerCase();

        return s;
    }
}

class MyPrinter
{
    public void print(String str,Parser p)
    {
        str = p.parse(str);
        System.out.println(str);
    }
}

public class FirstCode
{

    public static void main(String[] args)
    {
        StringParser sp = new StringParser();
        String str = "Nav";
        MyPrinter mp = new MyPrinter();
        mp.print(str,(sp::convert));

    }
}

```

Spring Boot Notes

IoC

Inversion of Control (IoC) refers to the principle where the control of objects or portions of a program is transferred to a container or framework. In simpler terms, it means that the creation and management of objects are handled by the Spring framework rather than being controlled manually by the application code.

Key Points of IoC in Spring:

5. **Dependency Injection (DI):** The most common way IoC is implemented in Spring. DI is a design pattern used to implement IoC, where the Spring container injects the dependencies into a class at runtime rather than at compile time.
6. **Bean Management:** The Spring container manages the lifecycle of beans, including their creation, initialization, and destruction. The configuration of these beans can be done using XML configuration files, Java annotations, or Java-based configuration.
7. **Types of Dependency Injection:**
    a. **Constructor Injection:** Dependencies are provided through a class constructor.
    b. **Setter Injection:** Dependencies are provided through setter methods.
    c. **Field Injection:** Dependencies are directly assigned to fields. (Less preferred due to testing and maintenance difficulties.)

DI

Dependency Injection (DI) in Spring is a design pattern and a core concept that allows the Spring framework to manage dependencies between objects. It promotes loose coupling by injecting dependencies (objects a class needs to perform its functions) rather than creating them internally. This makes the code more modular, easier to test, and maintainable.

Advantages of Dependency Injection

4. **Loose Coupling:** Classes are not tightly coupled, making them easier to test and maintain.
5. **Reusability:** Components can be reused across different parts of the application.
6. **Testability:** Easier to write unit tests by injecting mock dependencies.
7. **Flexibility:** Easy to swap out implementations of a dependency without changing the dependent class.

IoC Container

An IoC (Inversion of Control) container in Spring is a core component responsible for managing the lifecycle and configuration of application objects. The IoC container uses

Dependency Injection (DI) to manage object creation, wiring, and destruction, promoting loose coupling and modular design.

Key Responsibilities of the IoC Container

6. **Object Creation:** The container instantiates and manages the lifecycle of beans (objects managed by the Spring container).
7. **Dependency Injection:** The container injects dependencies into beans, typically through constructor, setter, or field injection.
8. **Configuration Management:** The container reads configuration metadata (XML, annotations, or Java configuration) to determine how to assemble and configure beans.
9. **Bean Lifecycle Management:** The container manages the complete lifecycle of beans, including their creation, initialization, and destruction.
10. **Event Handling:** The container can publish and listen to application events, facilitating communication between beans.

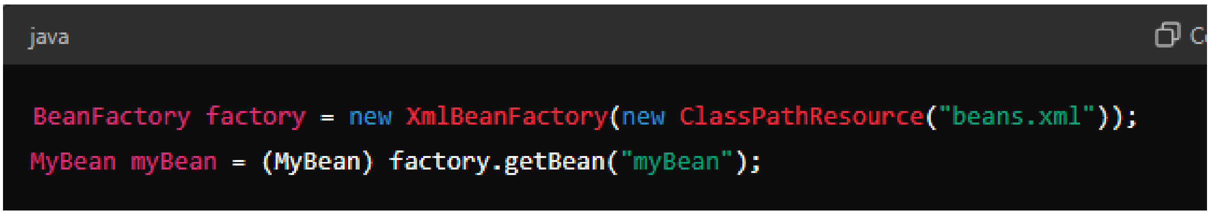
In Spring, there are primarily two types of IoC (Inversion of Control) containers: **BeanFactory** and **ApplicationContext**. Each serves different purposes and offers varying levels of functionality.

1. BeanFactory

The `BeanFactory` is the simplest container, providing the basic features of dependency injection. It is suitable for lightweight, non-enterprise applications where only basic DI is required.

Characteristics:

- **Lazy Initialization:** Beans are created only when they are requested, not at startup.
- **Minimal Overhead:** Provides only basic DI functionality, which results in lower memory and CPU usage.



```java
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
MyBean myBean = (MyBean) factory.getBean("myBean");
```

2. ApplicationContext

The `ApplicationContext` is a more advanced container that builds on the `BeanFactory` interface. It provides additional enterprise-specific functionality, making it suitable for most Spring-based applications.

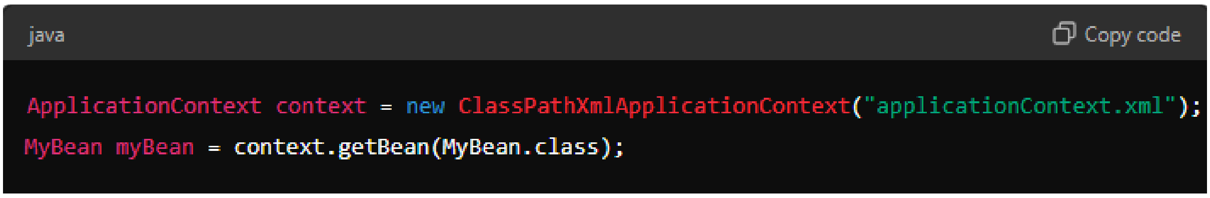
Characteristics:

- **Eager Initialization:** By default, beans are created at startup, leading to faster retrieval but potentially higher startup time.
- **Internationalization Support:** Provides support for message resources and internationalization.
- **Event Propagation:** Can publish and listen to application events.
- **Declarative Mechanisms:** Supports declarative mechanisms such as annotations and aspect-oriented programming (AOP).

Common Implementations of `ApplicationContext`:

a. ClassPathXmlApplicationContext

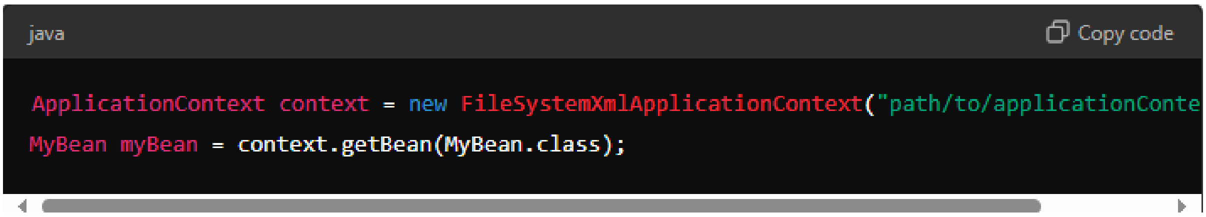
Loads context definition from an XML file located in the classpath.



```java
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
MyBean myBean = context.getBean(MyBean.class);
```

b. FileSystemXmlApplicationContext

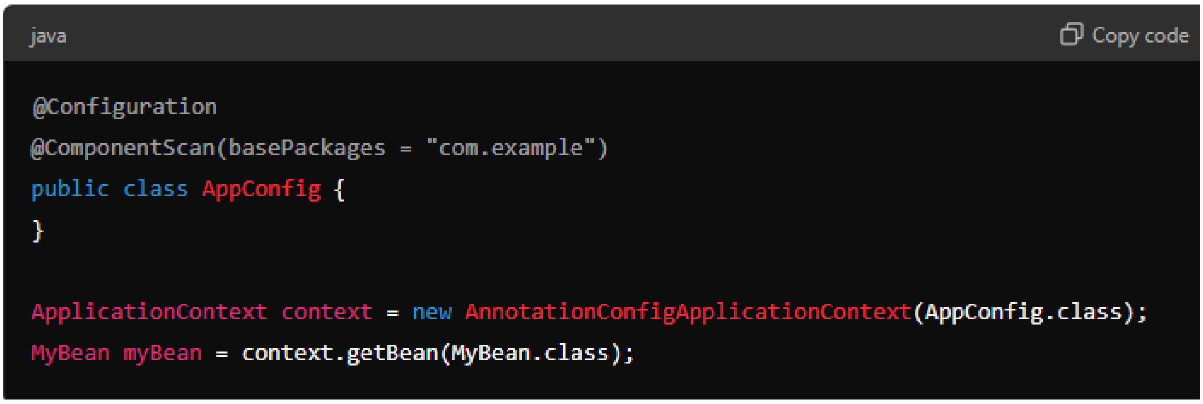
Loads context definition from an XML file located in the file system.



```java
ApplicationContext context = new FileSystemXmlApplicationContext("path/to/applicationConte
MyBean myBean = context.getBean(MyBean.class);
```

c. AnnotationConfigApplicationContext

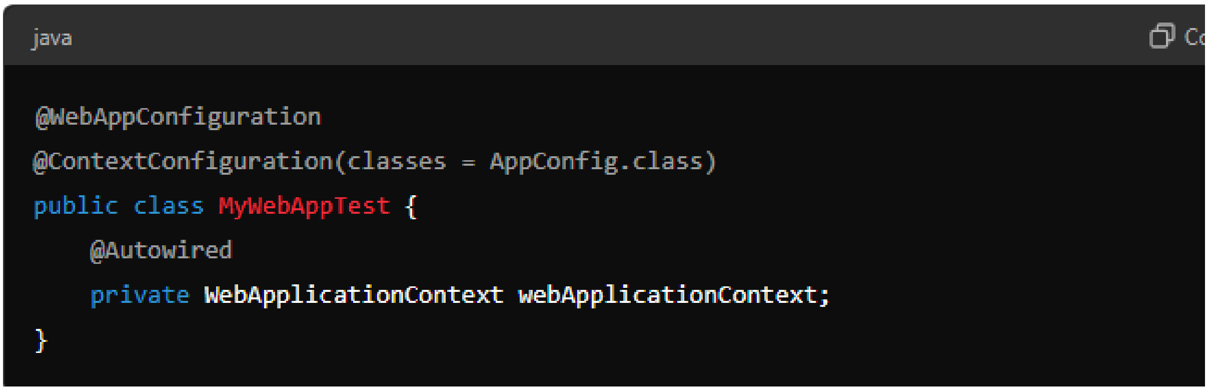
Loads context definition from annotated classes.



```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
MyBean myBean = context.getBean(MyBean.class);
```

d. WebApplicationContext

A specialized version of `ApplicationContext` for web applications. It integrates with the lifecycle of a web application and provides convenient access to Spring's Web MVC infrastructure.



```java
@WebAppConfiguration
@ContextConfiguration(classes = AppConfig.class)
public class MyWebAppTest {
    @Autowired
    private WebApplicationContext webApplicationContext;
}
```

Singleton Bean

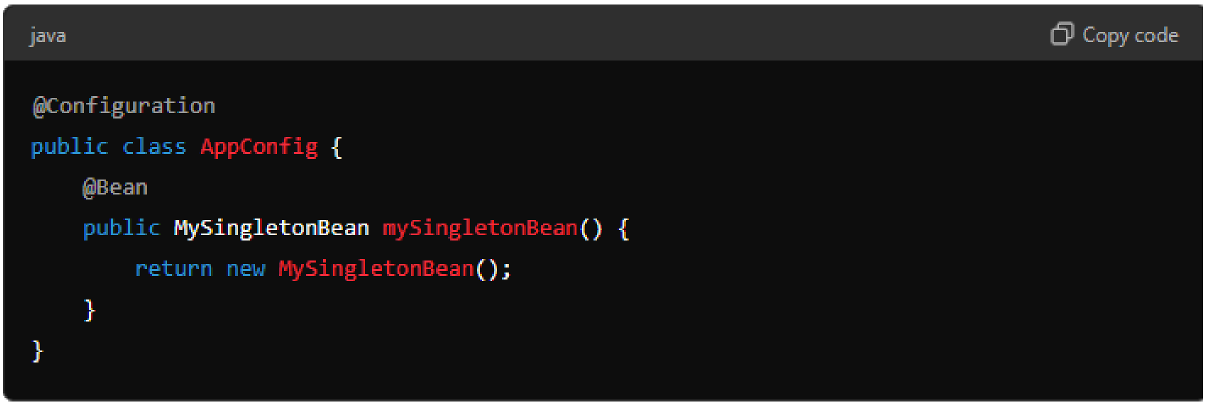
A **singleton** bean is the default scope in Spring. This means that there will be only one instance of the bean created per Spring IoC container. The same instance is shared across the entire application, and it is created when the application context is initialized.

Characteristics:

- **Single Instance:** Only one instance of the bean is created and used throughout the application.
- **Shared State:** All components that depend on the singleton bean share the same instance, which can lead to shared state.
- **Memory Efficiency:** Singleton beans are memory-efficient since only one instance is created.

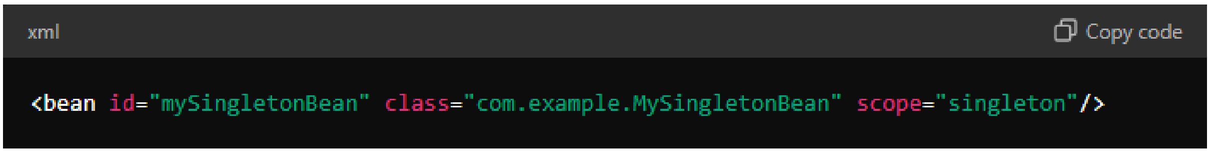
Example:

Bean Definition in Java Configuration:



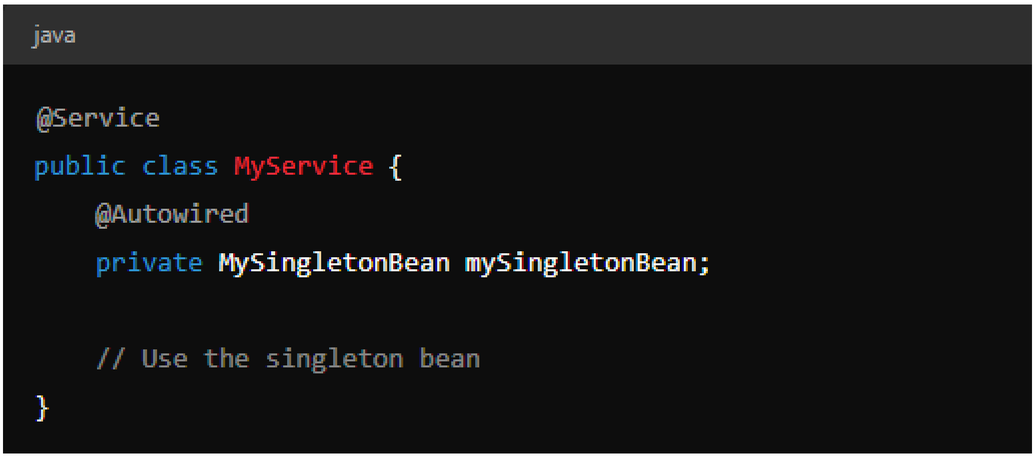
```java
@Configuration
public class AppConfig {
    @Bean
    public MySingletonBean mySingletonBean() {
        return new MySingletonBean();
    }
}
```

Bean Definition in XML Configuration:



```xml
<bean id="mySingletonBean" class="com.example.MySingletonBean" scope="singleton"/>
```

Usage:



```java
@Service
public class MyService {
    @Autowired
    private MySingletonBean mySingletonBean;

    // Use the singleton bean
}
```

Prototype Bean

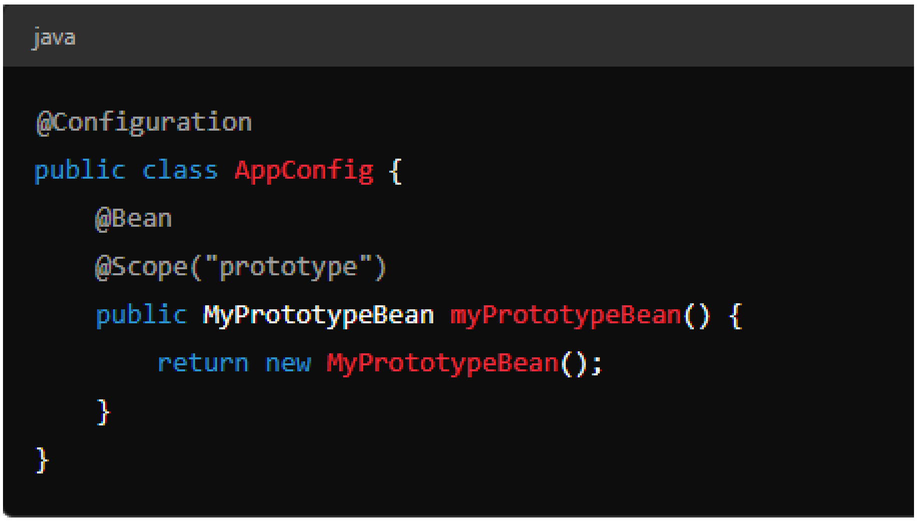
A **prototype** bean is created each time it is requested from the Spring container. This means that a new instance of the bean is created for every call to `getBean()`, resulting in multiple instances of the bean within the same application context.

Characteristics:

- **Multiple Instances:** A new instance is created every time the bean is requested.
- **No Shared State:** Each instance is independent, so no shared state exists between different instances.
- **Less Memory Efficient:** More memory and resources are consumed as multiple instances are created.

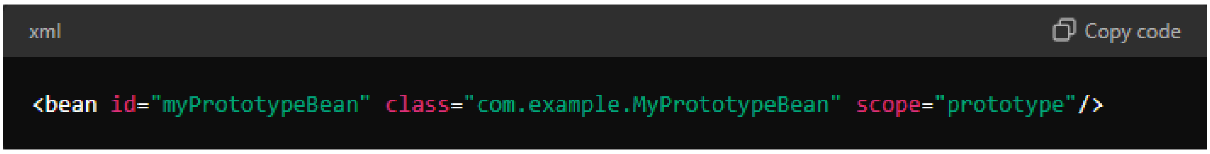
Example:

Bean Definition in Java Configuration:



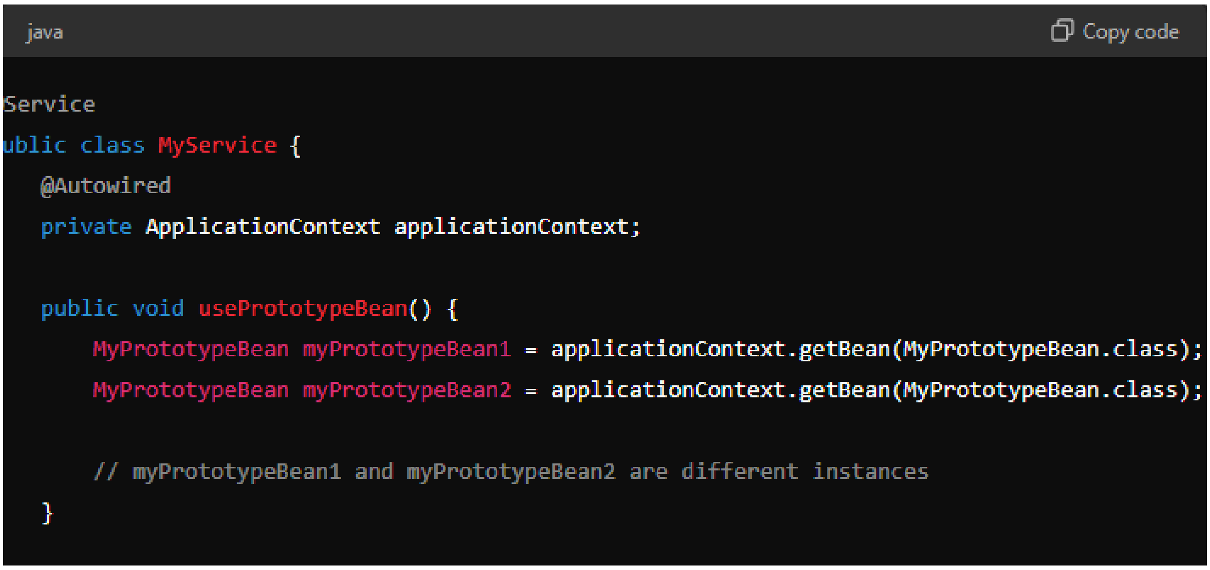
```java
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public MyPrototypeBean myPrototypeBean() {
        return new MyPrototypeBean();
    }
}
```

Bean Definition in XML Configuration:



```xml
<bean id="myPrototypeBean" class="com.example.MyPrototypeBean" scope="prototype"/>
```

Usage:



```java
Service
ublic class MyService {
   @Autowired
   private ApplicationContext applicationContext;

   public void usePrototypeBean() {
       MyPrototypeBean myPrototypeBean1 = applicationContext.getBean(MyPrototypeBean.class);
       MyPrototypeBean myPrototypeBean2 = applicationContext.getBean(MyPrototypeBean.class);

       // myPrototypeBean1 and myPrototypeBean2 are different instances
   }
```

Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules. For example- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore repeating the code at every method, define the functionality in a common class and control were to apply that functionality in the whole application. pring uses **proxy based mechanism** i.e. it creates a proxy Object which will wrap around the original object and will take up the advice which is relevant to the method call. Proxy objects can be created either manually through proxy factory bean or through auto proxy configuration in the XML file and get destroyed when the execution completes. Proxy objects are used to enrich the Original behaviour of the real object.

Common terminologies in AOP:

1. **Aspect:** The class which implements the JEE application cross-cutting concerns(transaction, logger etc) is known as the aspect. It can be normal class configured through XML configuration or through regular classes annotated with @Aspect.
2. **Weaving:** Weaving in Spring AOP is the process of applying aspects to target objects at specified join points, primarily done through proxy objects created at runtime. This allows Spring to provide AOP capabilities in a flexible and non-intrusive manner, enabling the separation of cross-cutting concerns from core business logic.
3. **Advice:** The job which is meant to be done by an Aspect or it can be defined as the action taken by the Aspect at a particular point. There are five types of Advice namely: Before, After, Around, AfterThrowing and AfterReturning. Let’s have a brief discussion about all the five types
4. **JoinPoints:** In Aspect-Oriented Programming (AOP), a join point is a specific point in the execution of the program where an aspect can be applied. Join points are typically method executions, but they can also include other points such as field access or exception handling. Join points are essentially the points in the program flow where advice (additional behavior) can be inserted.
5. **PointCut**: In Aspect-Oriented Programming (AOP), a pointcut is a set of one or more join points where advice should be applied. In Spring AOP, pointcuts are defined using expressions that match specific join points, allowing the advice to be applied selectively based on the pointcut definition.

Spring Expression Language (SpEL) is a powerful feature in the Spring Framework that allows you to query and manipulate objects at runtime using expressions. SpEL provides a way to work with Java objects, evaluate expressions, and perform operations dynamically in a Spring application.

Key Features of SpEL

1. **Expression Evaluation**: Evaluate expressions in a Spring context.
2. **Access to Bean Properties**: Access and manipulate bean properties and methods.
3. **Object Manipulation**: Perform operations on objects, such as invoking methods and accessing fields.
4. **Conditional Logic**: Use conditional logic and control flow within expressions.
5. **Integration with Spring Annotations**: Use SpEL within annotations like `@Value`, `@Condition`, and `@Cacheable`.

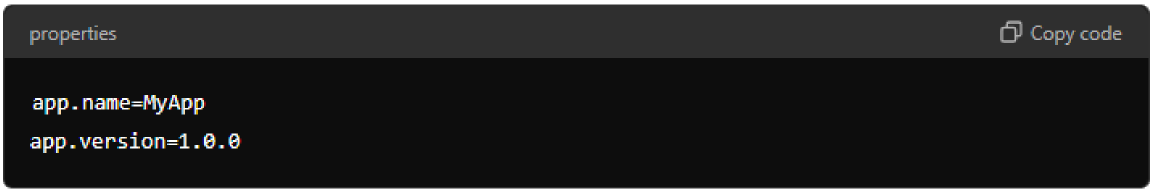
Common Use Cases

1. **Property Injection**: Inject values from properties files or environment variables into Spring beans.
2. **Conditional Logic**: Control bean instantiation and behavior based on dynamic conditions.
3. **Dynamic Method Invocation**: Invoke methods and access properties dynamically.

@Value Annotation

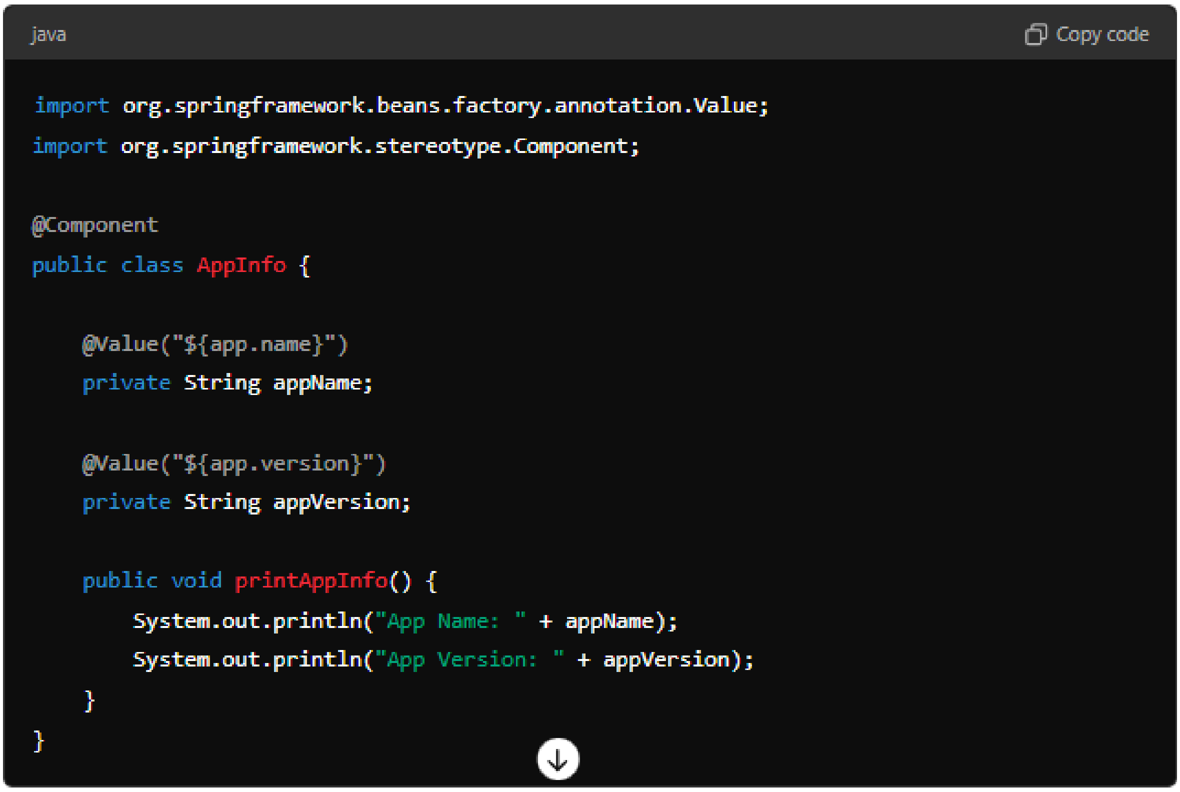
The `@Value` annotation is used for injecting values from external sources, providing more flexibility and ease of configuration.

application.properties



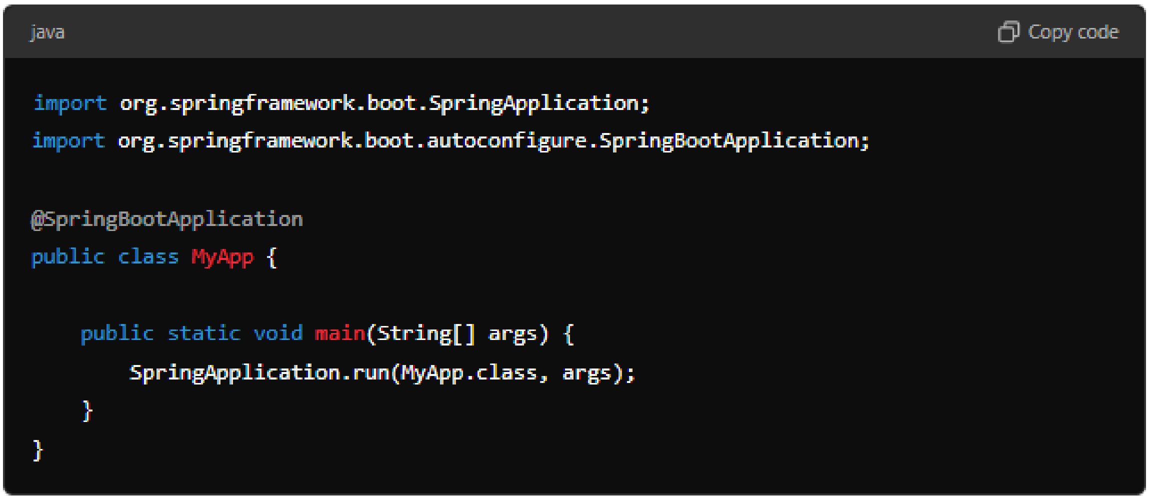
```properties
app.name=MyApp
app.version=1.0.0
```

AppInfo Class

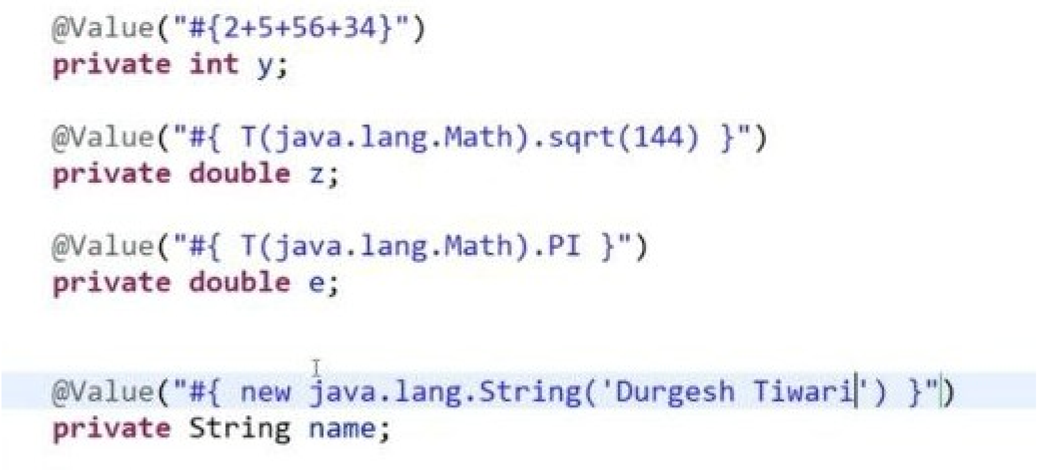


```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppInfo {

    @Value("${app.name}")
    private String appName;

    @Value("${app.version}")
    private String appVersion;

    public void printAppInfo() {
        System.out.println("App Name: " + appName);
        System.out.println("App Version: " + appVersion);
    }
}
```

Main Application Class



```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApp {

    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```



```java
@Value("#{2+5+56+34}")
private int y;

@Value("#{ T(java.lang.Math).sqrt(144) }")
private double z;

@Value("#{ T(java.lang.Math).PI }")
private double e;


@Value("#{ new java.lang.String('Durgesh Tiwari') }")
private String name;
```