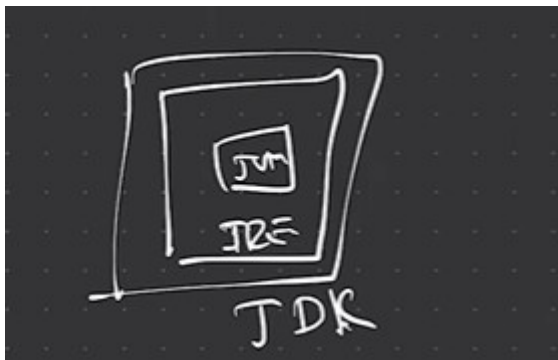
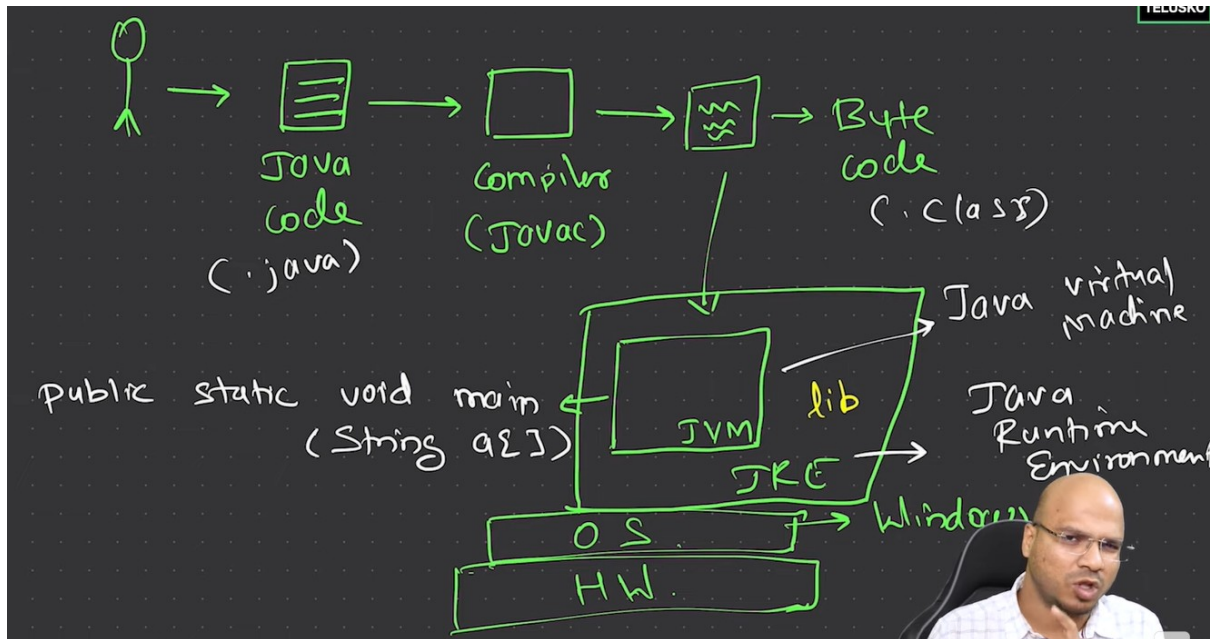


Java Notes



>> 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, Architecture-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 **ArrayIndexOutOfBoundsException** 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. If 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);
}
```

```
3      public static void main(String a[])
4      {
5          // literals
6
7          int num1 = 10_00_00_000;
8          System.out.println(num1);
9
10
11      }
12
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
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 %
```

```
1 Hello.java
2  {
3      public static void main(String a[])
4      {
5          // literals
6
7          double num1 = 12e10;
8          System.out.println(num1);
9
10
11      }
12
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
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 %
```

```
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 conversion 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 conversion 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

What's new in Java Switch | Switch Statement and Expression

```
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";
        };

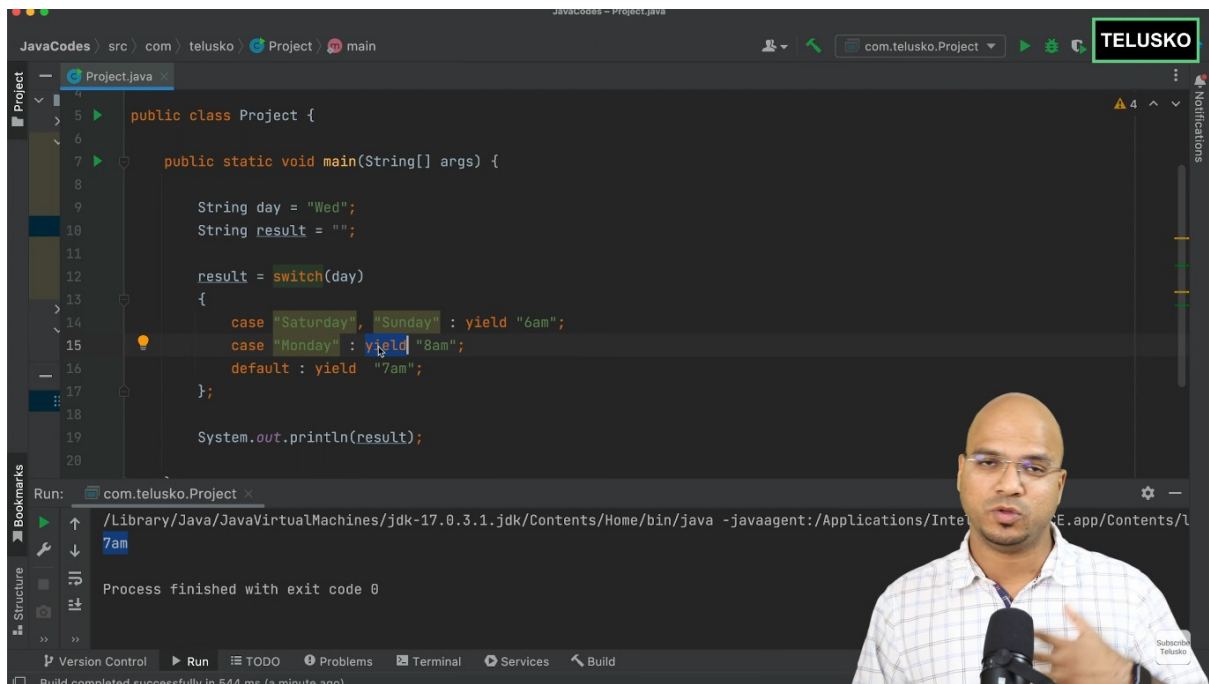
        System.out.println(result);
    }
}
```

Run: com.telusko.Project

/Library/Java/JavaVirtualMachines/jdk-17.0.3.1.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar -Dcom.telusko.Project

6am

Process finished with exit code 0



>>

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.

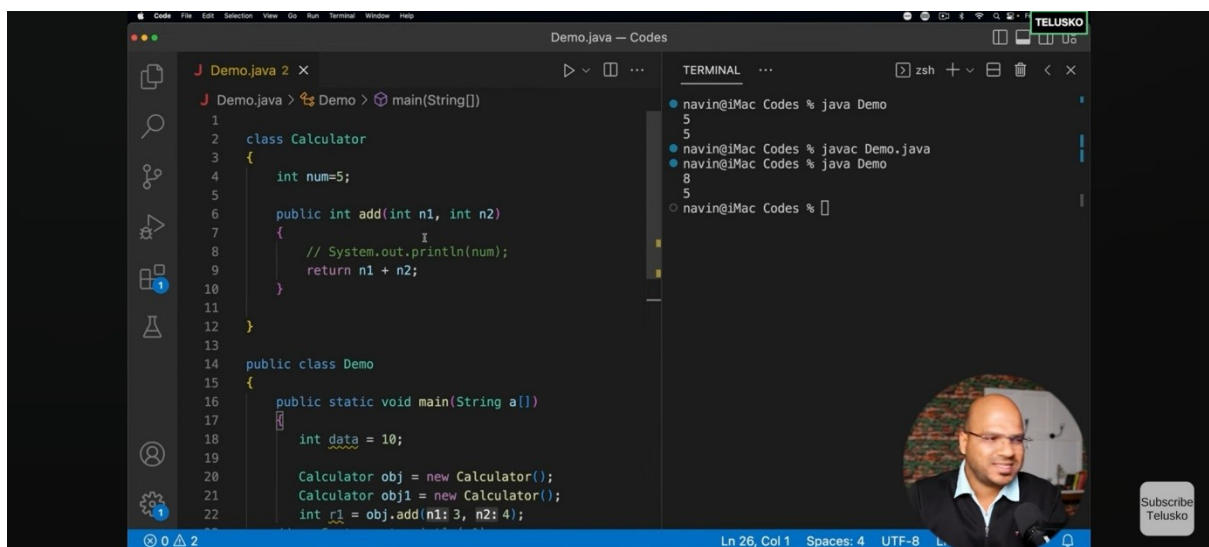
So the correct order is:

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)

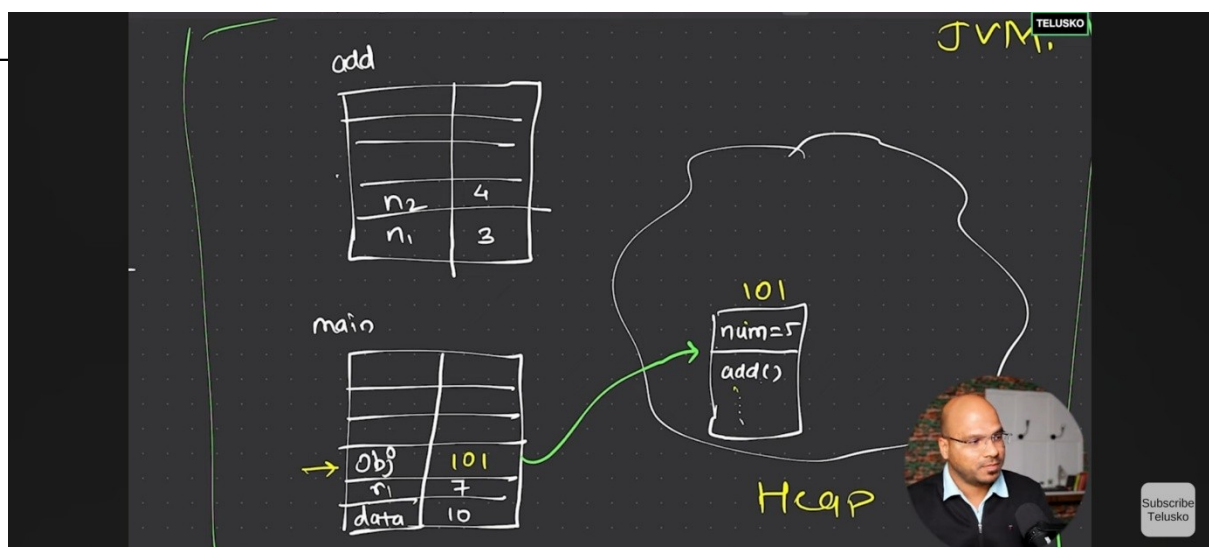
This revised sequence correctly reflects how the JVM manages memory when the statement `Car c = new Car();` is executed.



The screenshot shows an IDE with a file named `Demo.java`. The code defines a `Calculator` class with an `add` method and a `Demo` class with a `main` method. The terminal shows the execution of `java Demo`, which outputs `5`, `5`, `8`, and `5` on separate lines. A small video inset of a man is visible in the bottom right corner.

```
1 class Calculator
2 {
3     int num=5;
4
5     public int add(int n1, int n2)
6     {
7         // System.out.println(num);
8         return n1 + n2;
9     }
10 }
11
12
13
14 public class Demo
15 {
16     public static void main(String a[])
17     {
18         int data = 10;
19
20         Calculator obj = new Calculator();
21         Calculator obj1 = new Calculator();
22         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;
    }

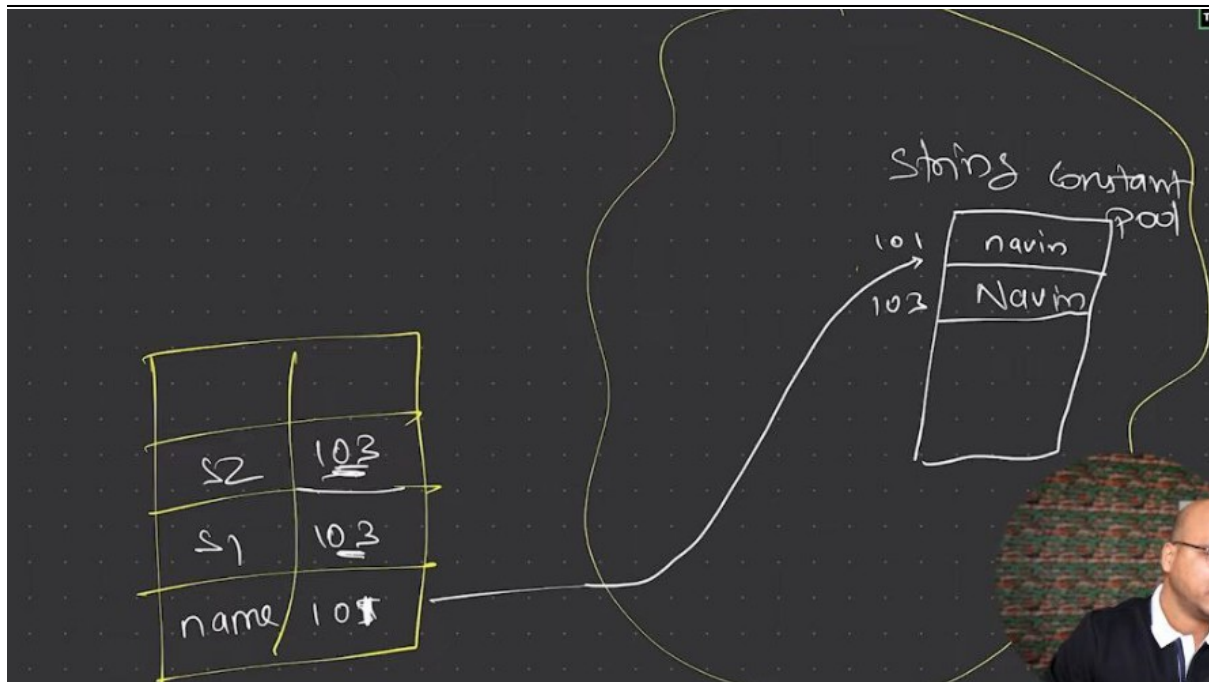
    public void method() {
        // Local variable
        int localVar = 10;
        System.out.println("Instance
Var: " + instanceVar);
        System.out.println("Static
Var: " + staticVar);
        System.out.println("Local
Var: " + localVar);
    }

    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, Runnable
// variable and method - marks, show()
// constants - PIE, BRAND

// showMyMarks()
// MyData

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

>>

Advantages of Using List Reference while creating a ArrayList object:

Flexibility:

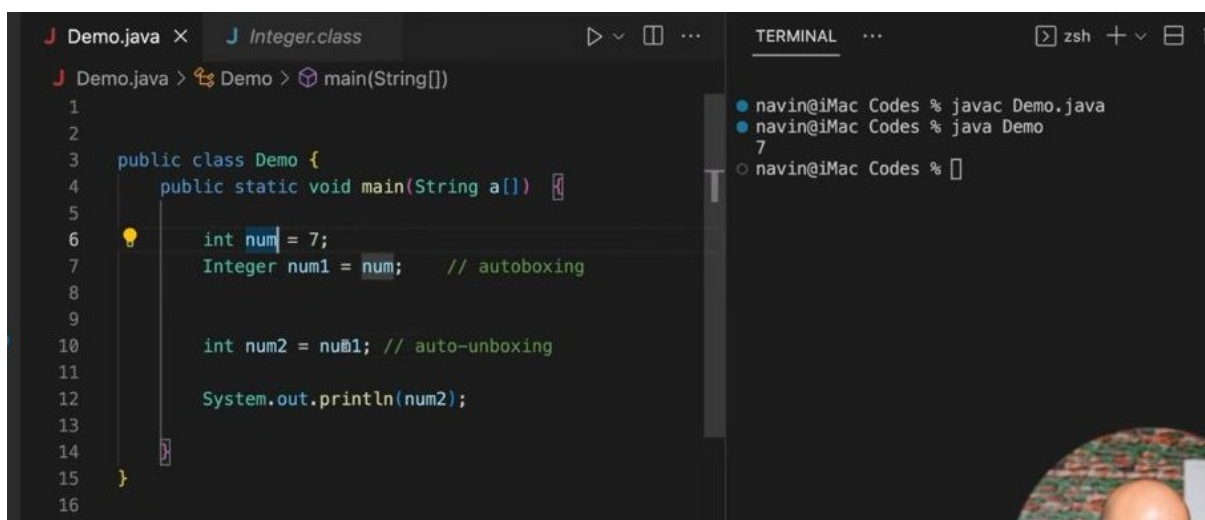
If you use a List reference, you can easily change the implementation to another List type (e.g., LinkedList) without changing the code that uses the List.
`List<String> list = new LinkedList<>();`

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.



The screenshot shows an IDE with two tabs: `Demo.java` and `Integer.class`. The `Demo.java` file contains the following code:

```
1  
2  
3 public class Demo {  
4     public static void main(String a[]) {  
5  
6         int num = 7;  
7         Integer num1 = num; // autoboxing  
8  
9  
10        int num2 = num1; // auto-unboxing  
11  
12        System.out.println(num2);  
13  
14    }  
15  
16 }
```

The terminal window on the right shows the following commands and output:

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

```
Demo.java x Integer.class
J Demo.java > Demo > main(String[])
1
2
3 public class Demo {
4     public static void main(String a[]) {
5
6         int num = 7;
7         Integer num1 = num; // autoboxing
8
9
10        int num2 = num1.intValue(); // unboxing
11
12        System.out.println(num2);
13
14    }
15
16
17
```

TERMINAL ... zsh + -

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

```
Demo.java — Codes
J Demo.java x
J Demo.java > Demo > main(String[])
1
2
3 public class Demo {
4     public static void main(String a[]) {
5
6         int num = 7;
7         Integer num1 = num; // autoboxing
8
9
10        System.out.println(num1);
11
12    }
13
14
15
```

TERMINAL ... zsh + -

```
navin@iMac Codes % javac Demo.java
^[[ADemo.java:7: warning: [removal] Integer(int) in
Integer has been deprecated and marked for removal
    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 %
```

```
Demo.java 1 x
J Demo.java > Demo > main(String[])
1
2
3 public class Demo {
4     public static void main(String a[]) {
5
6         int num = 7;
7         Integer num1 = new Integer(num); // boxing
8
9
10        System.out.println(num1);
11
12    }
13
14
15
```

TERMINAL ... zsh + -

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

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.

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 Copy code

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

```
J Testjava > ...
1  class A{
2      public void show1(){
3          System.out.println(x:"in A show");
4      }
5
6      public void test1(){
7          System.out.println(x:"in test1 parent");
8      }
9  }
10
11 class B extends A{
12     public void show2(){
13         System.out.println(x:"in B show");
14     }
15
16     public void test1(){
17         System.out.println(x:"in test1 child");
18     }
19 }
20
21 public class Test {
22     Run | Debug
23     public static void main(String[] args) {
24         A obj = (A) new B(); //by default (A) is there even if it is not mentioned
25         obj.show1();
26         obj.test1();
27
28         B obj1 = (B) obj;
29         obj1.show2();
30     }
31 }
```

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

E:\Study\java-code>java Test

in A show

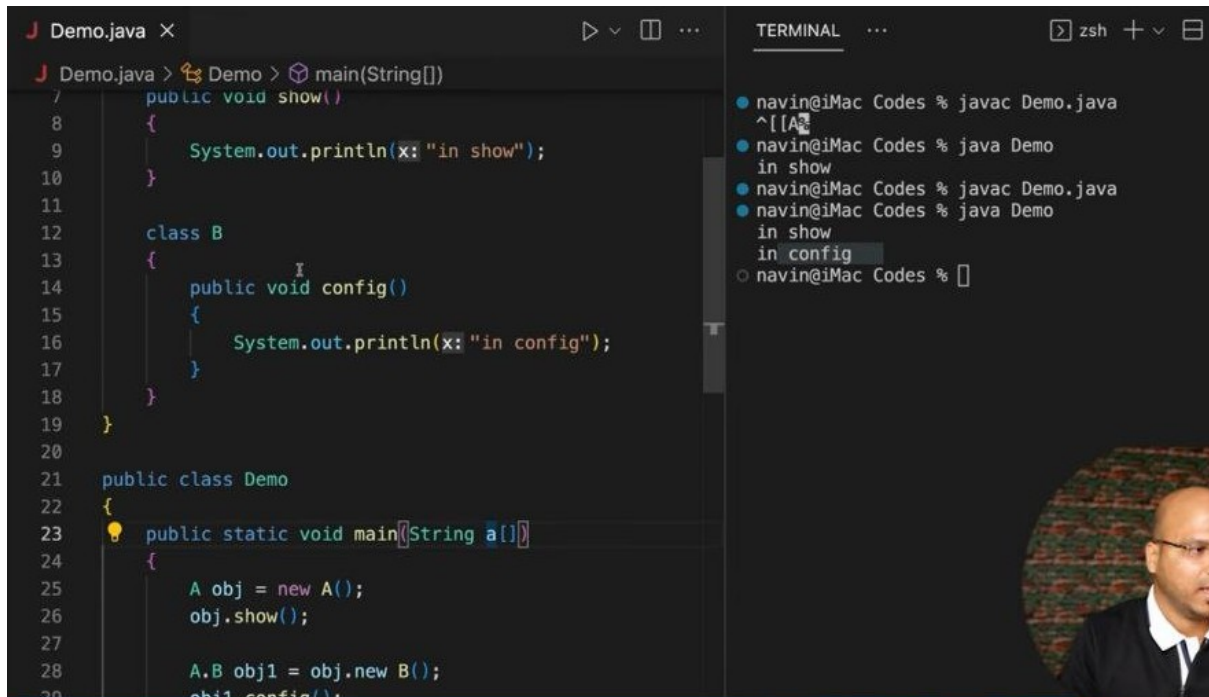
in test1 child

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

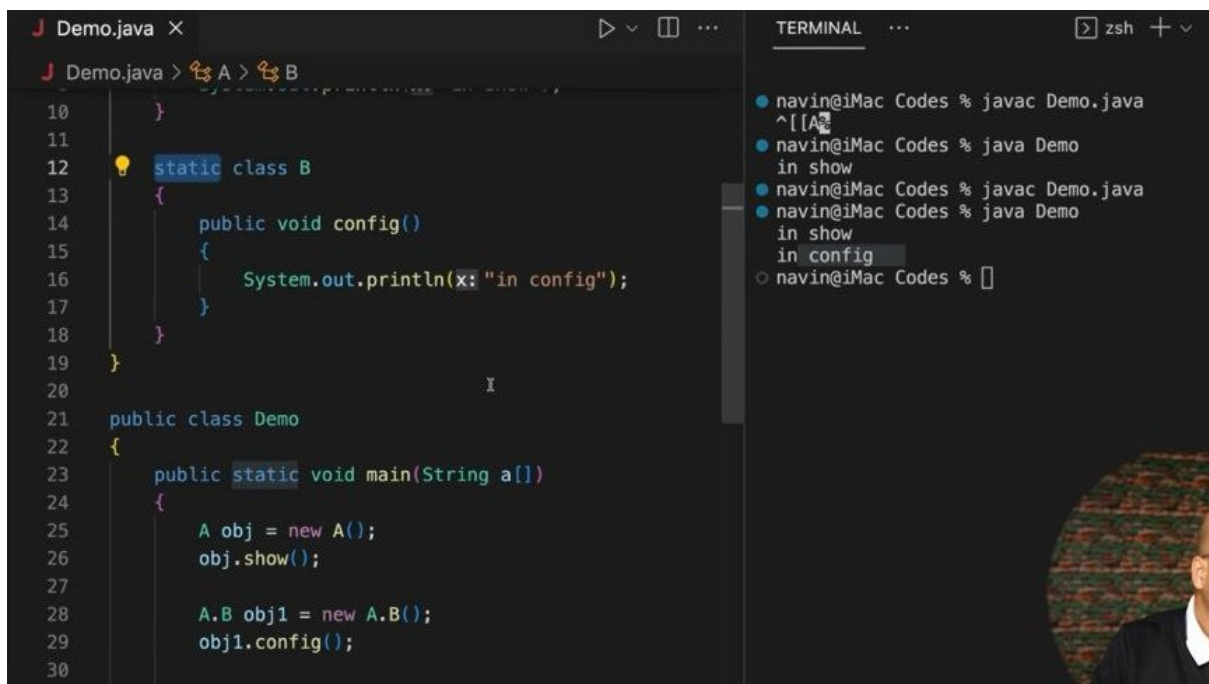
>>Inner Class



```
Demo.java X
J Demo.java > Demo > main(String[])
7   public void show()
8   {
9       System.out.println(x: "in show");
10  }
11
12  class B
13  {
14      public void config()
15      {
16          System.out.println(x: "in config");
17      }
18  }
19
20
21  public class Demo
22  {
23      public static void main(String a[])
24      {
25          A obj = new A();
26          obj.show();
27
28          A.B obj1 = obj.new B();
29          obj1.config();
30      }
31  }
```

TERMINAL ... zsh + v

- 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
- in config
- navin@iMac Codes %

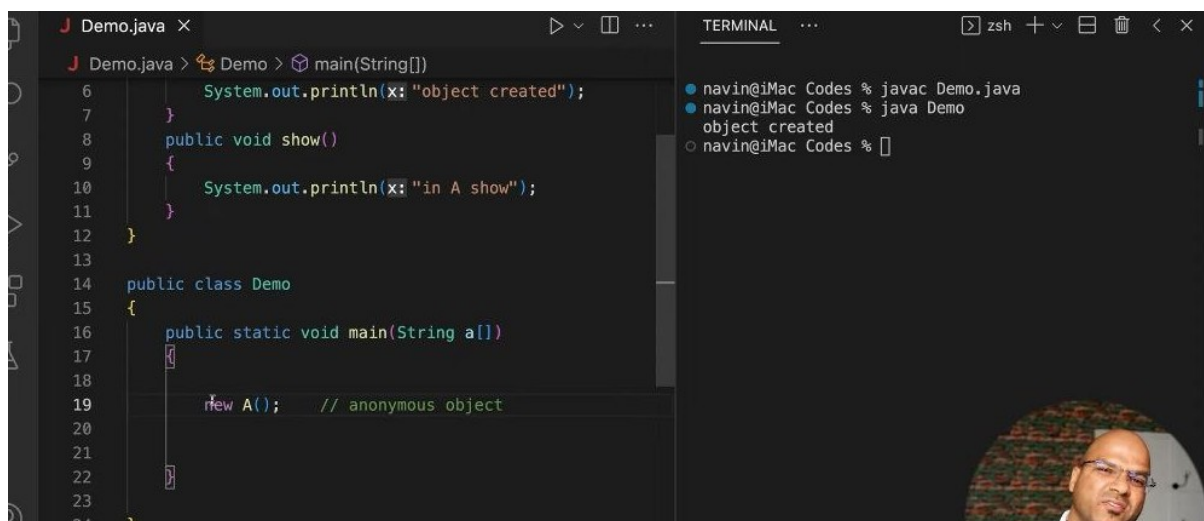


```
Demo.java X
J Demo.java > A > B
10  }
11
12  static class B
13  {
14      public void config()
15      {
16          System.out.println(x: "in config");
17      }
18  }
19
20
21  public class Demo
22  {
23      public static void main(String a[])
24      {
25          A obj = new A();
26          obj.show();
27
28          A.B obj1 = new A.B();
29          obj1.config();
30      }
31  }
```

TERMINAL ... zsh + v

- 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
- in config
- navin@iMac Codes %

>>Anonymous object

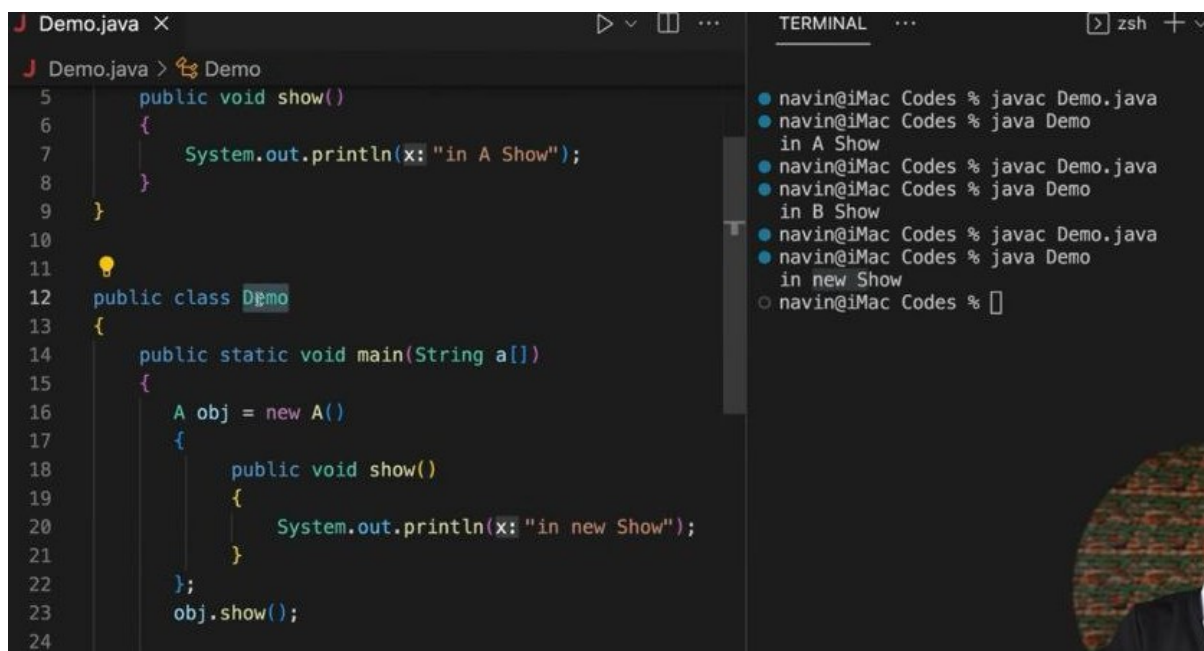


The screenshot shows an IDE with a file named `Demo.java`. The code defines a `main` method in a `Demo` class that creates an anonymous object of type `A` and calls its `show` method. The terminal output shows the compilation and execution of the code, resulting in the message "object created".

```
Demo.java > Demo > main(String[])
6      System.out.println(x: "object created");
7  }
8  public void show()
9  {
10     System.out.println(x: "in A show");
11 }
12
13
14 public class Demo
15 {
16     public static void main(String a[])
17     {
18         new A(); // anonymous object
19     }
20 }
21
22
23
24
```

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

>>anonymous class/anonymous inner class

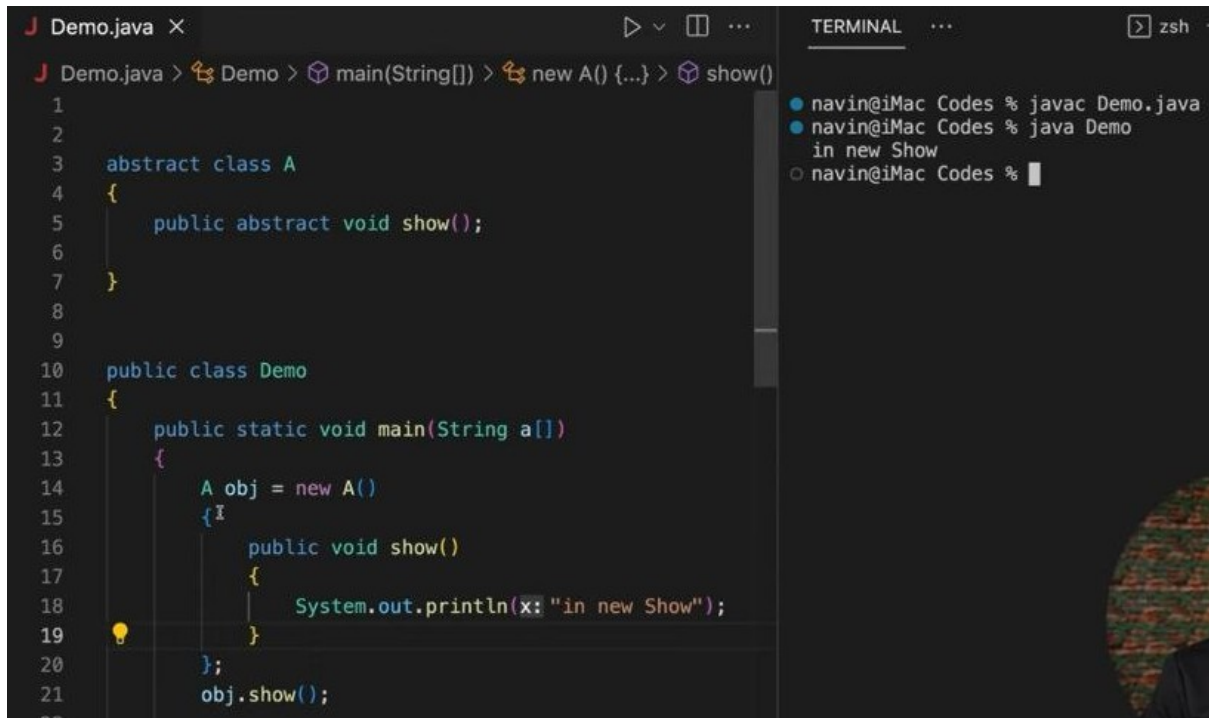


The screenshot shows an IDE with a file named `Demo.java`. The code defines a `main` method in a `Demo` class that creates an anonymous inner class `A` and calls its `show` method. The terminal output shows the compilation and execution of the code, resulting in the message "in new Show".

```
Demo.java > Demo
5  public void show()
6  {
7      System.out.println(x: "in A Show");
8  }
9  }
10
11
12 public class Demo
13 {
14     public static void main(String a[])
15     {
16         A obj = new A()
17         {
18             public void show()
19             {
20                 System.out.println(x: "in new Show");
21             }
22         };
23         obj.show();
24     }
25 }
```

```
TERMINAL
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



The screenshot shows an IDE with a file named `Demo.java`. The code defines an abstract class `A` with an abstract method `show()`. The `Demo` class has a `main` method that creates an anonymous inner class `A` and calls its `show()` method. The terminal window on the right shows the compilation and execution of the code, with the output `in new Show`.

```
1
2
3 abstract class A
4 {
5     public abstract void show();
6 }
7
8
9
10 public class Demo
11 {
12     public static void main(String a[])
13     {
14         A obj = new A()
15         {
16             public void show()
17             {
18                 System.out.println(x: "in new Show");
19             }
20         };
21         obj.show();
22     }
23 }
```

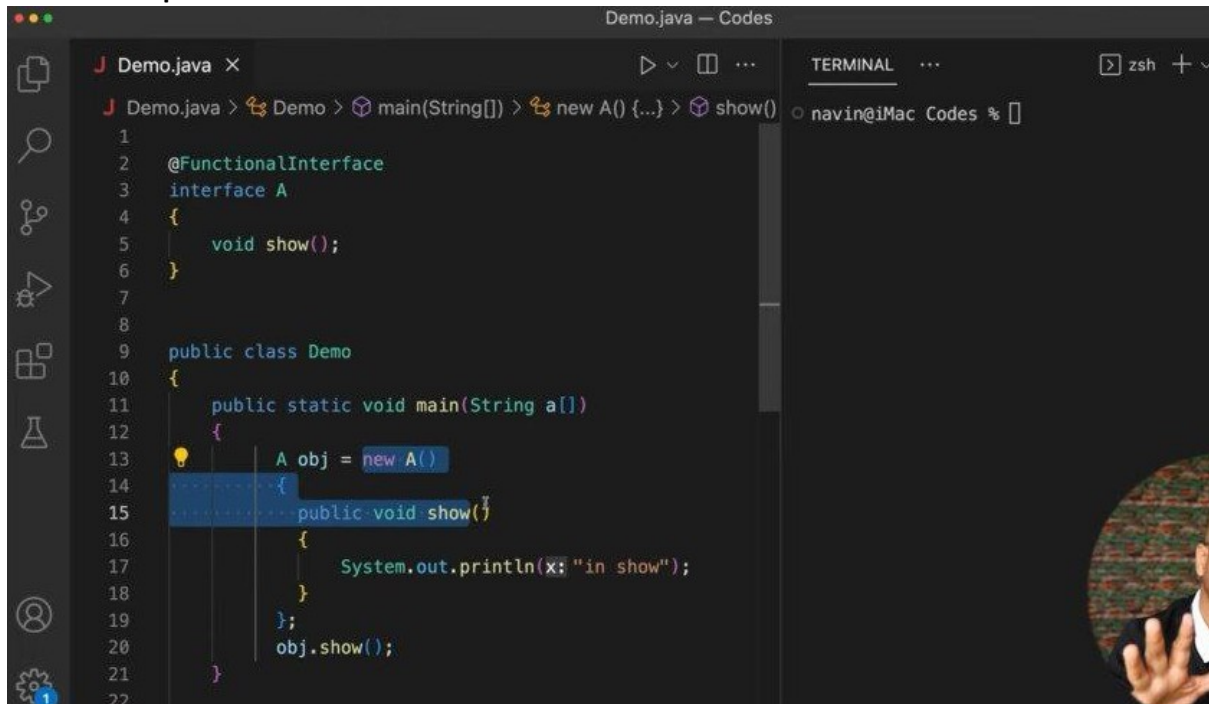
```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
in new Show
navin@iMac Codes %
```

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

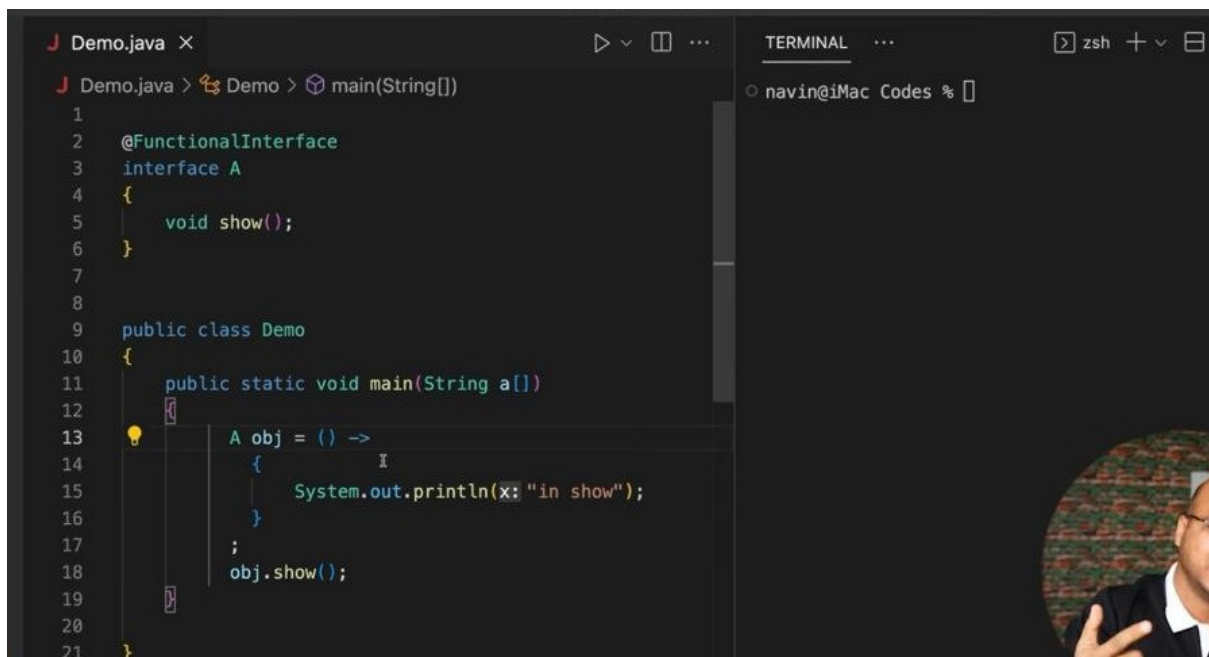
>> If you want to prove that the default constructor is getting invoked without explicitly adding any print statements in the constructor, you can demonstrate it indirectly by observing the initialization behavior of instance variables. Constructor is responsible for initialization of the instance variables

>> 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

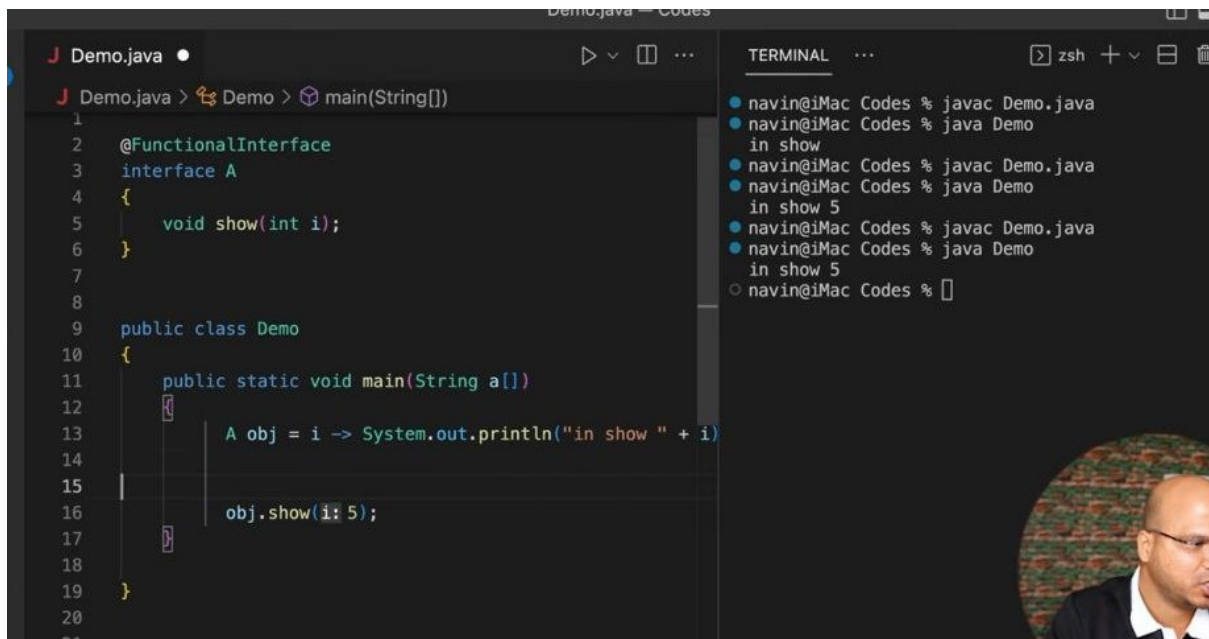
>>lambda expression



```
1
2 @FunctionalInterface
3 interface A
4 {
5     void show();
6 }
7
8
9 public class Demo
10 {
11     public static void main(String a[])
12     {
13         A obj = new A()
14         {
15             public void show()
16             {
17                 System.out.println(x; "in show");
18             }
19         };
20         obj.show();
21     }
22 }
```



```
1
2 @FunctionalInterface
3 interface A
4 {
5     void show();
6 }
7
8
9 public class Demo
10 {
11     public static void main(String a[])
12     {
13         A obj = () ->
14         {
15             System.out.println(x; "in show");
16         }
17         ;
18         obj.show();
19     }
20 }
21 }
```



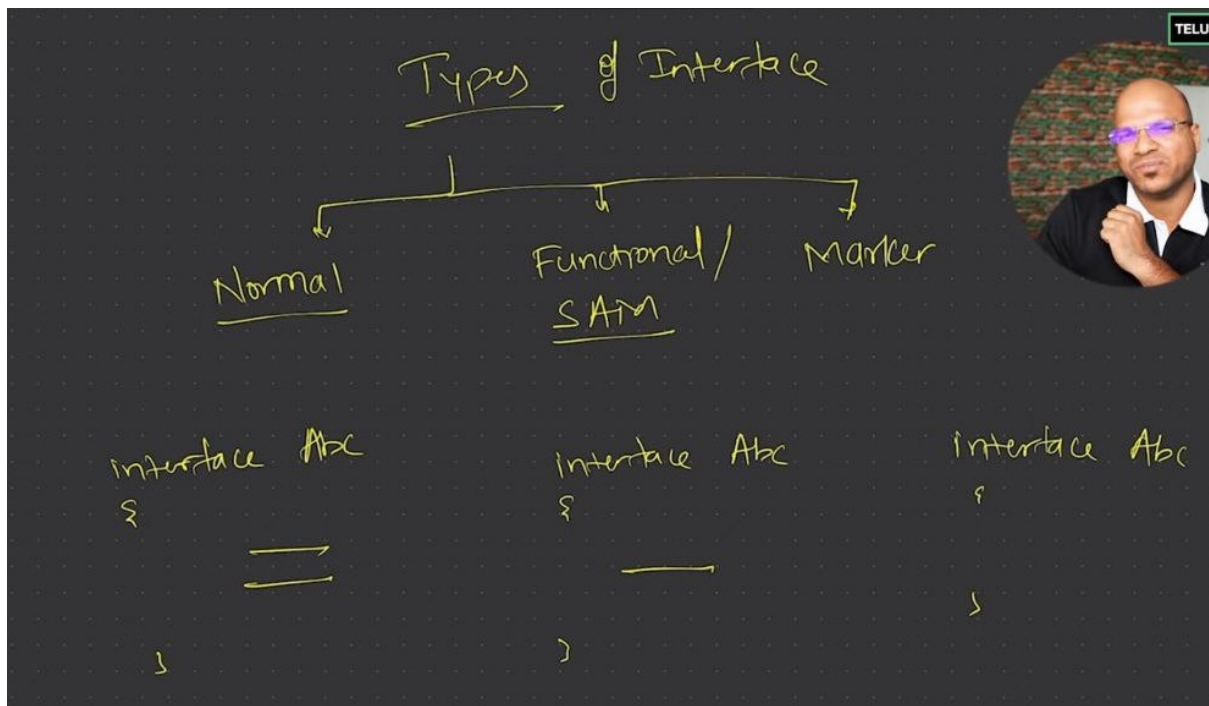
```
Demo.java •
J Demo.java > Demo > main(String[])
1
2 @FunctionalInterface
3 interface A
4 {
5     void show(int i);
6 }
7
8
9 public class Demo
10 {
11     public static void main(String a[])
12     {
13         A obj = i -> System.out.println("in show " + i);
14
15         obj.show(5);
16     }
17 }
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
```

```
Demo.java •
J Demo.java > Demo > main(String[])
6
7
8
9 public class Demo
10 {
11     public static void main(String a[])
12     {
13         A obj = (i,j) -> i+j;
14
15         |
16         int result = obj.add(i: 5, j: 4);
17         System.out.println(result);
18     }
19
20 }
21
22
```

TERMINAL ... zsh + -

- 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

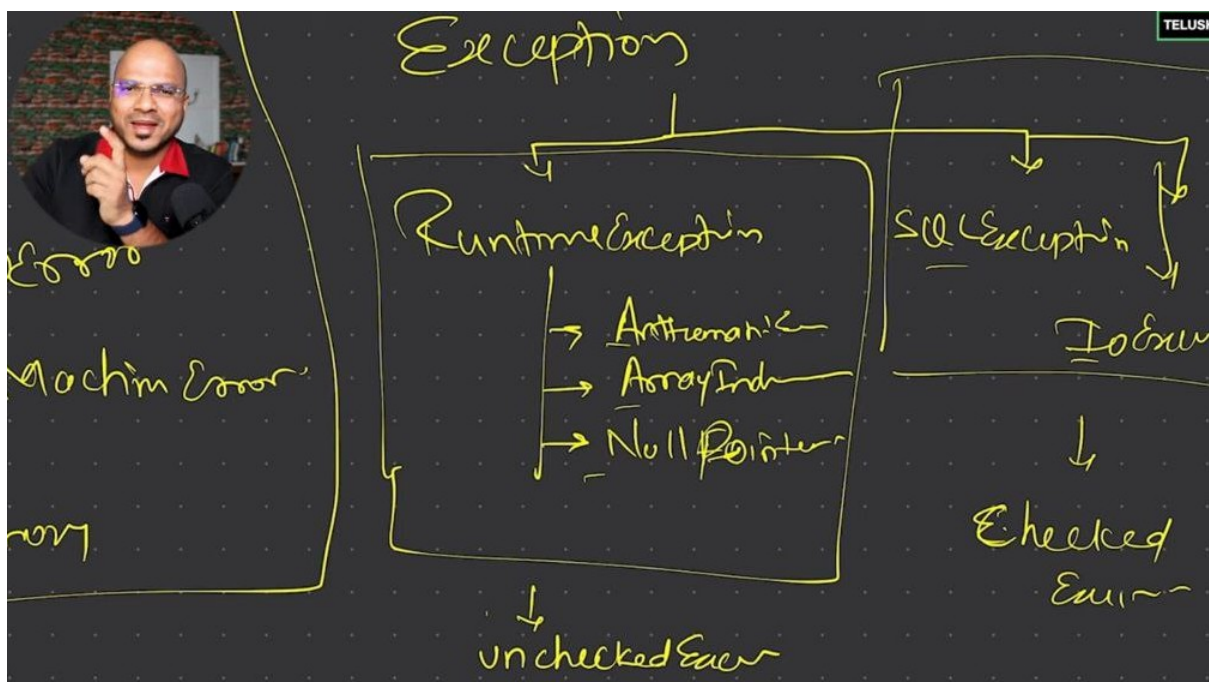
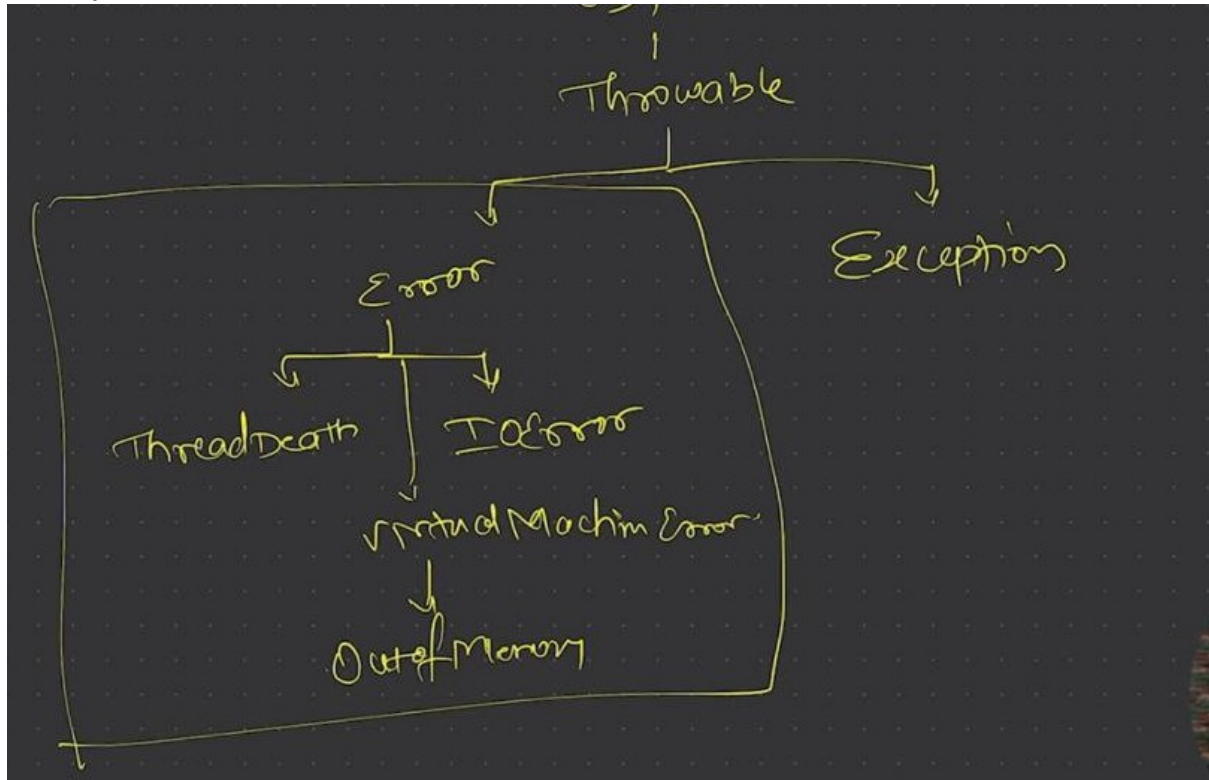
- Purpose: Indicates that a class can be used to create remote objects in Java RMI (Remote Method Invocation).
- Use Case: Defining objects whose methods can be called from another JVM.
- Example:

```
java

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

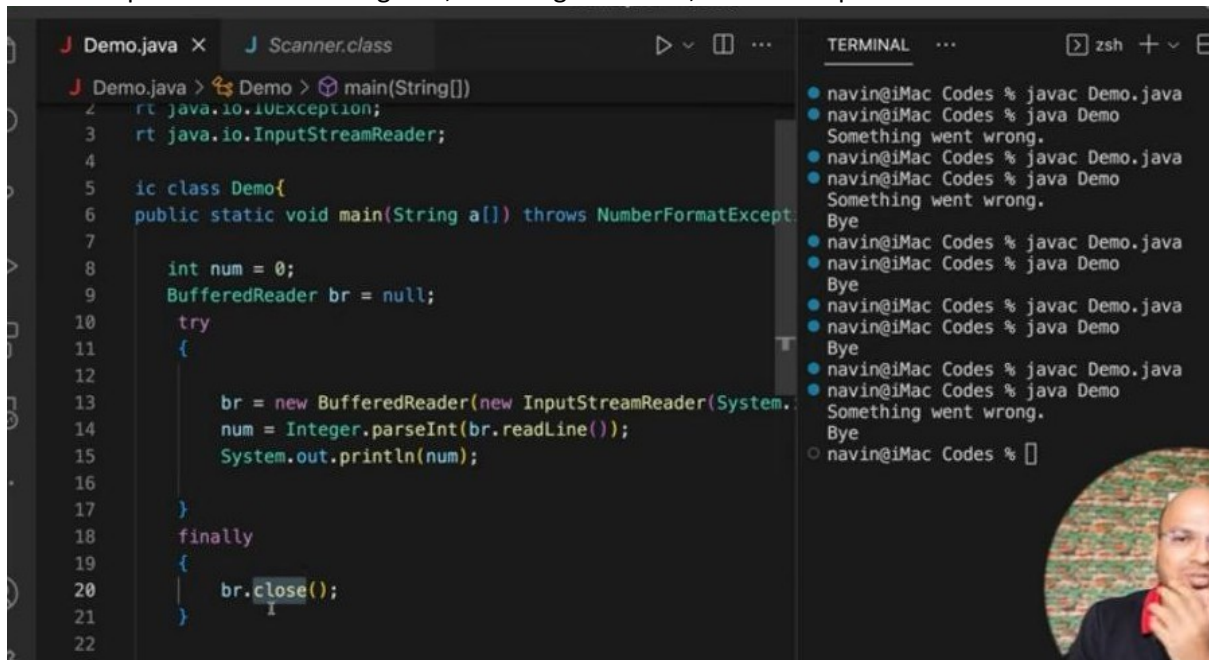
public interface MyRemoteService extends Remote {
    String sayHello() throws RemoteException;
}
```

>>Exception



>>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.



The screenshot shows an IDE with two tabs: `Demo.java` and `Scanner.class`. The `Demo.java` file contains the following code:

```
1 Demo.java > Demo > main(String[])
2   rt java.io.IOException;
3   rt java.io.InputStreamReader;
4
5   ic class Demo{
6   public static void main(String a[]) throws NumberFormatException{
7
8       int num = 0;
9       BufferedReader br = null;
10      try
11      {
12
13          br = new BufferedReader(new InputStreamReader(System.in));
14          num = Integer.parseInt(br.readLine());
15          System.out.println(num);
16
17      }
18      finally
19      {
20          br.close();
21      }
22  }
```

The terminal window on the right shows the execution of the program. It displays the output of the program, which is the number entered by the user. The terminal also shows the command prompt and the output of the `javac` and `java` commands. The output of the program is:

```
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
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.

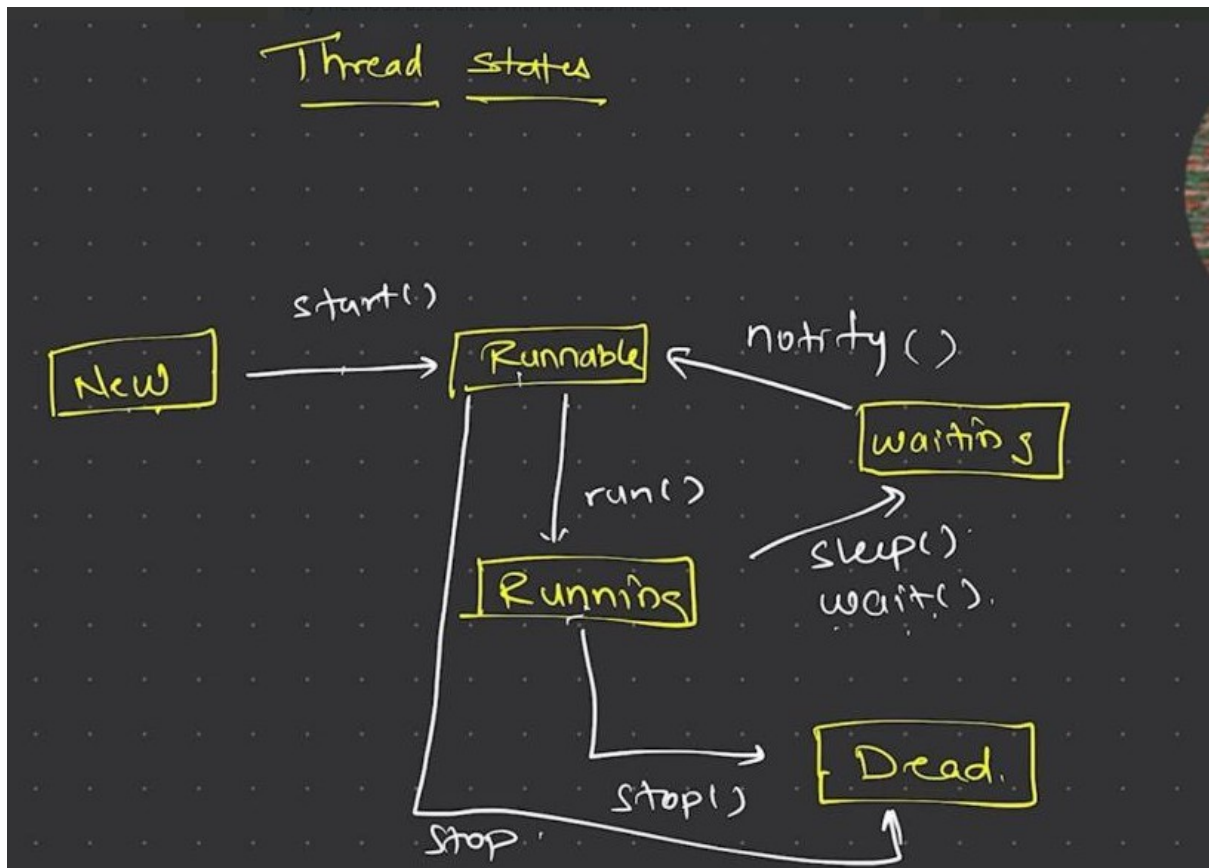
```
java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running.");
    }
}

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.

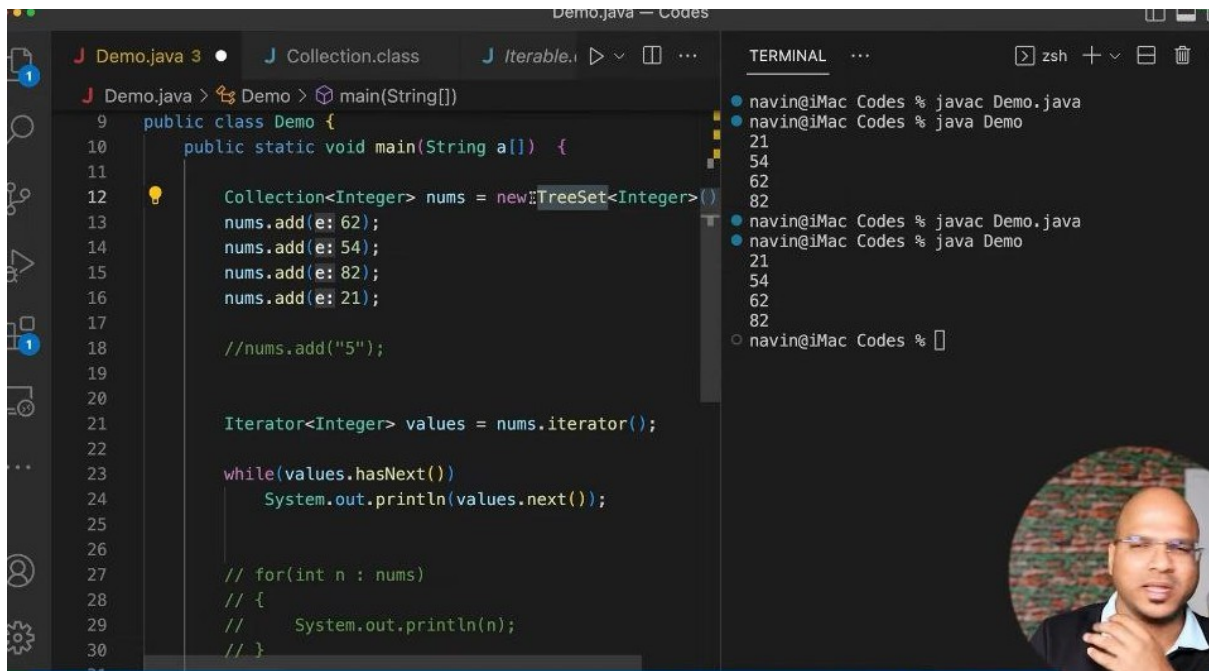
Using threads can improve the performance of a Java application by allowing multiple operations to run concurrently. However, it also introduces complexity, particularly with issues related to synchronization and shared resources



>>

`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()`).

A screenshot of an IDE window titled 'Demo.java - Codes'. The editor shows a Java file 'Demo.java' with the following code:

```
9 public class Demo {
10     public static void main(String a[]) {
11
12         Collection<Integer> nums = new TreeSet<Integer>();
13         nums.add(e: 62);
14         nums.add(e: 54);
15         nums.add(e: 82);
16         nums.add(e: 21);
17
18         //nums.add("5");
19
20
21         Iterator<Integer> values = nums.iterator();
22
23         while(values.hasNext())
24             System.out.println(values.next());
25
26
27         // for(int n : nums)
28         // {
29         //     System.out.println(n);
30         // }
```

The terminal on the right shows the execution of the code:

```
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 %
```

A small circular inset image of a man is visible in the bottom right corner of the IDE window.

>> HashSet 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.

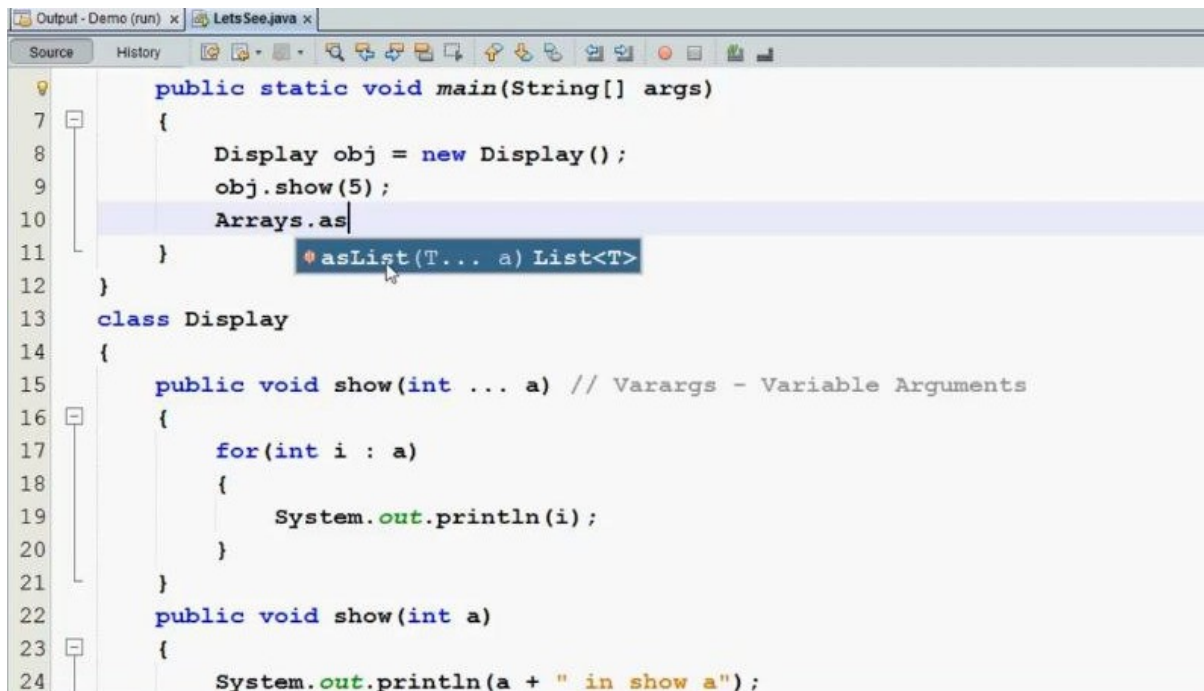
In practice, use `Comparable` when objects have a natural order and `Comparator` when you need multiple ways to compare objects or do not control the source code of the class being compared.

>>varargs

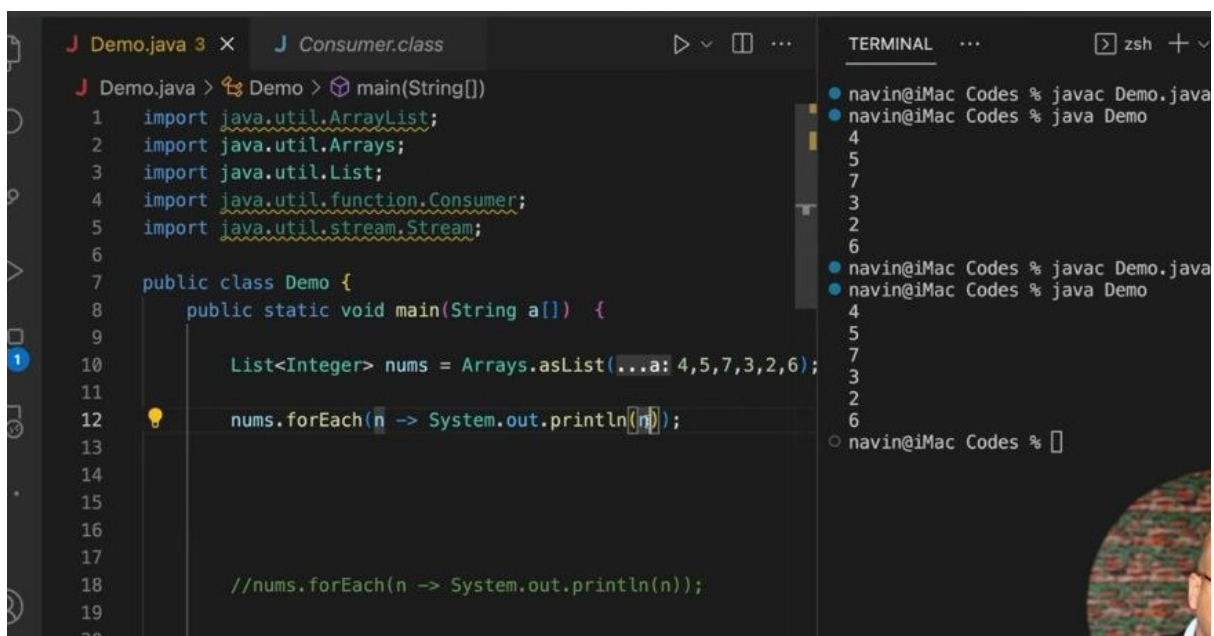
In Java, the term `varargs` refers to a feature that allows a method to accept a variable number of arguments. This is achieved using the ellipsis (`...`) syntax. The method can then handle these

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.



```
Output - Demo (run) x LetsSee.java x
Source History
7 public static void main(String[] args)
8 {
9     Display obj = new Display();
10    obj.show(5);
11    Arrays.asList(T... a) List<T>
12 }
13 class Display
14 {
15     public void show(int ... a) // Varargs - Variable Arguments
16     {
17         for(int i : a)
18         {
19             System.out.println(i);
20         }
21     }
22     public void show(int a)
23     {
24         System.out.println(a + " in show a");
```



```
J Demo.java 3 x J Consumer.class
J Demo.java > Demo > main(String[])
1 import java.util.ArrayList;
2 import java.util.Arrays;
3 import java.util.List;
4 import java.util.function.Consumer;
5 import java.util.stream.Stream;
6
7 public class Demo {
8     public static void main(String a[]) {
9
10        List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);
11
12        nums.forEach(n -> System.out.println(n));
13
14
15
16
17
18        //nums.forEach(n -> System.out.println(n));
19
20    }
21 }
```

```
TERMINAL ... zsh + v
● 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:

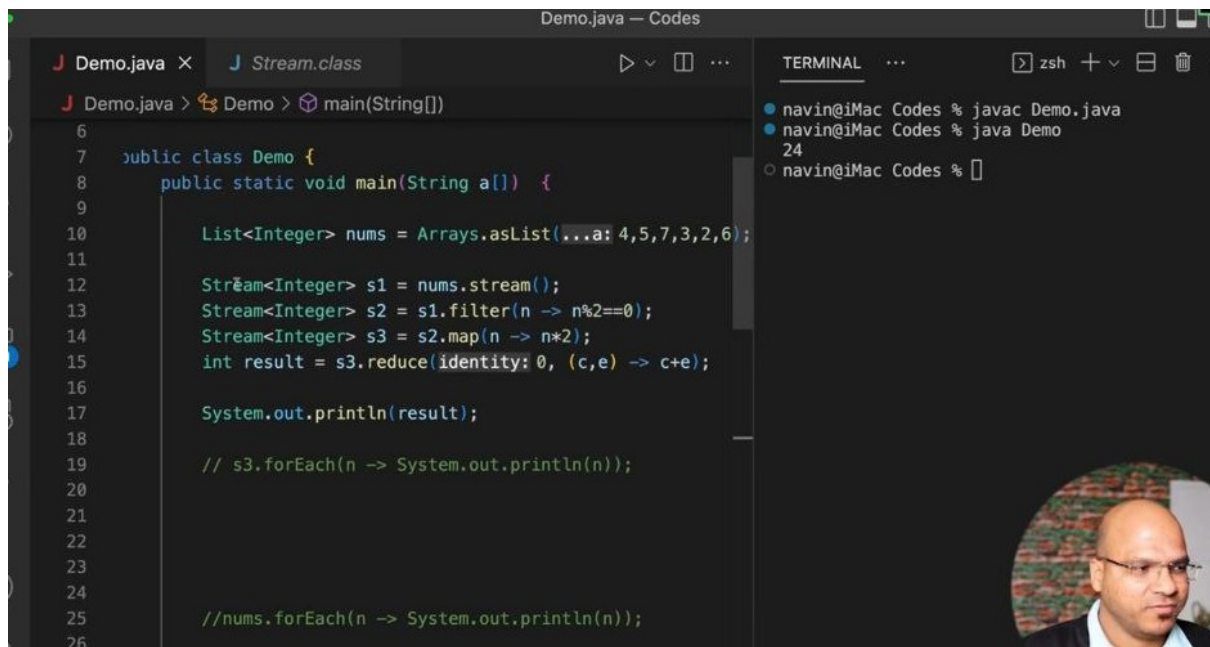
- **FileReader**
- **BufferedReader**
- **InputStreamReader**

- 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/OBufferedReaderInBufferedOutputStream** for
BufferedReaderBufferedWriterBufferedWriter** for character streams



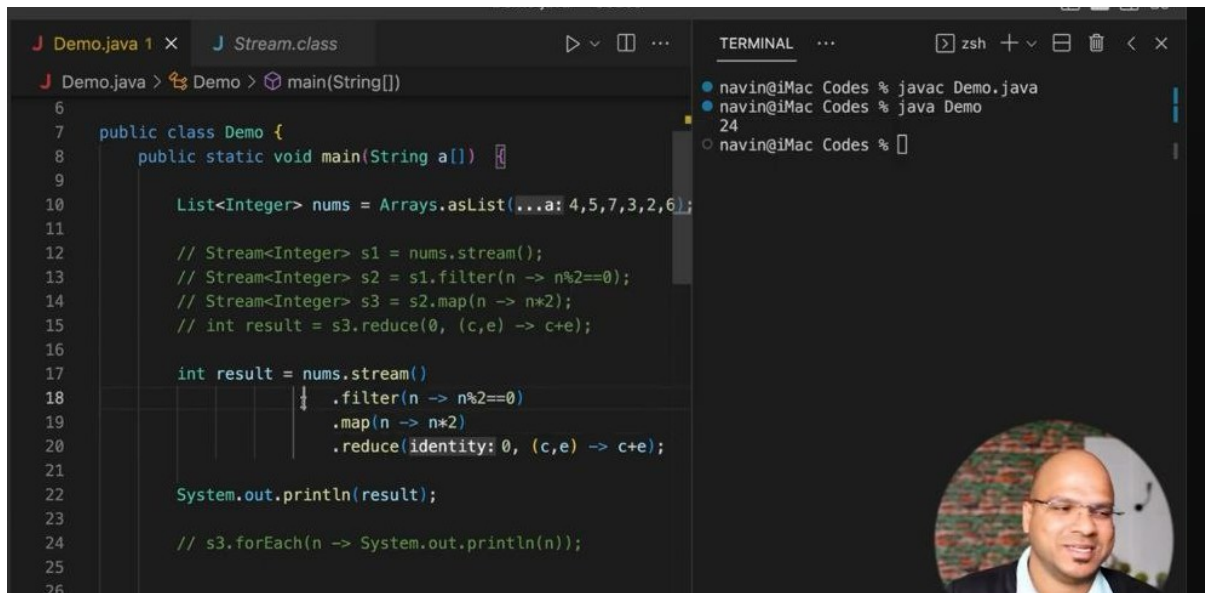
The screenshot shows an IDE window titled "Demo.java — Codes". The editor displays the following Java code:

```
6
7 public class Demo {
8     public static void main(String a[]) {
9
10        List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);
11
12        Stream<Integer> s1 = nums.stream();
13        Stream<Integer> s2 = s1.filter(n -> n%2==0);
14        Stream<Integer> s3 = s2.map(n -> n*2);
15        int result = s3.reduce(identity: 0, (c,e) -> c+e);
16
17        System.out.println(result);
18
19        // s3.forEach(n -> System.out.println(n));
20
21
22
23
24
25        //nums.forEach(n -> System.out.println(n));
26
```

The terminal window on the right shows the following output:

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

A small circular profile picture of a man with glasses is visible in the bottom right corner of the IDE window.



The screenshot shows an IDE with two tabs: 'Demo.java' and 'Stream.class'. The 'Demo.java' tab is active, displaying the following code:

```
6
7 public class Demo {
8     public static void main(String a[]) {
9
10        List<Integer> nums = Arrays.asList(4,5,7,3,2,6);
11
12        // Stream<Integer> s1 = nums.stream();
13        // Stream<Integer> s2 = s1.filter(n -> n%2==0);
14        // Stream<Integer> s3 = s2.map(n -> n*2);
15        // int result = s3.reduce(0, (c,e) -> c+e);
16
17        int result = nums.stream()
18                        .filter(n -> n%2==0)
19                        .map(n -> n*2)
20                        .reduce(identity: 0, (c,e) -> c+e);
21
22        System.out.println(result);
23
24        // s3.forEach(n -> System.out.println(n));
25
26    }
```

The 'Stream.class' tab is also visible. To the right, a terminal window shows the following commands and output:

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

A small circular profile picture of a man with glasses is visible in the bottom right corner of the IDE window.

>>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 class.

Deep vs Shallow Cloning

- *Shallow cloning**: 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 themselves).
- *Deep cloning**: 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

```
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 Cloneable:

Step 1: Define the Address Class

```

java
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

```

C:\Windows\system32\cmd.exe
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.CloneNotSupportedException;
    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

```

Source History
1 package collectiontuts;
2
3
4
5 public class StringDemo
6 {
7     public static void main(String[] args)
8     {
9         String str = "Navin, Mahesh, Rahul, Vijay";
10        String names[] = str.split(",");
11
12        for(String val : names)
13            System.out.println(val);
14    }
15 }
16

```

>>setProperties

```
App.java 23 Properties.class
9 public class App
10 {
11     public static void main( String[] args ) throws Exception
12     {
13         Properties p = new Properties();
14         OutputStream os = new FileOutputStream("dataConfig.properties");
15
16         p.setProperty("url", "localhost:3306/myDb");
17         p.setProperty("uname", "navin");
18         p.setProperty("pass", "0000");
19
20         p.store(os, null);
21     }
22 }
23
24
25
```

```
App.java 23 Properties.class
// Properties file
8
9 public class App
10 {
11     public static void main( String[] args ) throws Exception
12     {
13         Properties p = new Properties();
14
15         InputStream is = new FileInputStream("dataConfig.properties");
16         p.load(is);
17
18         System.out.println(p.getProperty("uname"));
19         System.out.println(p.getProperty("url"));
20
21
22         p.list(System.out);
23
24     }
25 }
```

```
<terminated> App (2) [Java Application] /Library/Java/JavaVirtualMachines/dk1.8.0_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
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
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

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

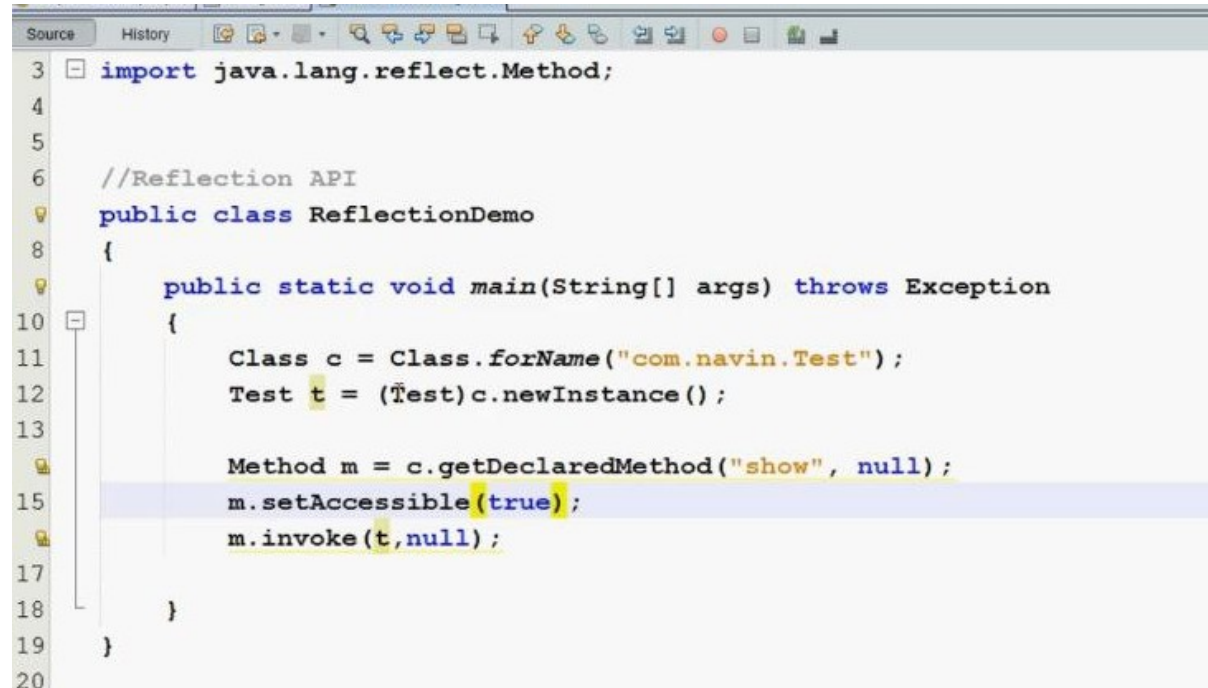
        try (FileInputStream fileIn = new FileInputStream("person.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn)) {
            person = (Person) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
        } catch (ClassNotFoundException c) {
        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

A screenshot of an IDE window showing a Java source file. The code is for a class named 'ReflectionDemo' which has a 'main' method. The 'main' method uses the Reflection API to load a class 'com.navin.Test', create an instance of it, get a method named 'show', make it accessible, and then invoke it. The line 'm.setAccessible(true);' is highlighted in blue. The IDE interface includes a 'Source' tab, a 'History' tab, and a toolbar with various icons. A line number gutter on the left shows lines 3 through 20.

```
3  import java.lang.reflect.Method;
4
5
6  //Reflection API
7  public class ReflectionDemo
8  {
9      public static void main(String[] args) throws Exception
10     {
11         Class c = Class.forName("com.navin.Test");
12         Test t = (Test)c.newInstance();
13
14         Method m = c.getDeclaredMethod("show", null);
15         m.setAccessible(true);
16         m.invoke(t,null);
17     }
18 }
19
20
```

>>Method reference

```
FirstCode.java Consumer.class PrintStream.class
11 {
12     public String convert(String s)
13     {
14         if(s.length() <= 3)
15             s = s.toUpperCase();
16         else
17             s = s.toLowerCase();
18
19         return s;
20     }
21 }
22
23 class MyPrinter
24 {
25     public void print(String str, Parser p)
26     {
27         str = p.parse(str);
28         System.out.println(str);
29     }
30 }
31
32 public class FirstCode
33 {
34
35     public static void main(String[] args)
36     {
37         StringParser sp = new StringParser();
38         String str = "Nav";
39         MyPrinter mp = new MyPrinter();
40         mp.print(str, sp::convert);
41
42     }
43 }
44
```

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 Copy code  
  
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 Copy code  
  
ApplicationContext context = new FileSystemXmlApplicationContext("path/to/applicationContext.xml");  
MyBean myBean = context.getBean(MyBean.class);
```

c. `AnnotationConfigApplicationContext`

Loads context definition from annotated classes.

```
java Copy code

@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 Copy code

@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

 Copy code

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

Bean Definition in XML Configuration:

xml

 Copy code

```
<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 Copy code

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

Usage:

```
java Copy code

Service
public 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. Spring 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
```

 Copy code

AppInfo Class

java

 Copy code

```
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

 Copy code

```
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);
    }
}
```

```
@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 Ijava.lang.String('Durgesh Tiwari') }")  
private String name;
```
