Java Backend Development

11 November 2022 18:25

© **RAJAT KUMAR**
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Backend Development (References) :

- Backend Developer Roadmap
- Core Java LinkedIn Path
- Java Collections Frameworks

- Twitter : Java Backend Development
- 1) Core Java (Done)
  - Java Programming Tutorial (YouTube Playlist)
  - Github : Java Tutorial for Beginners Crash Course
- 2) JDBC
- 3) MySQL
- 4) JSP + Servlet (-> Maven -> Design Pattern)
- 5.1) Core Spring Framework
- 5.2) spring REST & spring DATA
- 5.3) Spring Security
- 6) Hibernate Framework
- 7) Spring Boot
- 8.1) Learn to Use AWS & Deploy Java Apps
- 8.2) Learn Basic Docker
- 8.3) Learn Basic Kubernetes
- 8.4) Deploy Spring Boot App on Kubernetes

DSA -> System Design -> Resume Java, DBMS

DSA :
Coding Decoded, Algorithms Made Easy

- Design Patterns,
- C++, Modern C++
- UML
- OOPS
  - OOPS CheatSheet
- Solid Design Principle
- System Design :
  - System Design Playlist

DSA/Coding :

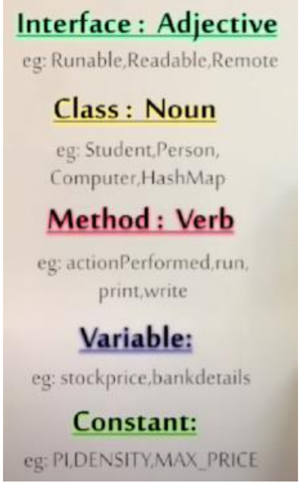
1

Java Programming Tutorial (YouTube Playlist)

```
public class FirstCode {

        public static void main(String args[]) {

                System.out.println("Hello World");
        }
}
```

Source File -> **Compiler** -> Output : Byte Code
Byte Code -> Run on **JVM (Java Virtual Machine)** -> Get Result

Naming Convention :



Interface : Adjective
eg: Runable,Readable,Remote

Class : Noun
eg: Student,Person, Computer,HashMap

Method : Verb
eg: actionPerformed,run, print,write

Variable:
eg: stockprice,bankdetails

Constant:
eg: PI,DENSITY,MAX_PRICE

Constructor :

- Constructor is a member method - ClassName()
- Constructor has the same name as class name
- Constructor never return anything - public ClassName()
- It will used to allocate memory

Static variable :

- If we don't won't a variable object specific but class specific. We can make that variable static. That variable is stored in loader memory
- When is have static variable, we can use class name to access it or object name. We prefer to use class name.

- If we initialize no static variable, we can use constructor.
- If we want to define static variable, we can use static block. (Static block runs only once, and first, before constructor.)

```
class Emp
{
```

2

Jagged Array : An array of arrays such that member arrays can be of different sizes.
Example : int[][] jaggedArray3 = { new int[] { 1, 3, 5, 7, 9 }, new int[] { 0, 2, 4, 6 }, new int[] { 11, 22 } };

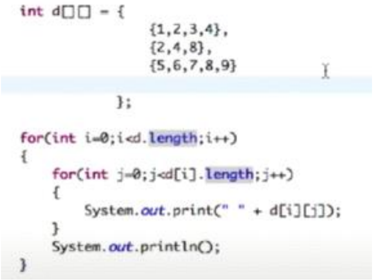
Create an Array :

- int nums[] = new int[4];
- Int nums[] = {1, 2, 3, 4, 5};

2D Array :

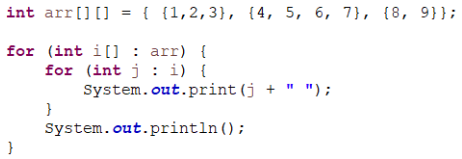
- int two_d_array[][] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

Jagged Array :



```
int d□□ = {
                {1,2,3,4},
                {2,4,8},
                {5,6,7,8,9}

           };

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

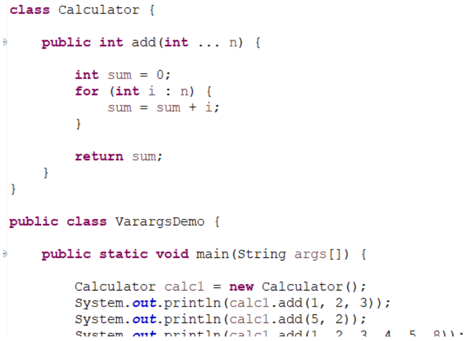
Enhanced for Loop :



```
int arr[][] = { {1,2,3}, {4, 5, 6, 7}, {8, 9}};

for (int i[] : arr) {
    for (int j : i) {
        System.out.print(j + " ");
    }
    System.out.println();
}
```

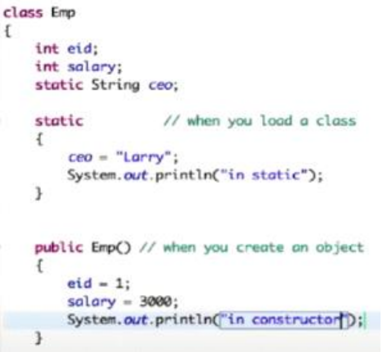
Variable Arguments (varargs) :

- Used when we don't know how many argument are going to pass in a function.



```
class Calculator {

    public int add(int ... n) {

        int sum = 0;
        for (int i : n) {
            sum = sum + i;
        }

        return sum;
    }
}

public class VarargsDemo {

    public static void main(String args[]) {

        Calculator calc1 = new Calculator();
        System.out.println(calc1.add(1, 2, 3));
        System.out.println(calc1.add(5, 2));
        System.out.println(calc1.add(1, 2, 3, 4, 5, 8));
```

- If we want to define static variable, we can use static block. (Static block runs only once, and first, before constructor.)



```
class Emp
{
    int eid;
    int salary;
    static String ceo;

    static          // when you load a class
    {
        ceo = "Larry";
        System.out.println("in static");
    }


    public Emp() // when you create an object
    {
        eid = 1;
        salary = 3000;
        System.out.println("in constructor");
    }
```

- We can't access not static variable inside static block, we also need to make that variable static to access it inside static block. (even if it's static main block)

Inner Class : A class inside a class.

Outer class -> Inner class;

How to access :

Outer obj = new Outer();

Outer.Inner obj2 = obj.new Inner();

When there is a requirement we require a class only for the main class purpose

static class InnerStatic {

```
    public void display() {
        System.out.println("In static display()");
    }
}
```

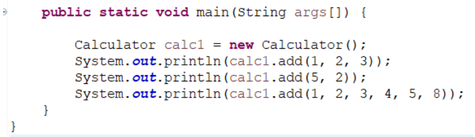
How to access :

Outer.InnerStatic obj3 = new Outer.InnerStatic();

obj3.display();

3 types of inner classes :

- Member class (normal inner class as shown above)
- Static class (static inner class as shown above)
- Anonymous class



```
    public static void main(String args[]) {

        Calculator calc1 = new Calculator();
        System.out.println(calc1.add(1, 2, 3));
        System.out.println(calc1.add(5, 2));
        System.out.println(calc1.add(1, 2, 3, 4, 5, 8));
    }
}
```

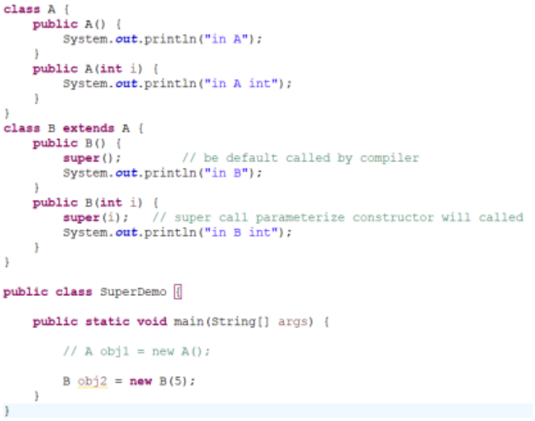
- The above add function will work for any number of arguments.
- All arguments will be pass as an array.
- Add(int … n) It's called variable length argument (3 dots are must require)

Inheritance :

- Used using **extends** word.
- Java supports single, muti level inheritance, but not support multiple inheritance (because of ambiguity)
- IS-A relationship - When a class extends another class.
- HAS-A relationship - When we have a class in which we are creating the object of another class

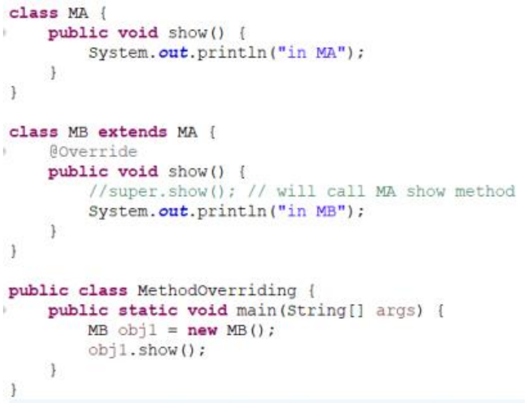
Super Keyword :

- When we create sub class object, super class default constructor will also called first automatically and sub class constructor will be called.
- Every sub class has super() called by compiler in constructor.
- If we change super() to super(i), then instead of default constructor called, it can call parameterize constructor.



```
class A {
    public A() {
        System.out.println("in A");
    }
    public A(int i) {
        System.out.println("in A int");
    }
}
class B extends A {
    public B() {
        super();        // be default called by compiler
        System.out.println("in B");
    }
    public B(int i) {
        super(i);   // super call parameterize constructor will called
        System.out.println("in B int");
    }
}

public class SuperDemo {

    public static void main(String[] args) {

        // A obj1 = new A();

        B obj2 = new B(5);
    }
}
```

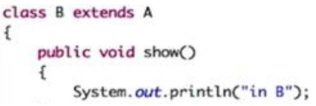
Method Overriding



```
class MA {
    public void show() {
        System.out.println("in MA");
    }
}

class MB extends MA {
    @Override
    public void show() {
        //super.show(); // will call MA show method
        System.out.println("in MB");
    }
}

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

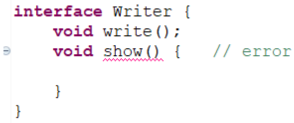
Runtime Polymorphism & Dynamic Method Dispatch

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



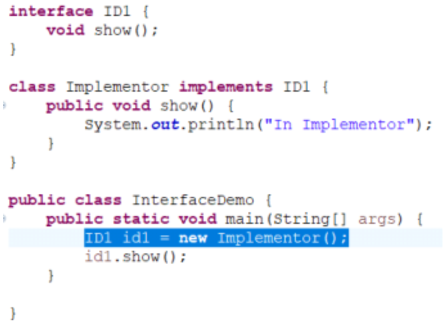
```
class B extends A
{
    public void show()
    {
        System.out.println("in B");
```

Interface

- Interface is same as abstract class, the difference is :
    - In abstract class we can have abstract method as well as normal method (that we can define).
    - But in interface we can have only abstract methods, we can't define any methods.
    - 

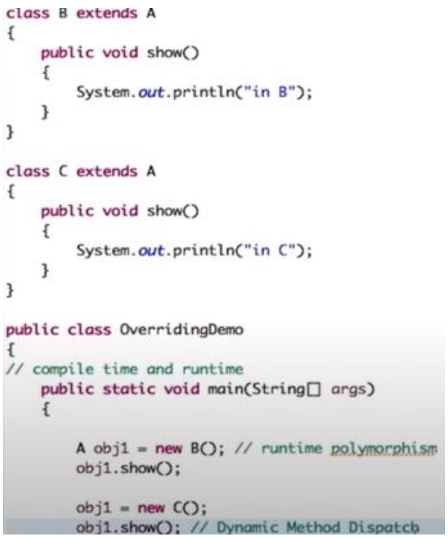
```
interface Writer {
    void write();
    void show() {    // error

    }
}
```

    - By default all methods in interface are public abstract.
- We can't extends 2 classes as it's multiple inheritance and java doesn't support
- So we can create an interface and *implements* it and can extend another class. In this way we can achieve multiple inheritance.
- We can create reference of interface but can't create the object of interface.



```
interface ID1 {
    void show();
}

class Implementor implements ID1 {
    public void show() {
        System.out.println("In Implementor");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        ID1 id1 = new Implementor();
        id1.show();
    }

}
```

- Given a choice always go with interface instead of abstract classes, because



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

class C extends A
{
    public void show()
    {
        System.out.println("in C");
    }
}

public class OverridingDemo
{
// compile time and runtime
    public static void main(String[] args)
    {

        A obj1 = new B(); // runtime polymorphism
        obj1.show();

        obj1 = new C();
        obj1.show(); // Dynamic Method Dispatch
```

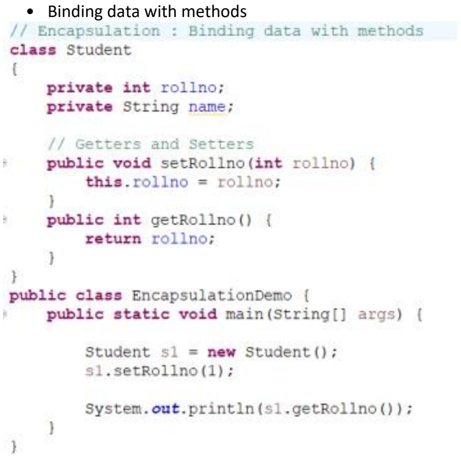
// Class A must have show method, only then A obj = new B() -> obj1.show() will work.

- Method overriding is called as run time polymorphism.
- Once we changed the object, it's changing the calling, it's called as Dynamic Method Dispatch.

Rule: Runtime polymorphism can't be achieved by data members.
A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

Encapsulation

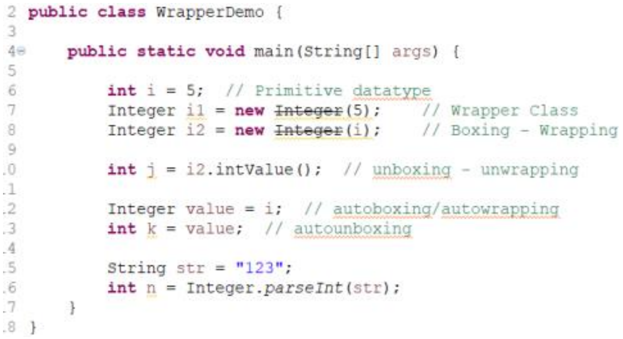
- Binding data with methods



```
// Encapsulation : Binding data with methods
class Student
{
    private int rollno;
    private String name;

    // Getters and Setters
    public void setRollno(int rollno) {
        this.rollno = rollno;
    }
    public int getRollno() {
        return rollno;
    }
}
public class EncapsulationDemo {
    public static void main(String[] args) {

        Student s1 = new Student();
        s1.setRollno(1);

        System.out.println(s1.getRollno());
    }
}
```

- We use encapsulation to make data safe
- Can use methods for logging. Can't be done using variable (obj1.rollno = 2, not work)
- Always better to access the variables with the help of methods.

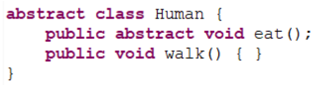
Wrapper Class | AutoBoxing



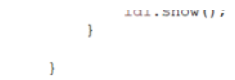
```
public class WrapperDemo {

    public static void main(String[] args) {

        int i = 5;  // Primitive datatype
        Integer i1 = new Integer(5);    // Wrapper Class
        Integer i2 = new Integer(i);    // Boxing - Wrapping

        int j = i2.intValue();  // unboxing - unwrapping

        Integer value = i;  // autoboxing/autowrapping
        int k = value;  // autounboxing

        String str = "123";
        int n = Integer.parseInt(str);
    }
}
```

- Putting Primitive value inside the object, it's called **boxing/Wrapping**.
- Fetching value from wrapper class to primitive datatype value is called **unboxing/unwrapping**.
- Primitive works faster than wrapper classes.

Abstract Keyword



```
abstract class Human {
    public abstract void eat();
    public void walk() { }
}
```

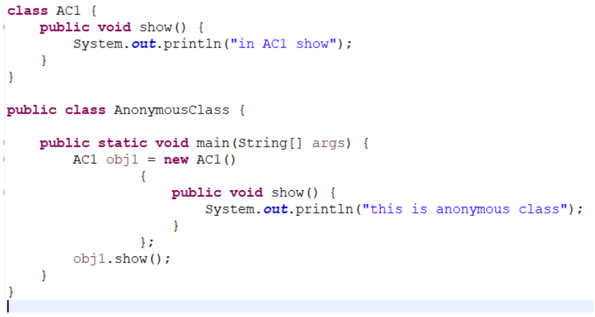


```
                id1.show();
        }

    }
```

- Given a choice always go with interface instead of abstract classes, because may be in future if require we can achieve multiple inheritance.

Anonymous Inner Class

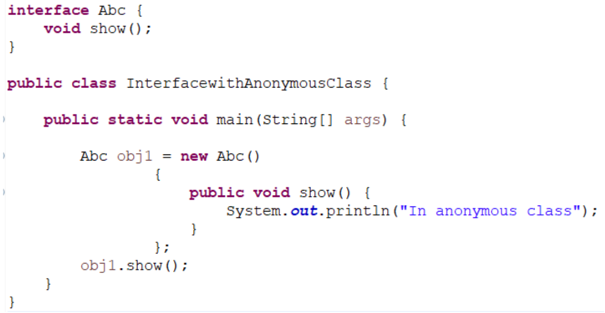
- A class which doesn’t have any name.
- Anonymous classes can be created by providing the implementation just before the semicolon when creating an object.
- The purpose of anonymous classes is to avoid creating a new class when the only purpose is to override a method.
- Anonymous classes do not have a name and their scope is limited to the current context.
- If out intension is to class as one time only, we can prefer to use anonymous class



```
class AC1 {
    public void show() {
        System.out.println("in AC1 show");
    }
}

public class AnonymousClass {

    public static void main(String[] args) {
        AC1 obj1 = new AC1()
                {
                    public void show() {
                        System.out.println("this is anonymous class");
                    }
                };
        obj1.show();
    }
}
```

Anonymous Class with interface

- In an interface, we cannot create an object directly because it lacks implementation but,
- We can create object of interface using anonymous class

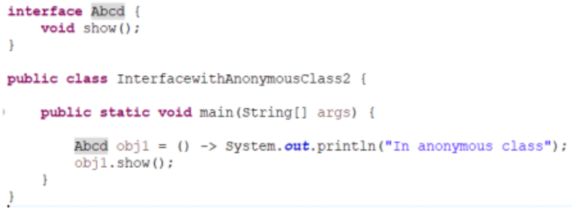


```
interface Abc {
    void show();
}

public class InterfacewithAnonymousClass {

    public static void main(String[] args) {

        Abc obj1 = new Abc()
                {
                    public void show() {
                        System.out.println("In anonymous class");
                    }
                };
        obj1.show();
    }
}
```

Types of Interface

1) **Normal interface** : has more than one method
2) **SAM interface** : has only one method (In Java 8, it's **Functional interface**, allows the use of Lambda expressions)
3) **Marker interface** : has no method.

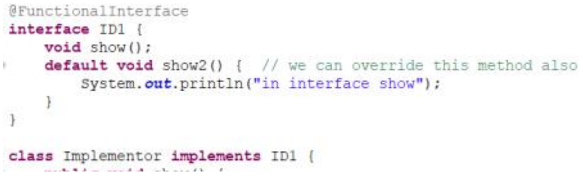
- Lambda expressions are a feature from Scala language that Java 8 adopted and Java has all features of Scala.
- A functional interface can be identified using an annotation called @FunctionalInterface.
- Functional interfaces can be used to create objects of the interface in one line of code using lambda expressions.
- Lambda expressions can only be used with functional interfaces.



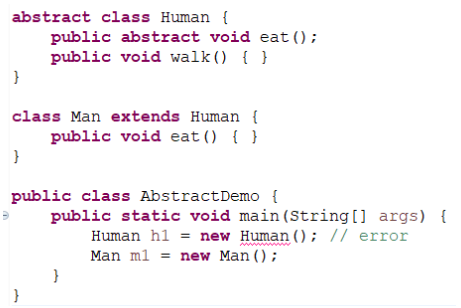
```
interface Abcd {
    void show();
}

public class InterfacewithAnonymousClass2 {

    public static void main(String[] args) {

        Abcd obj1 = () -> System.out.println("In anonymous class");
        obj1.show();
    }
}
```

Default Method in Interface

- In abstract classes, both the declaration and definition of methods are possible. However, in interfaces, only the declaration of methods is possible.
- Starting from Java 1.8, methods can be defined in interfaces using keyword default.
- A functional interface can have only one abstract method but can have multiple default methods.

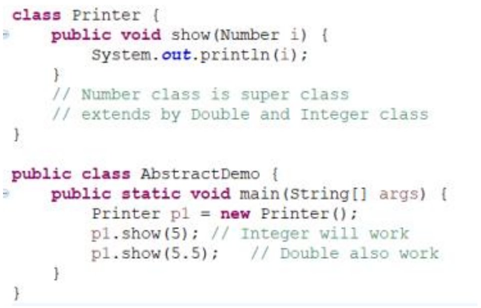


```
@FunctionalInterface
interface ID1 {
    void show();
    default void show2() {  // we can override this method also
        System.out.println("in interface show");
    }
}

class Implementor implements ID1 {
```



```
abstract class Human {
    public abstract void eat();
    public void walk() { }
}

class Man extends Human {
    public void eat() { }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Human h1 = new Human(); // error
        Man m1 = new Man();
    }
}
```

- Abstract keyword can be used with both methods and class.
- **Abstract Class** : When we don't want anyone to create any object of that class we can make it abstract (But we can create a reference : Human h1 = new Man();)
    - But the abstract class can be extend by a new class then we can create the object of that new class
    - The new class should declared all the abstract method that are in abstract class.
- If we have an abstract method in class, the class has to be abstract class.
- If we only declare a method in abstract class it has to abstract method.

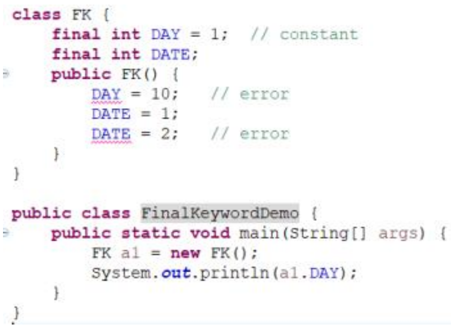


```
class Printer {
    public void show(Number i) {
        System.out.println(i);
    }
    // Number class is super class
    // extends by Double and Integer class
}

public class AbstractDemo {
    public static void main(String[] args) {
        Printer p1 = new Printer();
        p1.show(5); // Integer will work
        p1.show(5.5);   // Double also work
    }
}
```

- We use abstract classes because we don't want to create similar multiple methods, we can use that abstract class just like we did above as Number abstract class.

Final Keyword

- Final Keyword can be used with:
    - Variables
    - Methods
    - Class

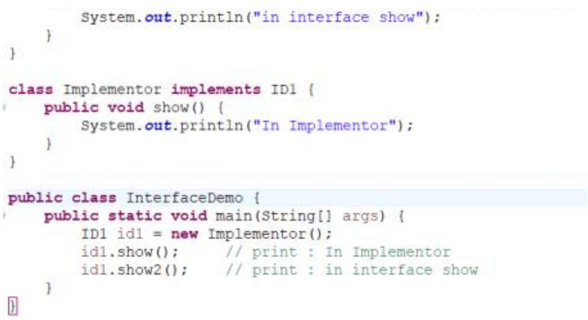


```
class FK {
    final int DAY = 1;  // constant
    final int DATE;
    public FK() {
        DAY = 10;   // error
        DATE = 1;
        DATE = 2;   // error
    }
}

public class FinalKeywordDemo {
    public static void main(String[] args) {
        FK a1 = new FK();
        System.out.println(a1.DAY);
    }
}
```

- When we make a variable final, it becomes a constant. And it's value can't be changed.
- Once value of final variable defined, it can't be changed.
- If we make a class final, no other class can extends it.
- If we make a method final, no one can override that by extending the class.

Exception Handling

- There are two types of exceptions :
    - Checked
    - Unchecked.
- Unchecked exceptions are not specified by the compiler, but the program throws an exception at runtime.
- To handle exceptions, we use try-catch. Try block contains the critical statements, and catch block catches the exception object.
- If an exception is thrown, the program jumps to the catch block, and if there is no error,



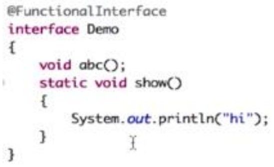
```
        System.out.println("in interface show");
    }
}

class Implementor implements ID1 {
    public void show() {
        System.out.println("In Implementor");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        ID1 id1 = new Implementor();
        id1.show();      // print : In Implementor
        id1.show2();     // print : in interface show
    }
}
```

Multiple Inheritance issue with Interface

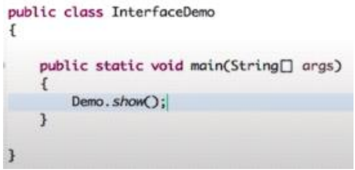
- Multiple Inheritance issue with Interface when we implements 2 interface and both have default method with same name.
- One solution to this issue is to define a method with the same name inside the implementing class to remove ambiguity.
- Another solution is to override the default method and call the super interface method using the interface name and the super keyword

Static method in Interface

- By default when we create a variable in interface it become constant/final
- With static methods in interfaces, there is no need to create an object of the class that implements the interface to call the method.
- It is not possible to create an object of an interface, so static methods provide an alternative way to call methods in interfaces.



```
@FunctionalInterface
interface Demo
{
    void abc();
    static void show()
    {
        System.out.println("hi");
    }
}
```



```
public class InterfaceDemo
{

    public static void main(String[] args)
    {
        Demo.show();
    }

}
```

Packages

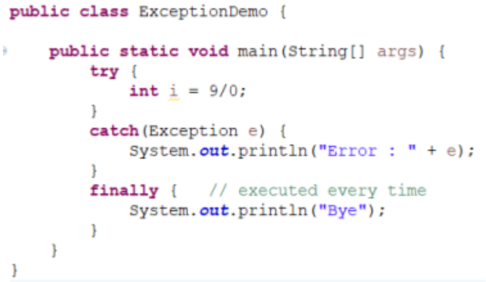
- Java has many built-in classes, as well as external libraries, which are organized into packages.
  A package is a folder that contains classes that are classified based on their functionality and use.
- The use of packages makes it easy to manage classes and locate them quickly.
- Naming a package after a domain name can make it unique and avoid naming conflicts. Example : com.rajat.package
- The star (*) can be used to retrieve all the classes within a package, but it only retrieves classes and not sub-packages.

Access Modifiers

- Only modifier we can use with class is final, abstract and public.
- If we have inner class we can use private keyword.
- If we write just class A, it will be default not public class
- If we don't mention the public keyword in class, we can't access that class outside the package.
- Default class can only be used inside the same package.
- If we want to use a variable outside the package, make sure it's public.
- Int a means default variable and this variable can't be access outside the package.

Private: Specific Class
Default: Specific Package
Public: Any Class or Package
Protected: Subsiding Class

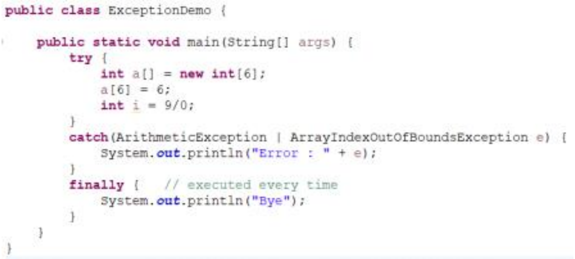
- Unchecked exceptions are not specified by the compiler, but the program throws an exception at runtime.
- To handle exceptions, we use try-catch. Try block contains the critical statements, and catch block catches the exception object.
- If an exception is thrown, the program jumps to the catch block, and if there is no error, the program continues to execute the statements outside of the try-catch block.
- We can use "finally" block, where we can write the statement that we want to execute whether the exception occurs or not.
- If instead of System.out.println, we write System.err.println, the colour of error will be red instead of black.



```
public class ExceptionDemo {

    public static void main(String[] args) {
        try {
            int i = 9/0;
        }
        catch(Exception e) {
            System.out.println("Error : " + e);
        }
        finally {   // executed every time
            System.out.println("Bye");
        }
    }
}
```

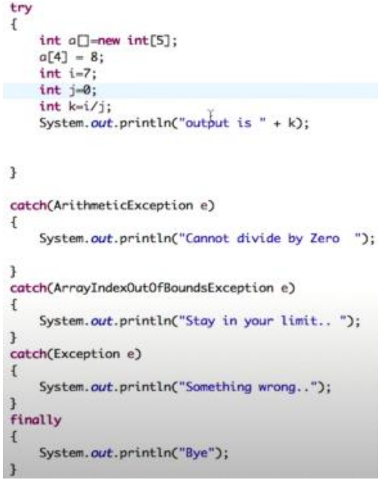
Multiple Catch Blocks

- In Java 1.7 and later, we can have multiple exceptions in one catch block. In Java 1.6 and earlier, this is not possible.



```
public class ExceptionDemo {

    public static void main(String[] args) {
        try {
            int a[] = new int[6];
            a[6] = 6;
            int i = 9/0;
        }
        catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Error : " + e);
        }
        finally {   // executed every time
            System.out.println("Bye");
        }
    }
}
```

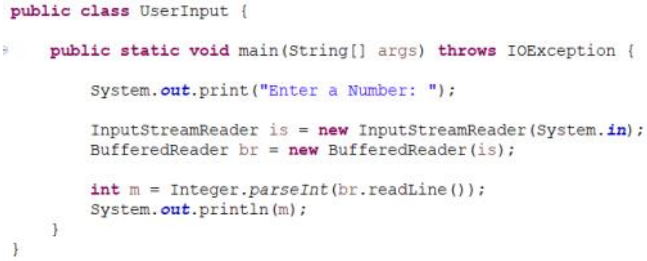
- Multiple catch blocks can be useful when you want to handle different exceptions differently.
- The order of catch blocks matters, and the catch block that handles a particular exception should be placed first in the order and only in last Exception master of all should be placed



```
try
{
    int a[]=new int[5];
    a[4] = 8;
    int i=7;
    int j=0;
    int k=i/j;
    System.out.println("output is " + k);


}

catch(ArithmeticException e)
{
    System.out.println("Cannot divide by Zero  ");

}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("Stay in your limit.. ");
}
catch(Exception e)
{
    System.out.println("Something wrong..");
}
finally
{
    System.out.println("Bye");
}
```

User Input BufferedReader

- The method "readLine()" is used to get input from the user, but it returns a String and needs to be converted to an int using "parseInt()" method of the Integer class.



```
public class UserInput {

    public static void main(String[] args) throws IOException {

        System.out.print("Enter a Number: ");

        InputStreamReader is = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(is);

        int m = Integer.parseInt(br.readLine());
        System.out.println(m);
    }
}
```

Checked Exception

- Checked exceptions may occur when asking for data from a BufferedReader due to user input errors.
- There are two ways to handle checked exceptions: using try-catch or using the throws exception statement.
- BufferedReaders are resources that must be closed to free up memory.
- Proper exception handling involves creating a resource, handling it in a try-catch block, and closing it in the finally block.

Try with Resources

7

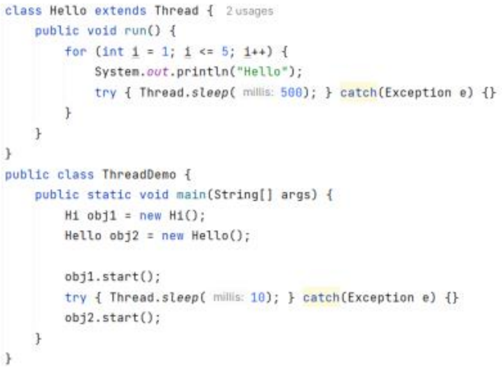
Multithreading

What is a thread :
MS Word : Main Process
Typing : Sub Process (Thread)
Spell Check : Sub Process (Thread)

Thread : Unit of a Process

- Multi-threading allows for multitasking and running multiple tasks simultaneously.
- In Java, threads are units of a process and can be used to perform tasks concurrently.
- The need for threads arises due to multi-core processors, where multiple threads can utilize the cores effectively.
- Threads can be implemented in Java by extending the Thread class or implementing the Runnable interface.
- Using threads, tasks can be divided among multiple threads to take advantage of parallel processing.
- Sleep, wait, and notify methods can be used in threads, but stopping a thread should be done with caution.

Thread Class

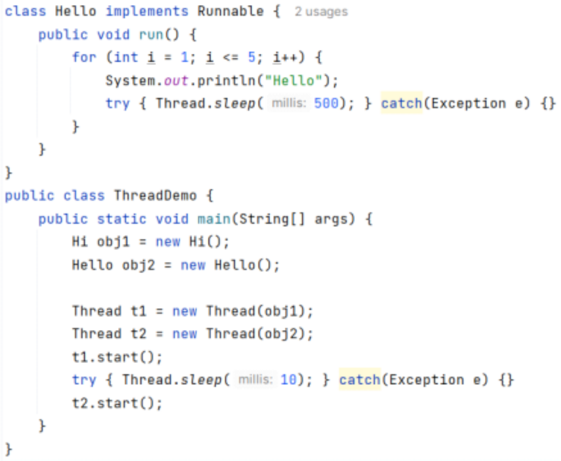


```
class Hello extends Thread {  2 usages
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hello");
            try { Thread.sleep( millis: 500); } catch(Exception e) {}
        }
    }
}
public class ThreadDemo {
    public static void main(String[] args) {
        Hi obj1 = new Hi();
        Hello obj2 = new Hello();

        obj1.start();
        try { Thread.sleep( millis: 10); } catch(Exception e) {}
        obj2.start();
    }
}
```

- By default, Java has a main thread that executes the code.
- The thread.sleep() method is used to introduce a pause in the execution of the threads.
- The thread.start() method is used to start the threads.

Runnable Interface

- Implementation of threads using the Thread class can be limited due to Java's lack of multiple inheritance.
- To overcome this limitation, the Runnable interface can be implemented instead of extending the Thread class.
- Instead of using extends Thread, implements Runnable is used to indicate implementation of the interface.
- The Runnable interface is a functional interface with a single method, run().



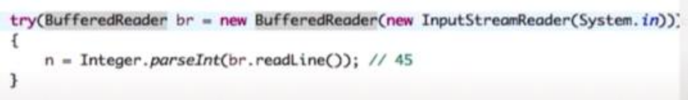
```
class Hello implements Runnable {  2 usages
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hello");
            try { Thread.sleep( millis: 500); } catch(Exception e) {}
        }
    }
}
public class ThreadDemo {
    public static void main(String[] args) {
        Hi obj1 = new Hi();
        Hello obj2 = new Hello();

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
        t1.start();
        try { Thread.sleep( millis: 10); } catch(Exception e) {}
        t2.start();
    }
}
```

- Runnable interface does not have a start() method, so a Thread object must be created
- Create Thread objects (e.g., t1 and t2) and call their start() methods

- Proper exception handling involves creating a resource, handling it in a try-catch block, and closing it in the finally block.

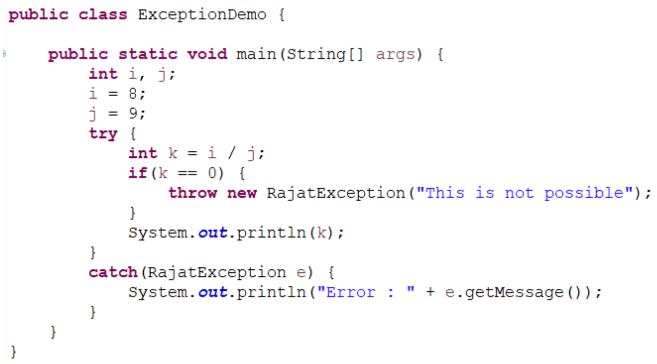
Try with Resources

- Private resources can be closed using try block without handling the exception.
- Try block with resources automatically closes the resource as soon as the object goes out of scope and it can be used without using catch or finally block.
- The syntax used for try block with resources is called "try with the source" as shown below.



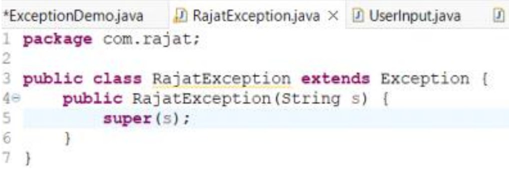
```
try(BufferedReader br = new BufferedReader(new InputStreamReader(System.in)))
{
    n = Integer.parseInt(br.readLine()); // 45
}
```

User Defined



```
public class ExceptionDemo {

    public static void main(String[] args) {
        int i, j;
        i = 8;
        j = 9;
        try {
            int k = i / j;
            if(k == 0) {
                throw new RajatException("This is not possible");
            }
            System.out.println(k);
        }
        catch(RajatException e) {
            System.out.println("Error : " + e.getMessage());
        }
    }
}
```

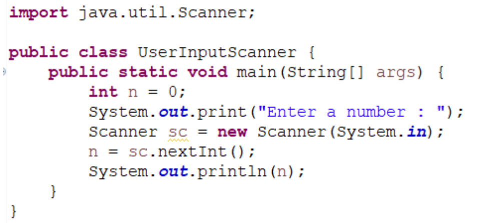
- The message is passed from the user-defined exception class to the "Throwable" constructor through the use of "super" and "getMessage()" methods.
- Creating a user-defined exception is not difficult and involves creating a class that extends "Exception", adding a constructor that accepts a string, and calling the superclass constructor using "super".



```
package com.rajat;

public class RajatException extends Exception {
    public RajatException(String s) {
        super(s);
    }
}
```

User Input using Scanner

- Scanner is a tool used for user input in Java programming.
- It is preferred over other input methods because it is simple to use and understand.
- Scanner has methods like nextInt, nextLong, nextDouble, etc. for different types of input.
- Using Scanner eliminates the need to handle exceptions and convert strings to integers like in other input methods (e.g. BufferedReader).



```
import java.util.Scanner;

public class UserInputScanner {
    public static void main(String[] args) {
        int n = 0;
        System.out.print("Enter a number : ");
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        System.out.println(n);
    }
}
```

8

Collection and Generics

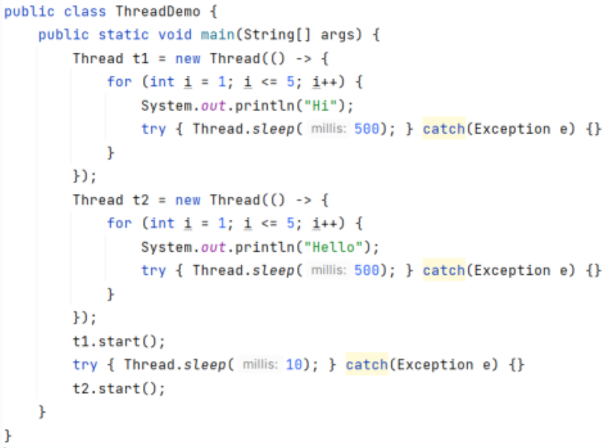
- collection (topic/concept), Collection (interface), and Collections (class).
- If we know size is fixed, always use array instead of ArrayList (collection), because array is faster.
- 1.2 version have collection, 1.5 have generic <integer>, 1.7 have :
    - Collection <Integer> values = new ArrayList<>();   // don't need to specify Integer on right side
    - Collection <Integer> values = new ArrayList<Integer>();   // need to specify integer on right side before 1.7 version
- Hierarchy = Collection -> List -> ArrayList   // List implements/extend collections

```
}
```

- Runnable interface does not have a start() method, so a Thread object must be created
- Create Thread objects (e.g., t1 and t2) and call their start() methods
- Pass the Runnable object (e.g., obj1 and obj2) to the Thread constructor

Using Lambda Expression

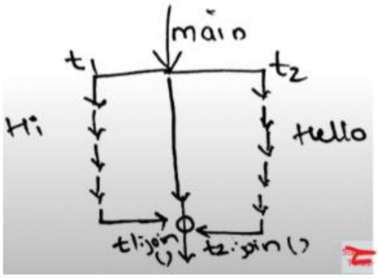
- If we have a class which use only once, then instead of creating a separate class, we can create an anonymous class.



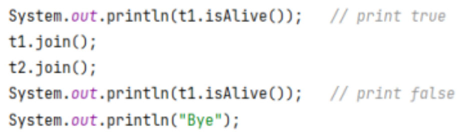
```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Hi");
                try { Thread.sleep( millis: 500); } catch(Exception e) {}
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Hello");
                try { Thread.sleep( millis: 500); } catch(Exception e) {}
            }
        });
        t1.start();
        try { Thread.sleep( millis: 10); } catch(Exception e) {}
        t2.start();
    }
}
```

- We have reduced the number of lines
- Now we are using anonymous class, lambda expression and making the code more efficient.

Join and is Alive Method



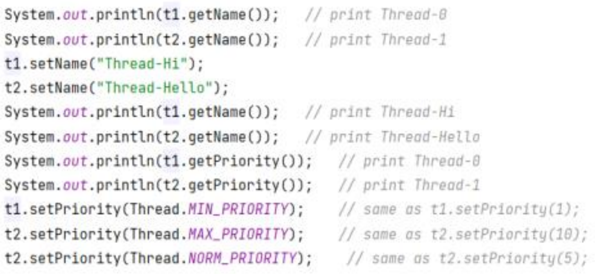
- The join method is used to make the main thread wait for t1 and t2 to complete their tasks.
- The isAlive method is used to check if a thread is still running.
- If t1 is checked with isAlive before joining, it will return true, but after joining, it will return false.



```
System.out.println(t1.isAlive());    // print true
t1.join();
t2.join();
System.out.println(t1.isAlive());    // print false
System.out.println("Bye");
```

Thread Name and Priority

- The name of a thread can be obtained using Thread.getName() and set using Thread.setName().
- Be default thread name is Thread-0, Thread-1, and so on.
- Multiple threads can have different names for better thread management.
- Threads can be created with a name using the constructor Thread(ThreadGroup group, String name).
- The default priority of a thread is 5 on a scale from 1 to 10.
- The current thread's priority can be obtained using Thread.currentThread().getPriority().
- Thread priorities can be changed using Thread.setPriority(int priority).
- Constants like Thread.MIN_PRIORITY, Thread.MAX_PRIORITY, and Thread.NORM_PRIORITY can be used for readability.

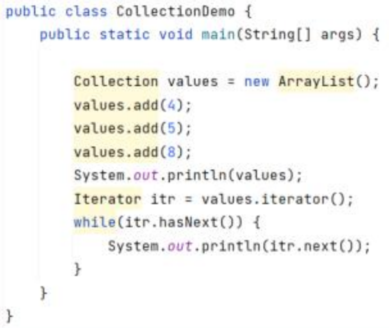


```
System.out.println(t1.getName());   // print Thread-0
System.out.println(t2.getName());   // print Thread-1
t1.setName("Thread-Hi");
t2.setName("Thread-Hello");
System.out.println(t1.getName());   // print Thread-Hi
System.out.println(t2.getName());   // print Thread-Hello
System.out.println(t1.getPriority());   // print Thread-0
System.out.println(t2.getPriority());   // print Thread-1
t1.setPriority(Thread.MIN_PRIORITY);    // same as t1.setPriority(1);
t2.setPriority(Thread.MAX_PRIORITY);    // same as t2.setPriority(10);
t2.setPriority(Thread.NORM_PRIORITY);    // same as t2.setPriority(5);
```

- Collection <Integer> values = new ArrayList<>(); // don't need to specify Integer on right side
    - Collection <Integer> values = new ArrayList<Integer>(); // need to specify integer on right side before 1.7 version
- Hierarchy = Collection -> List -> ArrayList // List implements/extend collections
- Collection and List are Interface, ArrayList is a class
    - Collection doesn't word with index number, but List support as it have extra features.
    - List<Integer> values = new ArrayList<>();
- Set<Integer> unique_numbers = new HashSet<>(); // order is random
- Set<Integer> unique_sorter_numbers = new TreeSet<>(); // sorted order
- Map<Interger, String> m = new HashMap<>(); // Map is also an interface.
    - We have 2 class which implements Map
        - HashMap (Synchronized)
        - HashTable (Not Synchronized)

Collection and Iterator Interface

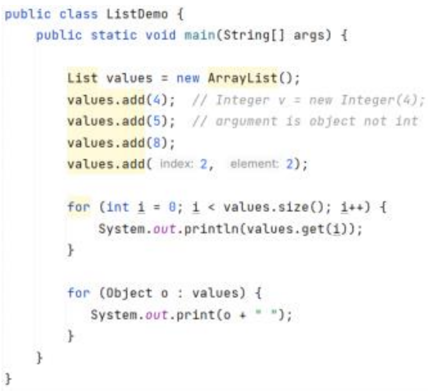
- To create a collection, an interface called Collection is used.
- The ArrayList class implements the Collection interface.
- Values can be added to a collection using the add method.
- The Collection interface has methods such as size, contains, add, remove, and convert to an array.
- Collections do not support index numbers, so a special interface called Iterator is used to fetch values from the collection.



```
public class CollectionDemo {
    public static void main(String[] args) {

        Collection values = new ArrayList();
        values.add(4);
        values.add(5);
        values.add(8);
        System.out.println(values);
        Iterator itr = values.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

List Interface

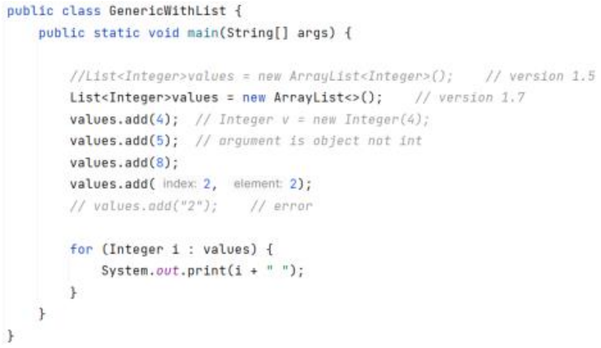
- Collection interface doesn't support index numbers
- List interface extends Collection and supports index numbers
- List can have any type of value, not just integers or strings
- List is a list of objects, not a specific type of element



```
public class ListDemo {
    public static void main(String[] args) {

        List values = new ArrayList();
        values.add(4);  // Integer v = new Integer(4);
        values.add(5);  // argument is object not int
        values.add(8);
        values.add( index: 2,  element: 2);

        for (int i = 0; i < values.size(); i++) {
            System.out.println(values.get(i));
        }

        for (Object o : values) {
           System.out.print(o + " ");
        }
    }
}
```

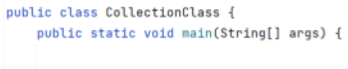
Using Generic with List

- Generic can be used to restrict a type to int, string, or anything so it doesn't support anything else.

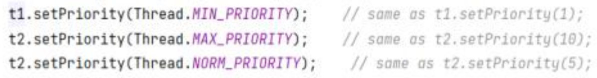


```
public class GenericWithList {
    public static void main(String[] args) {

        //List<Integer>values = new ArrayList<Integer>();    // version 1.5
        List<Integer>values = new ArrayList<>();    // version 1.7
        values.add(4);  // Integer v = new Integer(4);
        values.add(5);  // argument is object not int
        values.add(8);
        values.add( index: 2,  element: 2);
        // values.add("2");    // error

        for (Integer i : values) {
            System.out.print(i + " ");
        }
    }
}
```

Collection Class



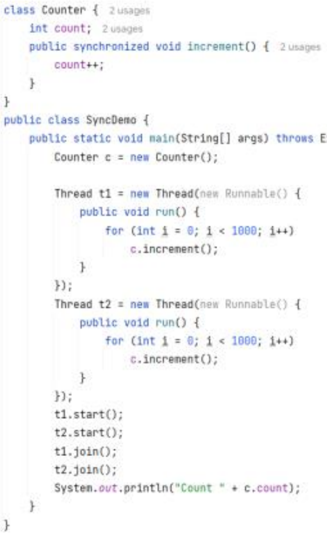
```
public class CollectionClass {
    public static void main(String[] args) {
```



```
t1.setPriority(Thread.MIN_PRIORITY);    // same as t1.setPriority(1);
t2.setPriority(Thread.MAX_PRIORITY);    // same as t2.setPriority(10);
t2.setPriority(Thread.NORM_PRIORITY);    // same as t2.setPriority(5);
```

Synchronized Method

- Multiple methods can access the same method at the same time, that's why we need to synchronized the method.
- Synchronized methods ensure thread safety by preventing interference between multiple threads accessing the same method simultaneously.

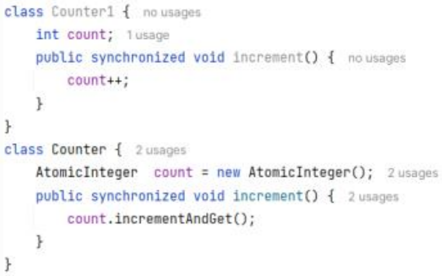


```
class Counter {  2 usages
    int count;  2 usages
    public synchronized void increment() {  2 usages
        count++;
    }
}
public class SyncDemo {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; i++)
                    c.increment();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; i++)
                    c.increment();
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Count " + c.count);
    }
}
```

- The issue is resolved by making the "increment" method synchronized, allowing only one thread to access it at a time.

Thread Safety

- Thread safety is an important concept in Java that deals with the issue of multiple threads accessing shared data simultaneously.
- Threads allow for concurrent execution of functions, which can improve performance by utilizing multiple cores.
- When multiple threads access shared data and attempt to change its values, problems can arise, leading to inconsistent results.
- In order to achieve thread safety, one approach is to use the synchronized keyword, which ensures that only one thread can access a method at a time.
- Another approach is to use atomic classes like AtomicInteger, which provide atomic operations for thread-safe increments.

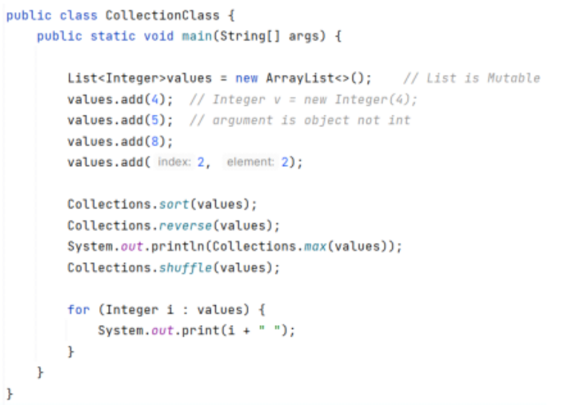


```
class Counter1 {  no usages
    int count;  1 usage
    public synchronized void increment() {  no usages
        count++;
    }
}
class Counter {  2 usages
    AtomicInteger  count = new AtomicInteger();  2 usages
    public synchronized void increment() {  2 usages
        count.incrementAndGet();
    }
}
```

Var keyword : New Java 10 Features

- LVTI (Local Variable Type Inference) is a feature in Java 10 that allows using local variables without explicitly declaring their data type.
- It is not applicable to instance variables, only to local variables used within methods.
- The keyword "var" is used instead of specifying the data type when declaring local variables.
- The purpose of LVTI is to improve code readability and maintainability.
- The type of the variable is still determined at compile time, so Java remains a statically typed language.
- Variables declared with "var" must be initialized with a value.

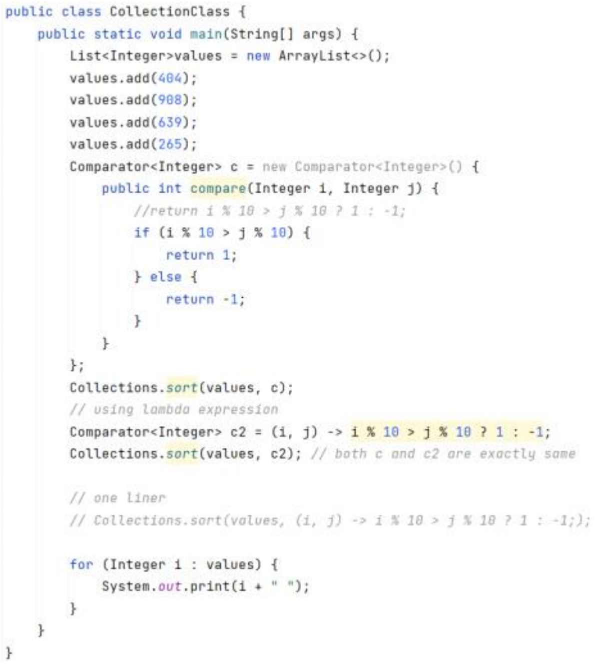
Collection Class



```
public class CollectionClass {
    public static void main(String[] args) {

        List<Integer>values = new ArrayList<>();    // List is Mutable
        values.add(4);  // Integer v = new Integer(4);
        values.add(5);  // argument is object not int
        values.add(8);
        values.add( index: 2,  element: 2);

        Collections.sort(values);
        Collections.reverse(values);
        System.out.println(Collections.max(values));
        Collections.shuffle(values);

        for (Integer i : values) {
            System.out.print(i + " ");
        }
    }
}
```

Comparator Interface

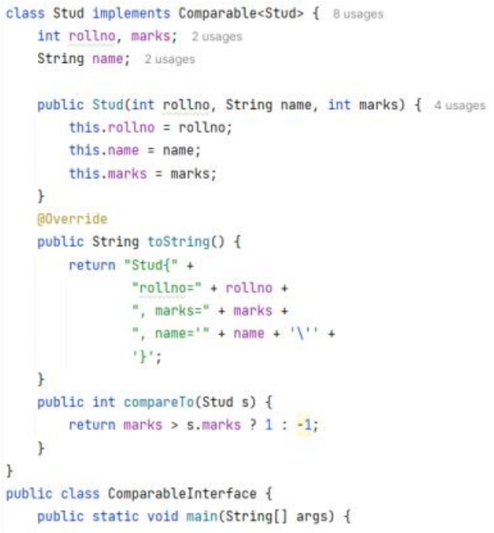
- The Collections.sort() method is used to sort elements, and it relies on the Comparator interface for defining the sorting logic.
- To override the default logic, a custom Comparator object needs to be created and passed to the sort() method.
- Creating a Comparator object requires implementing the compare() method.
- So Implements a class that implements the interface needs to be created.
- Or create an anonymous class that implements the Comparator interface.
- Lambda expressions allow for shorter code, optional type specification, and optional return statements.



```
public class CollectionClass {
    public static void main(String[] args) {
        List<Integer>values = new ArrayList<>();
        values.add(404);
        values.add(908);
        values.add(639);
        values.add(265);
        Comparator<Integer> c = new Comparator<Integer>() {
            public int compare(Integer i, Integer j) {
                //return i % 10 > j % 10 ? 1 : -1;
                if (i % 10 > j % 10) {
                    return 1;
                } else {
                    return -1;
                }
            }
        };
        Collections.sort(values, c);
        // using lambda expression
        Comparator<Integer> c2 = (i, j) -> i % 10 > j % 10 ? 1 : -1;
        Collections.sort(values, c2); // both c and c2 are exactly same

        // one liner
        // Collections.sort(values, (i, j) -> i % 10 > j % 10 ? 1 : -1;);

        for (Integer i : values) {
            System.out.print(i + " ");
        }
    }
}
```

Comparable Interface

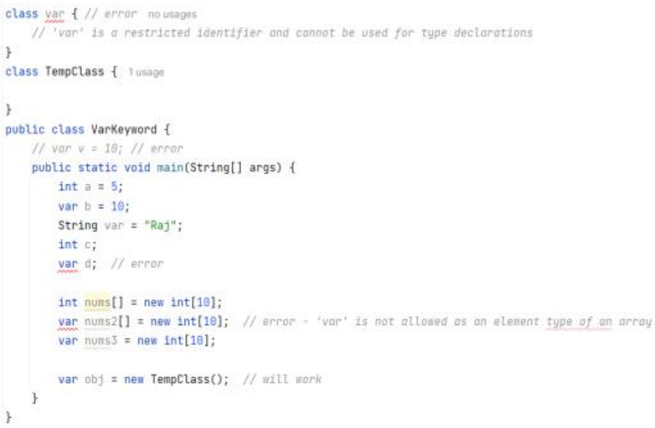
- The Comparable interface has a single method called compareTo(), which compares objects based on a specified criterion.



```
class Stud implements Comparable<Stud> {  8 usages
    int rollno, marks;  2 usages
    String name;  2 usages

    public Stud(int rollno, String name, int marks) {  4 usages
        this.rollno = rollno;
        this.name = name;
        this.marks = marks;
    }
    @Override
    public String toString() {
        return "Stud{" +
                "rollno=" + rollno +
                ", marks=" + marks +
                ", name='" + name + '\'' +
                '}';
    }
    public int compareTo(Stud s) {
        return marks > s.marks ? 1 : -1;
    }
}
public class ComparableInterface {
    public static void main(String[] args) {
```

variables.

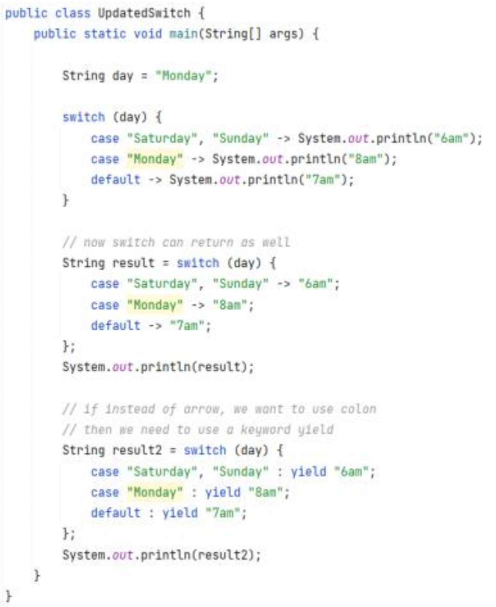
- The purpose of LVTI is to improve code readability and maintainability.
- The type of the variable is still determined at compile time, so Java remains a statically typed language.
- Variables declared with "var" must be initialized with a value.
- "Var" can be used as a variable name, but not as a class name.
- "Var" can be used when creating arrays or objects of a class.



```
class var { // error  no usages
    // 'var' is a restricted identifier and cannot be used for type declarations
}
class TempClass {  1 usage

}
public class VarKeyword {
    // var v = 10; // error
    public static void main(String[] args) {
        int a = 5;
        var b = 10;
        String var = "Raj";
        int c;
        var d;  // error

        int nums[] = new int[10];
        var nums2[] = new int[10];  // error - 'var' is not allowed as an element type of an array
        var nums3 = new int[10];

        var obj = new TempClass();  // will work
    }
}
```

Updated version of Switch Statement and Expression

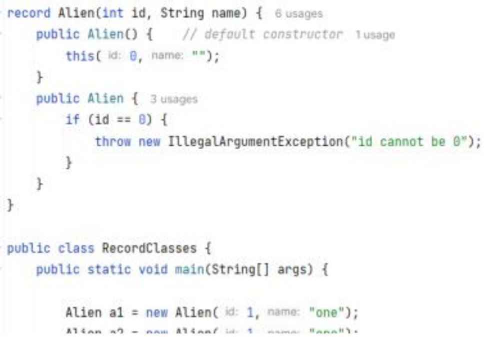
- The old switch statement required the use of brackets and break statements.
- The new switch statement allows for the omission of brackets and the use of an arrow (->) instead of a colon (:).
- To use switch as an expression, the arrow (->) or the keyword "yield" (for returning a value with colon syntax) can be used instead of a colon (:).



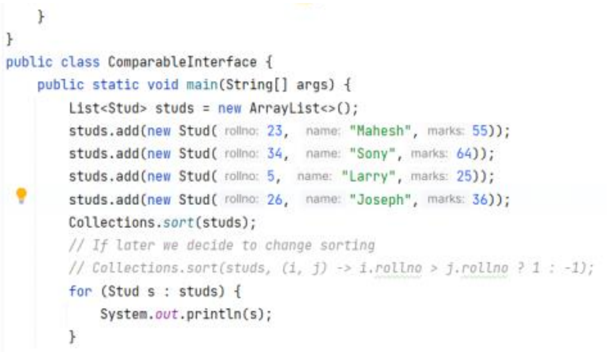
```
public class UpdatedSwitch {
    public static void main(String[] args) {

        String day = "Monday";

        switch (day) {
            case "Saturday", "Sunday" -> System.out.println("6am");
            case "Monday" -> System.out.println("8am");
            default -> System.out.println("7am");
        }

        // now switch can return as well
        String result = switch (day) {
            case "Saturday", "Sunday" -> "6am";
            case "Monday" -> "8am";
            default -> "7am";
        };
        System.out.println(result);

        // if instead of arrow, we want to use colon
        // then we need to use a keyword yield
        String result2 = switch (day) {
            case "Saturday", "Sunday" : yield "6am";
            case "Monday" : yield "8am";
            default : yield "7am";
        };
        System.out.println(result2);
    }
}
```

Record Classes | Java 17 Features

- Java 17 introduces the concept of record classes to simplify data storage classes.
- Record classes are defined using the "record" keyword, followed by the class name and variables.
- Record classes automatically generate constructors, getters, "equals," and "hashCode" methods.
- They are concise, immutable, and focused solely on data storage, and their variables are private and final by default.
- Record classes can implement interfaces but cannot extend other classes.



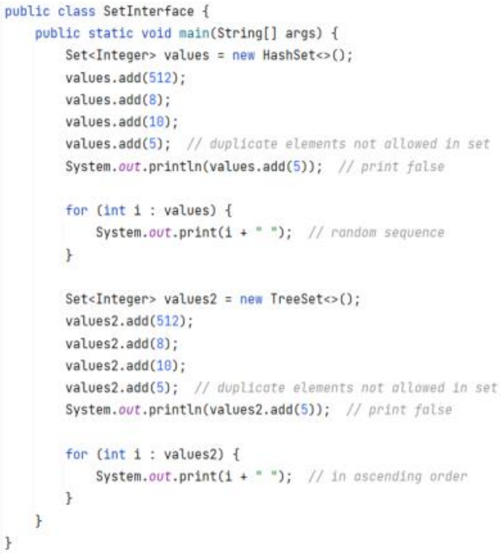
```
record Alien(int id, String name) {  6 usages
    public Alien() {    // default constructor  1 usage
        this( id: 0, name: "");
    }
    public Alien {  3 usages
        if (id == 0) {
            throw new IllegalArgumentException("id cannot be 0");
        }
    }
}

public class RecordClasses {
    public static void main(String[] args) {

        Alien a1 = new Alien( id: 1, name: "one");
        Alien a2 = new Alien( id: 1, name: "one");
```



```java
    }
}
public class ComparableInterface {
    public static void main(String[] args) {
        List<Stud> studs = new ArrayList<>();
        studs.add(new Stud( rollno: 23,  name: "Mahesh", marks: 55));
        studs.add(new Stud( rollno: 34,  name: "Sony", marks: 64));
        studs.add(new Stud( rollno: 5,  name: "Larry", marks: 25));
        studs.add(new Stud( rollno: 26,  name: "Joseph", marks: 36));
        Collections.sort(studs);
        // If later we decide to change sorting
        // Collections.sort(studs, (i, j) -> i.rollno > j.rollno ? 1 : -1);
        for (Stud s : studs) {
            System.out.println(s);
        }
```

Set Interface

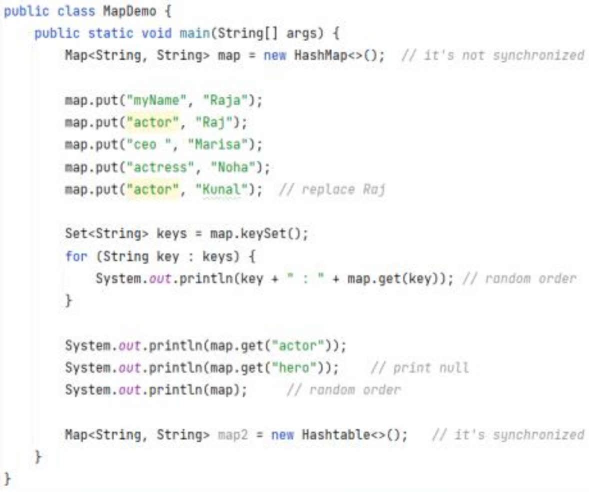
- In HashSet, duplicate element are not allowed and sequence is random
- In TreeSet, duplicate element are not allowed but sequence is in ascending order.



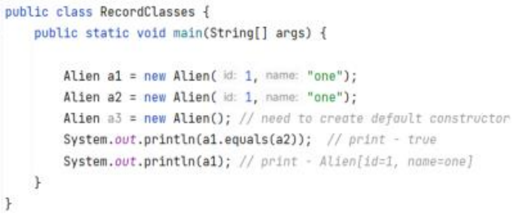
```java
public class SetInterface {
    public static void main(String[] args) {
        Set<Integer> values = new HashSet<>();
        values.add(512);
        values.add(8);
        values.add(10);
        values.add(5);  // duplicate elements not allowed in set
        System.out.println(values.add(5));  // print false

        for (int i : values) {
            System.out.print(i + " ");  // random sequence
        }

        Set<Integer> values2 = new TreeSet<>();
        values2.add(512);
        values2.add(8);
        values2.add(10);
        values2.add(5);  // duplicate elements not allowed in set
        System.out.println(values2.add(5));  // print false

        for (int i : values2) {
            System.out.print(i + " ");  // in ascending order
        }
    }
}
```

Map Interface

- Map interface is used to store key-value pairs.
- Values are added to the map using the "put" method.
- Type safety can be achieved by using generics, specifying the types for keys and values
- The sequence of values in a map is not guaranteed and may not follow the order of insertion.
- Values can be retrieved from the map using the "get" method and specifying the key.
- The keySet() method returns a set of keys in the map, which can be used to iterate over the map.
- HashMap is the preferred choice for implementing maps, while Hashtable provides thread safety but is less commonly used.



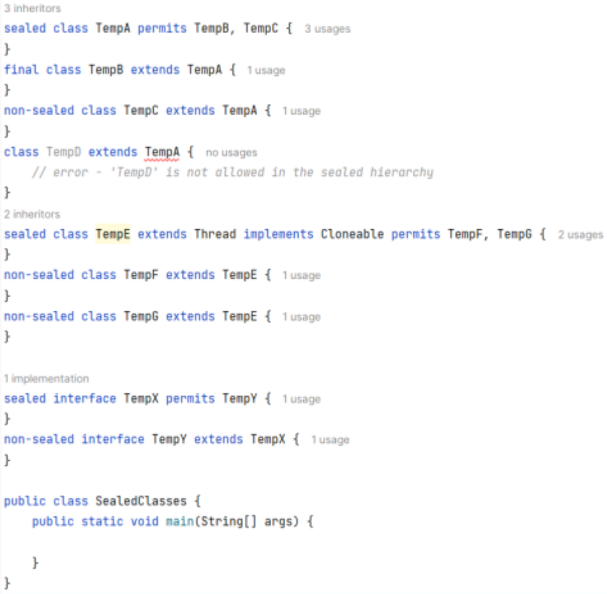
```java
public class MapDemo {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();  // it's not synchronized

        map.put("myName", "Raja");
        map.put("actor", "Raj");
        map.put("ceo ", "Marisa");
        map.put("actress", "Noha");
        map.put("actor", "Kunal");  // replace Raj

        Set<String> keys = map.keySet();
        for (String key : keys) {
            System.out.println(key + " : " + map.get(key)); // random order
        }

        System.out.println(map.get("actor"));
        System.out.println(map.get("hero"));    // print null
        System.out.println(map);     // random order

        Map<String, String> map2 = new Hashtable<>();   // it's synchronized
    }
}
```



```java
public class RecordClasses {
    public static void main(String[] args) {

        Alien a1 = new Alien( id: 1, name: "one");
        Alien a2 = new Alien( id: 1, name: "one");
        Alien a3 = new Alien(); // need to create default constructor
        System.out.println(a1.equals(a2));  // print - true
        System.out.println(a1); // print - Alien[id=1, name=one]
    }
}
```

Sealed Classes | Java 17 Features

- Sealed classes are introduced to restrict inheritance in Java.
- Final classes cannot be inherited by any class.
- Sealed classes provide a way to have limited inheritance by specifying which subclasses or subinterfaces can inherit from a particular class or interface.
- The sealed keyword is used to make a class sealed.
- The "permits" keyword is used to specify which classes are allowed to inherit from the sealed class.
- Sealed classes can be final, sealed, or non-sealed.



```java
3 inheritors
sealed class TempA permits TempB, TempC {  3 usages
}
final class TempB extends TempA {  1 usage
}
non-sealed class TempC extends TempA {  1 usage
}
class TempD extends TempA {  no usages
    // error - 'TempD' is not allowed in the sealed hierarchy
}
2 inheritors
sealed class TempE extends Thread implements Cloneable permits TempF, TempG {  2 usages
}
non-sealed class TempF extends TempE {  1 usage
}
non-sealed class TempG extends TempE {  1 usage
}

1 implementation
sealed interface TempX permits TempY {  1 usage
}
non-sealed interface TempY extends TempX {  1 usage
}

public class SealedClasses {
    public static void main(String[] args) {

    }
}
```