**unit-3**

**Functional Interfaces:**

A functional interface is an interface that contains only one abstract method. Java 8 introduced the **@FunctionalInterface** annotation to explicitly mark interfaces as functional interfaces. This annotation is optional, but it is a good practice to use it for clarity and to ensure that the interface remains functional.

**=>** Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward

=> In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

**Example-:**

@FunctionalInterface

interface xyz1 {

    public void prints();

}

@FunctionalInterface

interface xyz2 {

    public int sum(int a, int b);

}

public class fninterface {

    public static void main(String[] args) {

        xyz1 p = new xyz1() {

            @Override

            public void prints() {

                {

                    System.out.println("hello world");

                }

            }

        };

        xyz2 s = new xyz2() {

            @Override

            public int sum(int a, int b) {

                return a + b;

            }

        };

        p.prints();

        System.out.println(s.sum(40, 50));

    }

}

**Lambda Expressions:**

Lambda expressions provide a concise way to represent anonymous functions (functions without a name). They enable you to treat functionality as a method argument, or to create simple instances of single-method interfaces (functional interfaces). Lambda expressions consist of parameters, an arrow (->), and a body.

**Syntax:**

(parameters) -> expression

**Example-1:**

@*FunctionalInterface*

interface MyFunctionalInterface {

*void* myMethod();

}

public class Main {

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

*MyFunctionalInterface* ob = () *->* System.out.println("calling myMethod function");

        ob.myMethod();

 }

}

**//output**

calling myMethod function

**Example-2:**

@*FunctionalInterface*

interface Square {

*int* calculate(*int* *x*);

}

public class test123 {

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

*int* a = 5;

        Square s = (*int* *x*) *->*{ return  *x* \* *x*;};

// parameter passed and return type must be same as defined in the   prototype

*int* ans = s.calculate(a);

        System.out.println(ans);

    }

}

**// output**

**// 25**

**example-3:**

@FunctionalInterface

interface xyz1{

 public void prints();

}

@FunctionalInterface

interface xyz2{

 public int sum(int a,int b);

}

public class fninterface {

    public static void main(String[] args) {

        xyz1 p=()->{

              System.out.println("hello world");

        };

       xyz2 sum=(i,j)->{return i+j;};

       p.prints();

        System.out.println(sum.sum(3, 4));

    }

}

**hello world**

**7**

**Example-4**

@*FunctionalInterFace*

interface xyz1{

    abstract public *void* f1();

}

class xyz3 {

  xyz1 ob=() *->*{

    System.out.println("hello;");

  };

}

public class facto1 {

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

      xyz3 ob1=new xyz3();

       ob1.ob.f1();// hello

    }

}

**method references :**

In Java, method references provide a way to refer to methods or constructors without invoking them. They are a shorthand notation that can be used to simplify lambda expressions when the lambda body simply calls an existing method. Method references are often more readable and concise than equivalent lambda expressions.

There are four types of method references:

**1.Reference to a static method:** ContainingClass::staticMethodName

**2.Reference to an instance method of a particular object:** containingObject::instanceMethodName

**3.Reference to an instance method of an arbitrary object of a particular type:** ContainingType::methodName

**4.Reference to a constructor:** ClassName::new

**Example:**

import java.util.function.Function;

 class test1 {

 // Static method

 static Integer f1(String *n*) {

    return Integer.parseInt(*n*);

}

// Instance method

String f2(String *str*) {

    return *str*.toUpperCase();

}

static *boolean* check\_even(*int* *n*) {

   if (*n* % 2 == 0) {

        return true;

    } else {

        return false;

    }

}

    }

class test2 {

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

        // Reference or object to a static method

        Function<String, Integer> o1 = test1::f1;

        System.out.println(o1.apply("123")); // Output: 123

        // make object 1st bcoz non-static

        test1 o2 = new test1();

        Function<String, String> o3 = o2::f2;

        System.out.println(o3.apply("hello")); // Output: "HELLO"

        Function<Integer,Boolean> o4 = test1::check\_even;

        System.out.println(o4.apply(45));

    }

}

**Example-2:**

import java.util.function.Function;

class Person {

    private String name;

   public Person(String *name*) {

        this.name = *name*;

    }

  public String getName() {

        return name;

    }

}

public class method\_ref {

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

         // Using constructor reference to create a Function that takes a String and

         // returns a Person

        Function<String, Person> ob = Person::new;

  // Creating a new Person instance using the function

        Person person = ob.apply("ajay");

        System.out.println(person.getName());  // Output: ajay

    }

}

**Example-3:**

import java.util.function.Function;

class method\_ref\_lamda{

  //static fn

   public static String reverse(String *str*){

       String str1="";

       for (*int* i= *str*.length()-1;i>=0; i--) {

            str1=str1+*str*.charAt(i);

       }

       return str1;

   }

   //instance fn

   //////////////////////////////

   public *int* fibo(*int* *n*){//take only one parameter

*int* f=1;

       for (*int* i = 1; i <=*n*; i++) {

        f=f\*i;

       }

      return f;

   }

   //constructer

   public method\_ref\_lamda(*int* *num*){

      System.out.println("it is constructer "+*num*);

   }

}

public class fninterface {

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

///use method reference to static

        Function<String,String> ob1=method\_ref\_lamda::reverse;

           System.out.println(ob1.apply("abcd"));//dcba

        //////////////////////

        //reference to instance

        method\_ref\_lamda ob2=new method\_ref\_lamda(5);//random value for this

        Function<Integer,Integer> ob3=ob2::fibo;//but not used value

        System.out.println(ob3.apply(5));//120

        //////////////////////////

        // refernce to constructer

        Function<Integer,method\_ref\_lamda> o4=method\_ref\_lamda::new ;

        o4.apply(60);// it is constructer 60

        //////////////////////////////////////////////////////////////////

}

    }

}

**Stream API**

The Stream API in Java provides a powerful and flexible way to process collections of objects. It allows you to express common operations (such as filtering, mapping, sorting, and reducing) on collections in a functional and declarative style. Streams enable efficient, parallel execution of operations, which can result in improved performance for large datasets.

**Stream:** A stream is a sequence of elements from a source (e.g., a collection) that supports aggregate operations. Streams do not store elements; they convey data from a source through a pipeline of operations.

**Intermediate Operations:** These operations transform a stream into another stream. Examples include filter, map, sorted, distinct, limit, and skip. Intermediate operations are lazy, meaning they do not process elements until a terminal operation is invoked.

**Terminal Operations:** These operations produce a result or side-effect. They are not streams and execute the stream pipeline to produce a result.

Examples include forEach, collect, reduce, count, min, max, anyMatch, allMatch, and noneMatch.

**Reduction Operations:** These operations combine all elements of a stream into a single result. Examples include reduce, collect, sum, average, count, min, and max.

**Parallel Streams:** Streams can be processed in parallel to improve performance for large datasets. Parallel streams use the parallel() method to enable parallel execution and the sequential() method to switch back to sequential execution.

**Stream API:**

The Stream API in Java allows for functional-style operations on streams of elements.

* **Stream Creation**: Streams can be created from collections using the **stream()** method or from arrays using the **Arrays.stream()** method.
* **Intermediate Operations**: These operations transform an existing stream into another stream. Common intermediate operations include **map()**, **filter()**, **sorted()**, **distinct()**, **flatMap()**, and **peek()**.
* **Terminal Operations**: These operations produce a result or side-effect. Common terminal operations include **forEach()**, **collect()**, **reduce()**, **count()**, **min()**, **max()**, and **anyMatch()**.
* **Lazy Evaluation**: Stream operations are evaluated lazily, meaning intermediate operations are only executed when a terminal operation is invoked. This allows for more efficient processing of large datasets.

**Example:-1**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamExample {

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

        List<String> words = Arrays.asList("hello", "world", "java", "programming");

        List<String> upperCaseWords = words.stream().

                              map(String::toUpperCase).

                              collect(Collectors.toList());

         System.out.println(upperCaseWords);//[HELLO, WORLD, JAVA, PROGRAMMING]

    } }

**Example:-2**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Streams {

    static *void* f1() {

        //////////////////////////////////////////////

        List<Integer> arr = Arrays.asList(4, 5, 6, 9);

        System.out.println(arr);

*int* sum = 0;

        for (Integer i : arr) {

            sum += i;

        }

        System.out.println("sum is " + sum);

        /////////////////////////////////////////////////////

        // m-2 for print list

        arr.forEach((Integer *n*) *->* {

            System.out.print(*n* + " ");

        });

        // m-3 for print list with Stream

        System.out.println();

        System.out.println("after using stream ");

        Stream<Integer> ob = arr.stream();

        ob.forEach((Integer *n*) *->* {

            System.out.print(*n* + " ");

        });

        // used some method of Stram Interface map,filter,reduce

        System.out.println();

        System.out.println("this is list2 ");

        List<Integer> arr2 = Arrays.asList(4, 5, 7, 6, 3, 10, 11, 13, 18);

        arr2.forEach(*n* *->* System.out.print(*n* + " "));

        System.out.println();

        Stream<Integer> r2 = arr2.stream().filter((*n*) *->* *n* % 2 == 0);

        System.out.println("result of r2 for even number ");

        r2.forEach((*n*) *->* System.out.print(*n* + " "));

        System.out.println();

        System.out.println("result of r3 for double of even numbers");

        Stream<Integer> r3 = arr2.stream().filter((*n*) *->* *n* % 2 == 0).map(*n* *->* *n* \* 2);

        r3.forEach((*n*) *->* System.out.print(*n* + " "));

        System.out.println("after using reduce function it will add all element ");

        Integer r4 = arr2.stream().filter((*n*) *->* *n* % 2 == 0).map(*n* *->* *n* \* 2).reduce(0, (*c*, *e*) *->* *c* + *e*);

        System.out.println("result of r4 " + r4);

        System.out.println("sorted list");

        Stream<Integer> r5 = arr2.stream().sorted();

        r5.forEach(*n* *->* System.out.print(*n* + " "));

    }

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

        f1();

    }

}

**//output**

**[4, 5, 6, 9]**

**sum is 24**

**4 5 6 9**

**after using stream**

**4 5 6 9**

**this is list2**

**4 5 7 6 3 10 11 13 18**

**result of r2 for even number**

**4 6 10 18**

**result of r3 for double of even numbers**

**8 12 20 36 after using reduce function it will add all element**

**result of r4 76**

**sorted list**

**3 4 5 6 7 10 11 13 18**

**Example-3:**

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

public class streams2 {

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

        List<String> names = Arrays.asList("ajay", "Bob", "Charlie");

*long* count = names.stream().count();

        System.out.println(count);// 3

        ////////////////////////////////////////////////

        List<Integer> numbers1 = Arrays.asList(3, 5, 1, 2, 4);

        Optional<Integer> min = numbers1.stream().min(Integer::compareTo);

        System.out.println(min);// Optional[1]

        /////////////////////////////////////////////////////////////////

        List<Integer> numbers2 = Arrays.asList(3, 5, 1, 2, 4);

        Optional<Integer> max = numbers2.stream().max(Integer::compareTo);

        System.out.println(max);// Optional[5]

        ///////////////////////////////////////////////////////////////////

        List<String> names1 = Arrays.asList("Alice", "Bob", "Charlie");

*boolean* hasAlice = names1.stream().anyMatch(*name* *->* *name*.equals("Alice"));

        System.out.println(hasAlice);// true

        //////////////////////////////////////////////////////////////

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

*int* sum = numbers.stream().reduce(0, Integer::sum);

        System.out.println(sum);// 15

        ////////////////////////////////////////////////////////////

        List<String> names2 = Arrays.asList("Alice", "Bob", "Charlie");

        List<Integer> nameLengths = names2.stream()

                .map(String::length)

                .collect(Collectors.toList());

        System.out.println(nameLengths);// [5, 3, 7]

        /////////////////////////////////////////////////////////////////

        List<String> names3 = Arrays.asList("Charlie", "Alice", "Bob");

        List<String> sortedNames = names3.stream()

                .sorted()

                .collect(Collectors.toList());

        System.out.println(sortedNames);// [Alice, Bob, Charlie]

        /////////////////////////////////////////////////////////////

        List<Integer> numbers4 = Arrays.asList(1, 2, 2, 3, 4, 4, 5);

        List<Integer> distinctNumbers = numbers4.stream()

                .distinct()

                .collect(Collectors.toList());

        System.out.println(distinctNumbers); // [1, 2, 3, 4, 5]

        ////////////////////////////////////////////////////////////////

        List<String> names4 = Arrays.asList("Alice", "Bob", "Charlie");

        List<String> processedNames = names4.stream()

                .peek(*name* *->* System.out.println("Processing: " + *name*))

                .map(String::toUpperCase)

                .collect(Collectors.toList());

        System.out.println(processedNames); // [ALICE, BOB, CHARLIE]

        ////////////////////////////////////////////////////////////

        List<List<String>> listOfLists = Arrays.asList(

                Arrays.asList("a", "b", "c"),

                Arrays.asList("d", "e", "f"),

                Arrays.asList("g", "h", "i"));

        List<String> flattenedList = listOfLists.stream()

                .flatMap(List::stream)

                .collect(Collectors.toList());

        System.out.println(flattenedList);  //[a, b, c, d, e, f, g, h, i]

    }

}

**forEach :**

In Java, the forEach method is commonly used to iterate over collections and perform operations on each element. It is part of the Iterable interface and was introduced in Java 8. This method is particularly useful with lambda expressions and method references to make the code more readable and concise.

 static *void* f1() {

        //////////////////////////////////////////////

        List<Integer> arr = Arrays.asList(4, 5, 6, 9);

 arr.forEach((Integer *n*) *->* {

            System.out.print(*n* + " ");

        });

}

**Default Methods:**

Default methods were introduced in Java 8 to enable the addition of new methods to interfaces without breaking existing implementations.

* **Why Default Methods?**: Prior to Java 8, adding a new method to an interface would require modifying all implementing classes, which is impractical for large codebases. Default methods solve this problem by providing a default implementation in the interface itself.
* **Interface Inheritance**: When a class implements multiple interfaces with default methods that have the same signature, the implementing class must provide its own implementation of the method, resolving the conflict.
* **Extension and Evolution**: Default methods enable interfaces to evolve over time without breaking existing code, facilitating the extension of APIs.

**Key Points:**

1. **Default Keyword**: A default method is declared with the default keyword.
2. **Implementation in Interface**: It provides a method implementation within the interface itself.
3. **Backward Compatibility**: It helps in maintaining backward compatibility when new methods are added to an interface.
4. **Optional Override**: Implementing classes may or may not override default methods. If they don't, the default implementation is used.

interface MyInterface {

    default *void* defaultMethod() {

        System.out.println("Default method implementation");

    } }

class MyClass implements MyInterface {

        // No need to implement defaultMethod()

}

public class StreamExample {

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

  MyClass obj = new MyClass();

     obj.defaultMethod(); // Output: Default method implementation

                }

            }

**1.Static Method in class :**

Static methods in Java belong to the class rather than to any specific instance of the class. They are associated with the class itself rather than with instances of the class.

* **Utility Functions**: Static methods are commonly used to define utility functions that perform common tasks. Since they do not require an instance of the class, they can be accessed directly using the class name.
* **Mathematical Operations**: Static methods are often used to define mathematical operations or conversion functions.
* **Factory Methods**: Static methods can also be used as factory methods to create instances of a class.

**2. Static methods in interfaces :**

**=>**Static methods in interfaces are similar to static methods in classes. They belong to the interface and not to instances of the interface. They can be called directly on the interface rather than on an instance of the interface.

**Key Points:**

1. **Static Keyword**: A static method is declared with the static keyword.
2. **Utility Methods**: They are often used to define utility methods.
3. **Cannot be Overridden**: Static methods cannot be overridden by implementing classes.
4. **Called on Interface**: They are called using the interface name.

**Example-1**:

interface MyStaticInterface {

    // Static method

    static *void* staticMethod() {

        System.out.println("This is a static method.");

    }

}

public class Main {

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

        // Calling static method using interface name

        MyStaticInterface.staticMethod(); // Outputs: This is a static method.

    }

}

Example-2

interface Innerdefult\_static {

   // default method

    default public *void* f1() {// for default must specify body

        System.out.println("default method in interface ");

    }

    // static method in interface can be called usings

    // interface.method\_name in main method

    public static *void* f2() {

        System.out.println("static method f2 in interface");

    }

    ///////// abstract method

    abstract public *void* f3();// no need to implementation in interface

}

class cyz implements Innerdefult\_static {

    @*Override*

    public *void* f1() {

        System.out.println("default1 method in class");// optional Override

        // bcoz default method

    }

  @*Override* // not optional must be overriden in class bcoz abstract

    public *void* f3() {

        System.out.println("abstract overriden method in class ");

    }

// here we can not write @Override bcoz f2 is separate static fn in

    // class cyz so static fn can not Override in class

    public static *void* f2() {

        System.out.println("separate static fn f2 in class ");

    }

}

public class defult\_static {

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

        cyz o = new cyz();

        o.f1();

        o.f3();

        Innerdefult\_static.f2();

        cyz.f2();}

}

//output

default1 method in class

abstract overriden method in class

static method f2 in interface

separate static fn f2 in class

**Base64 Encode and Decode:**

Base64 encoding and decoding is a common requirement when dealing with data that needs to be stored and transferred in a text format. Java provides built-in support for Base64 encoding and decoding in the java.util.Base64 class, introduced in Java 8.

Base64 Encoding:Encoding a byte array into a Base64 string involves converting binary data into a text representation using the Base64 scheme. This is useful for encoding data that needs to be included in URLs, XML, or JSON.

**Example:**

import java.util.Base64;

public class Base64Example {

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

        // Input byte array

        String originalInput = "Hello, World!";

*byte*[] inputBytes = originalInput.getBytes();

           // Encode to Base64

        String encodedString = Base64.getEncoder().encodeToString(inputBytes);

        System.out.println("Encoded String: " + encodedString);

    } }

//output

Encoded String: SGVsbG8sIFdvcmxkIQ==

**Base64 Decoding :**

Decoding a Base64 string back into a byte array involves converting the Base64 text representation back into binary data.

**Example:**

import java.util.Base64;

public class Base64Example {

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

        // Base64 encoded string

        String encodedString = "SGVsbG8sIFdvcmxkIQ==";

        // Decode from Base64

*byte*[] decodedBytes = Base64.getDecoder().decode(encodedString);

        String decodedString = new String(decodedBytes);

        System.out.println("Decoded String: " + decodedString);

    }

}

//output

Decoded String: Hello, World!

**URL and Filename Safe Base64 Encoding and Decoding:**

Java also provides URL and Filename safe variants of Base64 encoding and decoding, which replace + and / with - and \_ respectively, and omit padding characters. This is useful for encoding data to be included in URLs or filenames.

**Example:**

import java.util.Base64;

public class Base64UrlExample {

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

        // Input byte array

        String originalInput = "Hello, World!";

*byte*[] inputBytes = originalInput.getBytes();

        // Encode to URL and Filename safe Base64

        String encodedString = Base64.getUrlEncoder().encodeToString(inputBytes);

        System.out.println("Encoded URL String: " + encodedString);

        // Decode from URL and Filename safe Base64

*byte*[] decodedBytes = Base64.getUrlDecoder().decode(encodedString);

        String decodedString = new String(decodedBytes);

        System.out.println("Decoded URL String: " + decodedString);

    }

}

Try-with-resources :

The try-with-resources statement ensures that each resource is closed at the end of the statement. A resource is an object that must be closed after the program is finished with it (e.g., a file, a database connection).

Example:

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class test1 {

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

    // writing data with resources

    try (FileOutputStream ob1 = new FileOutputStream("xyz.txt")) {

      String str = "hello";

*byte*[] b = str.getBytes();

      ob1.write(b);//hello will write in xyz.txt file

    } catch (IOException e) {

      e.printStackTrace();

    }

    /// reading data

    try (FileInputStream ob = new FileInputStream("xyz.txt")) {

*int* data;

      while (true) {

        data = ob.read();

        if (data == -1) {

          break;

 }

        System.out.print((*char*) data + " ");

      }

    } catch (IOException e) {

      e.printStackTrace();

    }

  }

}

**Type Annotations**

Type annotations can be used to annotate the use of types. They were introduced in Java 8 and can be applied wherever types are used: declarations, generic types, casts, etc.

**Example:**

import java.lang.annotation.*Target*;

import java.lang.annotation.*Retention*;

import java.lang.annotation.ElementType;

import java.lang.annotation.RetentionPolicy;

@*Target*({ElementType.TYPE\_USE, ElementType.TYPE\_PARAMETER})

@*Retention*(RetentionPolicy.RUNTIME)

@interface *NonNull\_xyz* {

}

public class type\_annotation<@*NonNull\_xyz* T\_xyz> {

    public static @*NonNull\_xyz* String str="abc";

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

        @*NonNull\_xyz* String myStr = "Hello";

        System.out.println(myStr);//hello

        System.out.println(str);//abc

    }

}

In this example, the @NonNull annotation is applied to the type usage.

**Repeating Annotations:**

Repeating annotations allow you to apply the same annotation more than once to the same declaration or type use. This was introduced in Java 8.

**Example:**

import java.lang.annotation.*Retention*;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.*Repeatable*;

@*Retention*(RetentionPolicy.RUNTIME)

@*Repeatable*(*Schedules*.class)

@interface *Schedule* {

    String day();

}

@*Retention*(RetentionPolicy.RUNTIME)

@interface *Schedules* {

*Schedule*[] value();

}

public class type\_annotation {

    @*Schedule*(day = "Monday")

    @*Schedule*(day = "Wednesday")

    @*Schedule*(day = "Friday")

    public *void* doWork() {

        System.out.println("Working...");

    }

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

*Schedule*[] i = type\_annotation.class.getDeclaredMethods()[0]

                        .getAnnotationsByType(*Schedule*.class);

        for (*Schedule* schedule : i) {

            System.out.println("Scheduled on: " + schedule.day());

        }

    }

}

In this example, the @Schedule annotation is used multiple times on the doWork method.

**5. Java Module System:**

The Java Platform Module System (JPMS), introduced in Java 9, allows you to modularize your code. It provides more reliable configuration, strong encapsulation, and improved security and performance.

**Example:**

Module Declaration (module-info.java):

module com.example.myapp {

    requires java.sql;

    exports com.example.myapp.api;

}

This module-info.java file declares a module named com.example.myapp that requires the java.sql module and exports the com.example.myapp.api package.

**6. Diamond Syntax**

The diamond syntax (<>) simplifies the instantiation of generic classes. Introduced in Java 7, it allows the compiler to infer the type arguments.

**Example-1:**

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class DiamondSyntaxExample {

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

        // Using diamond syntax

        Map<String, List<String>> myMap = new HashMap<>();

        // Without diamond syntax (pre-Java 7)

        // Map<String, List<String>> myMap = new HashMap<String, List<String>>();

        System.out.println("Created a Map using diamond syntax: " + myMap);

    }

}

 **ForEach Method**: Simplifies iteration over collections and streams using lambda expressions and method references.

 **Try-with-resources**: Ensures that resources are automatically closed after use, improving resource management.

 **Type Annotations**: Allow annotations to be applied to any use of a type, providing more fine-grained metadata.

 **Repeating Annotations**: Enable the same annotation to be applied multiple times to the same element.

 **Java Module System**: Provides a powerful way to modularize applications, improving encapsulation and maintainability.

 **Diamond Syntax**: Simplifies the syntax for generics, reducing boilerplate code.

Inner Anonymous Class :

An inner anonymous class in Java is a class declared and instantiated at the same time without a name. It's often used when you need to override methods of a class or interface without creating a separate subclass.

**Example:**

public class Outer {

  public *void* display() {

      // Inner anonymous class implementing Runnable interface

      Runnable r = new Runnable() {

          @*Override*

          public *void* run() {

              System.out.println("Inside inner anonymous class");

          }

      };

      Thread t = new Thread(r);

      t.start();

  }

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

      Outer outer = new Outer();

      outer.display();

  }

}

**Example of inner class:**

public class OuterClass {

    public *int* outerData;

    public OuterClass(*int* *data*) {

        this.outerData = *data*;

    }

    // Inner class

    public class InnerClass {

        public *int* innerData;

        public InnerClass(*int* *data*) {

            this.innerData = *data*;

        }

        public *void* display() {

            System.out.println("Outer Data: " + outerData);

            System.out.println("Inner Data: " + innerData);

            display2();

             }

        public *void* display2(){

            System.out.println("display2");

        }

        public *double* display3(){

            return 18;

        }

    }

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

       // OuterClass outerObj = new OuterClass(10); m-2

       // OuterClass.InnerClass innerObj = outerObj.new InnerClass(20);

        //innerObj.display();

        //m-2 for creating object

       new OuterClass(7).new InnerClass(99).display();//display2

       System.out.println(new OuterClass(89).new InnerClass(89).display3());//18.0

    }

}

2. Local Variable Type Inference (var)

Introduced in Java 10, local variable type inference allows you to declare local variables without explicitly specifying their type. The compiler infers the type from the initializer.

**Example:**

public class LocalVariableTypeInferenceExample {

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

      var message = "Hello, World!"; // Compiler infers String type

      var number = 10; // Compiler infers int type

      System.out.println(message);

      System.out.println(number);

  }

}

Switch Expressions :

Switch expressions were introduced as a preview feature in Java 12 and became a standard feature in Java 14. They provide a more concise and expressive way to write switch statements, allowing them to return a value.

**Example-1:**

import java.util.\*;

public class switch1 {

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

        Scanner sc = new Scanner(System.in);

        System.out.println("enter number ");

        String str = sc.nextLine();

        switch (str) {

            case "monday","thursday":

                System.out.println("6AM");

                break;

            case "sunday":

                System.out.println("7AM");

            case "tuesday":

                System.out.println("9AM");

            default:

                System.out.println("sleep permanent");

                break;

        }

    }

}

**Example-2:**

If you do not want to put break keyword in all statement then use arrow .

import java.util.\*;

public class switch1 {

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

        Scanner sc = new Scanner(System.in);

        System.out.println("enter number ");

        String str = sc.nextLine();

        switch (str) {

            case "monday","thursday"*->*

                System.out.println("6AM");

            case "sunday"*->*

                System.out.println("7AM");

            case "tuesday"*->*

                System.out.println("9AM");

            default*->*

                System.out.println("sleep permanent");

        }

    }

}

**Example-3 :**

public class SwitchExpressionExample {

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

*int* day = 3;

      String dayName = switch (day) {

          case 1 *->* "Monday";

          case 2 *->* "Tuesday";

          case 3 *->* "Wednesday";

          case 4 *->* "Thursday";

          case 5 *->* "Friday";

          default *->* "Unknown";

      };

      System.out.println("Day: " + dayName);

  }

}

4. Yield Keyword

The yield keyword was introduced in Java 13 as a preview feature and made final in Java 14. It's used in switch expressions to return a value from a switch branch.

Example-1:

public class YieldExample {

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

*int* day = 3;

      String dayName = switch (day) {

          case 1, 2, 3 *->* {

              yield "Weekday";

          }

          case 4, 5 *->* {

              yield "Weekend";

          }

          default *->* {

              yield "Unknown";

          }

      };

      System.out.println("Day: " + dayName);

  }

}

Example-2:

import java.util.\*;

public class switch1 {

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

        Scanner sc = new Scanner(System.in);

        System.out.println("enter number ");

        String str = sc.nextLine();

        String new\_str = "";

        new\_str = switch (str) {

            case "monday", "thursday": yield "6AM";

            case "sunday": yield "7AM";

            case "tuesday": yield "9AM";

           default: yield "sleep permanent";

        };

        System.out.println(new\_str);

    }

}

**Text Blocks :**

Text blocks, introduced in Java 13 as a preview feature and made final in Java 15, provide a multi-line string literal without the need for escape sequences.

Example:

public class switch1 {

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

        String html = """

                this is statement-1// no need to \n

                this is statement-2

                this is statement-3

                this is statement-4    """;

        System.out.println(html);

    }

}

//output

this is statement-1// no need to

this is statement-2

this is statement-3

this is statement-4

**Records:**

Records, introduced in Java 14, provide a compact way to declare classes that are transparent holders for immutable data. They implicitly define methods like equals(), hashCode(), and toString().

Example:

public record Person(String *name*, *int* *age*) {}

public class RecordsExample {

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

        Person person = new Person("Alice", 30);

        System.out.println(person.name());

        System.out.println(person.age());

        System.out.println(person); // toString() is automatically generated

    }

}

Sealed Classes :

Sealed classes, introduced in Java 15, restrict the subclasses that can extend them. They provide more control over inheritance and help in creating hierarchies that are easier to understand and maintain.

Example-1:

sealed class ram permits shyam,sohan{ //sealed can be  extended with permission

*void* ram(){

    System.out.println("ram in sealed class");

 }

}

final class shyam  extends ram{ // final can not be extended

*void* shyam(){

        System.out.println("shyam in final class ");

    }

}

non-sealed class sohan extends ram{ //do final or non-sealed

*void* sohan(){

        System.out.println("sohan in non-sealed class");

    }

}

public class switch1 {

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

        sohan o1=new sohan();

        o1.ram();

       // o1.shyam(); can not acess

        o1.sohan();

        shyam o2=new shyam();

        o2.shyam();

        o2.ram();

    }

}

These features enhance the expressiveness, readability, and flexibility of Java code, making it easier to develop and maintain robust applications.