E-BOOK

UNIT -3

PDF

OOPS WITH JAVA

written By : Abhay Kumar Singh

Switch Expressions

- Introduced in Java 12 as a preview feature and made standard in Java 14, enhance the traditional switch statement.
- **Key Features**
  1. No need for break statements.
  2. Use -> for case labels.
  3. Can return a value directly assignable to a variable.
  4. Multiple case labels can be combined using commas.

Comparison Table

Feature	Traditional Switch Statement	Switch Expression
Syntax	Uses `case` and `break` for each branch	Uses `->` for case labels
Break Statements	Requires `break` to prevent fall-through	No need for `break` statements
Returning a Value	Cannot directly return a value	Can directly return a value
Multiple Case Labels	Uses separate `case` statements for each label	Allows multiple labels combined with commas
Verbosity	More verbose with repeated `case` and `break`	More concise and readable

Ex:Traditional Switch Statements

```
public class TraditionalSwitchExample {
   public static void main(String[] args) {
      String day = "MONDAY";
      String result;

      switch (day) {
         case "MONDAY":
         case "FRIDAY":
         case "SUNDAY":
            result = "Weekday";
            break;
         case "TUESDAY":
            result = "Tuesday";
            break;
         case "THURSDAY":
         case "SATURDAY":
            result = "Almost the weekend";
            break;
         case "WEDNESDAY":
            result = "Midweek";
            break;
         default:
            throw new IllegalArgumentException("Invalid day: " + day);
      }

      System.out.println("Today is: " + result);
   }
}
```

Ex: Switch Expression

```
public class SwitchExpressionExample {
   public static void main(String[] args) {
      var day = "MONDAY";

      // Using switch expression
      var result = switch (day) {
         case "MONDAY", "FRIDAY", "SUNDAY" -> "Weekday";
         case "TUESDAY" -> "Tuesday";
         case "THURSDAY", "SATURDAY" -> "Almost the weekend";
         case "WEDNESDAY" -> "Midweek";
         default -> throw new IllegalArgumentException("Invalid day: " + day);
      };

      System.out.println("Today is: " + result);
   }
}
```

yield keyword

- The yield keyword is used within switch expressions to return a value from a case block.
- Useful when a case involves multiple statements and needs to return a result.
- Allows more complex expressions to be written in a readable and concise manner.

```
public class YieldKeywordExample {
   public static void main(String[] args) {
      String day = "MONDAY";

      // Using switch expression with yield
      String result = switch (day) {
         case "MONDAY", "FRIDAY", "SUNDAY" -> "Weekday";
         case "TUESDAY" -> {
            // Complex logic can be placed here
            System.out.println("Processing Tuesday");
            yield "Tuesday";
         }
         case "THURSDAY", "SATURDAY" -> "Almost the weekend";
         case "WEDNESDAY" -> {
            // Another example of using yield
            System.out.println("Processing Wednesday");
            yield "Midweek";
         }
         default -> throw new IllegalArgumentException("Invalid day: " + day);
      };

      System.out.println("Today is: " + result);
   }
}
```

Text blocks

- introduced in Java 13, simplify the creation of multi-line strings.
- They use triple double-quotes (""") to define the block.

```
public class TextBlockExample {
   public static void main(String[] args) {
      // Using a text block to define a multi-line string
      String str = """
                  {
                     "name": "John Doe",
                     "age": 30,
                     "city": "New York"
                  }
                  """;
      System.out.println(str);

   }
}
```

output :

```
{
"name": "John Doe",
 "age": 30,
"city": "New York"
}
```

Local Variable Type Inference

- introduced in Java 10, allows the compiler to infer the type of a local variable based on the initializer.
- The var keyword is used to declare the variable
- which can make the code more concise and readable.
- **Example:**

```java
import java.util.List;
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        // Using 'var' to declare local variables with type inference
        var message = "Hello, World!";  // inferred as String
        var number = 42;                // inferred as int
        var list = new ArrayList<String>(); // inferred as ArrayList<String>

        list.add("Java");
        list.add("Type Inference");

        // Using 'var' in a for loop
        for (var item : list) {
            System.out.println(item);
        }

        // Using 'var' in a lambda expression
        var sum = add(10, 20);
        System.out.println("Sum: " + sum);
    }

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

output:

```
Java
Type Inference
Sum: 30
```

Records

- Records, introduced in Java 14, provide a concise way to create immutable data classes.
- They reduce boilerplate code by automatically generating constructors, accessors, equals(), hashCode(), and toString() methods.

```java
public class RecordExample {
    // Defining a record
    public record Person(String firstName, String lastName, int age) {}

    public static void main(String[] args) {
        // Creating an instance of the record
        Person person = new Person("John", "Doe", 30);

        // Accessing fields
        System.out.println("First Name: " + person.firstName());
        System.out.println("Last Name: " + person.lastName());
        System.out.println("Age: " + person.age());

        // Automatically generated toString method
        System.out.println("Person Details: " + person);

        // Equality check
        Person anotherPerson = new Person("John", "Doe", 30);
        System.out.println("Are they equal? " +
person.equals(anotherPerson));
    }
}
```

Sealed Classes

- introduced in Java 15 as a preview feature and becoming a standard feature in later versions
- allow you to restrict which classes can extend or implement them.
- This provides more control over the inheritance hierarchy, enhancing encapsulation and maintaining a more predictable and controlled type system.

- **Example:**

```java
public class SealedClassShortExample {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);

        System.out.println(circle);
        System.out.println(rectangle);
    }
}
// Sealed class
sealed interface Shape permits Circle, Rectangle {}
// Permitted subclass 1
final class Circle implements Shape {
    private final double radius;
    public Circle(double radius) {
        this.radius = radius;  }

    public String toString() {
        return "Circle with radius " + radius;
    }  }

// Permitted subclass 2
final class Rectangle implements Shape {
    private final double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public String toString() {
        return "Rectangle with width " + width + " and height " + height;
    }
}
```

Diamond operator

- Diamond operator was introduced as a new feature in java SE 7.
- The purpose of diamond operator is to avoid redundant code by leaving the generic type in the right side of the expression

```java
// This is before Java 7. We have to explicitly mention generic type
// in the right side as well.
List<String> myList = new ArrayList<String>();

// Since Java 7, no need to mention generic type in the right side
// instead we can use diamond operator. Compiler can infer type.
List<String> myList = new ArrayList<>();
```

Example

```java
abstract class MyClass<T>{
    abstract T add(T num, T num2);
}
public class Main {
    public static void main(String[] args) {
        MyClass<Integer> obj = new MyClass<>() {
            Integer add(Integer x, Integer y) {
                return x+y;
            }
        };
        Integer sum = obj.add(100,101);
        System.out.println(sum);
    }
}
```

for more details :
https://beginnersbook.com/2018/05/java-9-anonymous-inner-classes-and-diamond-operator/

ForEach

- Allows iterating over elements of a collection or stream and performing an action on each element.
- Provides a more concise and readable alternative to traditional loops (for).

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

// Using forEach with a lambda expression
names.forEach(name -> System.out.println("Hello, " + name));
```

Base64 encoding and decoding

- It is useful techniques for converting binary data into a text format that is safe for transmission over text-based protocols like HTTP or storing data in text-based formats like XML or JSON.
- Java provides built-in support for Base64 encoding and decoding through the java.util.Base64 class, introduced in Java 8.
- Base64 encoding is commonly used for encoding binary data such as images, documents, or any binary file into a text-based format that can be easily transmitted or stored in a text format.

1.Base64 Encoding

To encode data into Base64 format in Java, follow these steps:

```
import java.util.Base64;

public class Base64Example {
    public static void main(String[] args) {
        String originalInput = "Hello, World!";
        // Encode
        String encodedString =
Base64.getEncoder().encodeToString(originalInput.getBytes());
        System.out.println("Encoded string: " + encodedString);
    }
}
```

output : Encoded string: SGVsbG8sIFdvcmxkIQ==

2.Base64 Decoding

To decode a Base64 encoded string back to its original form, use the Base64.getDecoder() method:

```
import java.util.Base64;

public class Base64Example {

    public static void main(String[] args) {
        String encodedString = "SGVsbG8sIFdvcmxkIQ==";
        // Decode
        byte[] decodedBytes = Base64.getDecoder().decode(encodedString);
        String decodedString = new String(decodedBytes);
        System.out.println("Decoded string: " + decodedString);
    }
}
```

output: Decoded string: Hello, World!

Functional interface

- A functional interface in Java is an interface that contains exactly one abstract method.
- It can have any number of default methods and static methods.
- This concept is crucial for leveraging lambda expressions and method references in Java's functional programming paradigm.
- **Example : functional interface**

```
@FunctionalInterface
interface Calculator {
    // Abstract method (necessary)
    int calculate(int a, int b);

    // Default method
    default void display() {
        System.out.println("Calculating...");
    }
    // Static method
    static void greet() {
        System.out.println("Hello from Calculator!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Using lambda expression to implement Calculator interface

        Calculator addition = (a, b) -> a + b;
        int result = addition.calculate(10, 5);  // result will be 15
        System.out.println("Result: " + result);

        // Calling default method
        addition.display();  // Output: Calculating...

        // Calling static method
        Calculator.greet();  // Output: Hello from Calculator!
    }
}
```

Lambda expressions

- Lambda expressions in Java offer concise syntax for defining anonymous functions.
- Defined using **(parameters) -> expression or { statements }.**
- They are anonymous and lack a name like regular methods or functions.
- Lambda expressions are primarily used to implement the abstract method(s) of functional interfaces.

Example : print list of elements using lambda with forEach

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        List<String> list=new ArrayList<String>(Arrays.asList("ankit", "mayank",
"irfan", "jai"));
        list.forEach((n)->System.out.println(n));
    }
}
```

Example : lambda without parameter

```
interface Sayable{
    public String say();
}

public class Main{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Example : lambda with parameter

```
interface Addable{
    int add(int a,int b);
}
public class Main{
    public static void main(String[] args) {
        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));
    }
}
```

Types of functional interface

Consumer: Consumes (uses) an input without returning any result.
Predicate: Tests a condition on an input and returns a boolean result.
Function: Transforms an input into an output of a different type.
Supplier: Supplies (provides) a result without taking any input.

```
import java.util.*;
import java.util.function.*;

public class Main {
   public static void main(String[] args) {
      // Example 1: Consumer
      List<String> names = new ArrayList<>();
      names.add("Alice");
      names.add("Bob");
      names.add("Charlie");

      Consumer<String> printName = (name) -> System.out.println("Hello, " + name);
      names.forEach(printName);

      // Example 2: Predicate
      Predicate<Integer> isPositive = num -> num > 0;
      System.out.println("Is 10 positive? " + isPositive.test(10)); // true
      System.out.println("Is -5 positive? " + isPositive.test(-5)); // false

      // Example 3: Function
      Function<Integer, String> convertToString = num ->
String.valueOf(num);
      String strNumber = convertToString.apply(123);
      System.out.println("Number as string: " + strNumber);

      // Example 4: Supplier
      Supplier<Integer> getRandomNumber = () -> {
         Random random = new Random();
         return random.nextInt(100);
      };
      int randomNumber = getRandomNumber.get();
      System.out.println("Generated random number: " + randomNumber);
   } }
```

Method references

- Method references provide a shorthand syntax for lambda expressions to refer to methods or constructors using :: operator.
- They can reference static methods, instance methods, and constructors.
- Method references improve code readability by reducing boilerplate code.
- **There are mainly three types of method references:**
- Static method references: **ContainingClass::staticMethodName**
- Instance method references: **object::instanceMethodName**
- Constructor references: **ClassName::new**

Example:static method reference

```
import java.util.function.Function;

public class MethodReferenceExample {
   // Static method to add two integers
   public static int add(int a, int b) {
      return a + b;
   }

   public static void main(String[] args) {
      // Using static method reference
      Function<Integer, Integer> adder = MethodReferenceExample::add;
      int result = adder.apply(5, 3);
      System.out.println("Result: " + result);  // Output: Result: 8
   }
}
```

Example: instance method reference

```
import java.util.function.Function;

public class MRExample {
   // Instance method to convert a String to uppercase
   public String toUpperCase(String str) {
      return str.toUpperCase();
   }

   public static void main(String[] args) {
      MRExample example = new MRExample();

      // Using instance method reference
      Function<String, String> convertToUpperCase = example::toUpperCase;

      String result = convertToUpperCase.apply("hello");
      System.out.println("Uppercase: " + result);  // Output: Uppercase: HELLO
   }
}
```

Example:constructor reference

```
import java.util.function.Function;

public class MRExample{

   private String value;
   // Constructor
   public MRExample(String value) {
      this.value = value;
   }
   // Instance method to return the value
   public String getValue() {
      return value;
   }

   public static void main(String[] args) {
      // Using constructor reference
      Function<String, MRExample> createInstance = MRExample::new;

      MRExample instance = createInstance.apply("Hello");
      System.out.println("Value: " + instance.getValue());  // Output: Value: Hello
   }
}
```

Stream API

- Streams provide functional-style operations for processing sequences of elements.
- They are created from collections, arrays, or methods like Stream.of.
- Operations like map, filter, reduce, forEach can be chained to process data.
- Streams support lazy evaluation, executing operations only when needed.
- Parallel streams utilize multiple cores for concurrent processing.

```
import java.util.Arrays;
import java.util.List;

public class StreamExample {

   public static void main(String[] args) {
      // Create a list of integers
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

      // Using Stream API to filter even numbers and print them
      numbers.stream()
           .filter(num -> num % 2 == 0) // Filter even numbers
           .forEach(System.out::println); // Print each even number
   }
}
output : 2 4 6 8 10
```

- **Static Method**: A method in an interface that can be called using the interface name, independent of any instance of the interface.
- **Default Method**: A method in an interface that provides a default implementation which can optionally be overridden by classes that implement the interface.

Example : static and default methods

```
// Interface with default and static methods
interface Vehicle {
   // Default method
   default void displayInfo() {
      System.out.println("Default Vehicle information");
   }

   // Static method
   static void honk() {
      System.out.println("Static method: Honk!");
   }
}

// Main class to demonstrate default and static method usage
public class DefaultStaticMethodExample {
   public static void main(String[] args) {
      // Using default method
      Vehicle vehicle = new Vehicle() {}; // Anonymous implementation
      vehicle.displayInfo(); // Output: Default Vehicle information

      // Calling static method using interface name
      Vehicle.honk(); // Output: Static method: Honk!
   }
}
```

Try-with-resources

- Simplifies and enhances resource management by automatically closing resources after they are no longer needed
- ensuring they are closed properly even if an exception occurs.
- **Syntax:**

```
try (ResourceType resource1 = initialization; ResourceType resource2 =
initialization) {
   // Use resources
} catch (ExceptionType e) {
   // Handle exception
} finally {
   // Resources are closed automatically at the end of the try block
}
```

Example:

```
import java.io.*;

public class Main {
   public static void main(String[] args) {
      String filePath = "example.txt";

      // Using try-with-resources to automatically close resources
      try (FileReader fileReader = new FileReader(filePath);
           BufferedReader bufferedReader = new BufferedReader(fileReader)) {
         String line;
         while ((line = bufferedReader.readLine()) != null) {
            System.out.println(line);
         }

      } catch (IOException e) {
         // Handle exception
         System.err.println("Error reading file: " + e.getMessage());
      }
   }
}
```

Annotations

- Annotations in Java are a form of metadata that provide data about a program but are not part of the program itself.
- They have no direct effect on the operation of the code they annotate. Annotations can be used for various purposes
- such as providing information for the compiler, runtime processing, or generating code.
- Annotations are created using the @ symbol followed by the annotation name

Built-in Annotations

- **@Override**: Indicates that a method declaration is intended to override a method declaration in a superclass.
- **@Deprecated**: Marks a method, class, or field as deprecated and should no longer be used.
- **@SuppressWarnings**: Instructs the compiler to suppress specific warnings.
- **@FunctionalInterface**: Indicates that the type declaration is intended to be a functional interface as defined by the Java Language Specification.

Type annotations

- It is powerful tools for adding metadata about type usage
- helping to improve code quality, readability, and maintainability.
- By defining custom annotations or using built-in ones
- you can enforce additional rules and constraints in your code, making it more robust and easier to maintain.

Built-in Type Annotations

- **@NonNull**: Indicates that a variable, parameter, or return type cannot be null.
- **@Nullable:** Indicates that a variable, parameter, or return type can be null.
- **@ReadOnly**: Indicates that a method does not modify the object.
- **@Tainted:** Indicates that a value may be tainted or untrusted.

Example:

```
import javax.annotation.Nullable;
import javax.annotation.Nonnull;

public class User {
   private @Nullable String fName;
   private @Nonnull String lName;
   public User(@Nonnull String lName) {
      this.lName = lName;
   }
   public @Nullable String getFName() { return fName; }
   public void setFName(@Nullable String fName) { this.fName = fName; }
   public @Nonnull String getLName() { return lName; }
   public void setLName(@Nonnull String lName) { this.lName = lName; }
}
```

Repeating annotations

- It allows multiple annotations of the same type to be applied to a single program element (class, method, field, etc.).
- This feature was introduced in Java 8.
- repeating annotations simplify and enhance the clarity of code when multiple instances of the same annotation type are needed on a single program element,
- **To use repeated annotations, you need to:**
  1. Define the repeatable annotation.
  2. Define a container annotation that holds an array of the repeatable annotations.
  3. Annotate the repeatable annotation with the @Repeatable meta-annotation, specifying the container annotation.

Example:

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

// Step 1: Define the repeatable annotation
@Repeatable(Schedules.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Schedule {
   String dayOfWeek();
   String time();
}

// Step 2: Define the container annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Schedules {
   Schedule[] value();
}

// Example class to demonstrate the usage of these annotations
public class Event {

   // Example method annotated with multiple @Schedule annotations
   @Schedule(dayOfWeek = "Monday", time = "10:00")
   @Schedule(dayOfWeek = "Wednesday", time = "12:00")
   @Schedule(dayOfWeek = "Friday", time = "14:00")
   public void weeklyMeeting() {
      System.out.println("Weekly meeting scheduled.");
   }

   // Main method to access annotations via reflection
   public static void main(String[] args) {
      try {
         Method method = Event.class.getMethod("weeklyMeeting");

         // Get all @Schedule annotations
         Schedule[] schedules =
method.getAnnotationsByType(Schedule.class);
         for (Schedule schedule : schedules) {
            System.out.println("Day: " + schedule.dayOfWeek() + ", Time: " +
schedule.time());
         }

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

Java Module System

- The Java Module System was introduced in Java 9.
- This major feature, part of Project Jigsaw, aimed to address long-standing issues related to the modularity, scalability, and maintainability of Java applications.
- With the introduction of modules, Java developers gained the ability to better organize their code and manage dependencies in a more reliable and maintainable way.

Example with two modules: `greetings` and `app`.

i. **Module greetings** :The greetings module provides a greeting message.

1.module-info.java

```
module com.example.greetings {
   exports com.example.greetings;
}
```

2.Greeter.java

```
package com.example.greetings;

public class Greeter {
   public String getGreeting() {
      return "Hello, Module System!";
   }
}
```

ii. Module app : The app module uses the greetings module to print a greeting message.

1.module-info.java

```
module com.example.app {
   requires com.example.greetings;
}
```

2.Main.java

```
package com.example.app;
import com.example.greetings.Greeter;

public class Main {
   public static void main(String[] args) {
      Greeter greeter = new Greeter();
      System.out.println(greeter.getGreeting());
   }
}
```

Project Structure

```
project-root/
|-- src/
|   |-- com.example.greetings/
|   |   |-- module-info.java
|   |   |-- com/example/greetings/Greeter.java
|   |-- com.example.app/
|       |-- module-info.java
|       |-- com/example/app/Main.java
|-- mods/
```

Compilation and Execution

1.Compile Modules:

```
>> javac -d mods/com.example.greetings
src/com.example.greetings/module-info.java
src/com.example.greetings/com/example/greetings/Greeter.java

>> javac -d mods/com.example.app --module-path mods
src/com.example.app/module-info.java
src/com.example.app/com/example/app/Main.java
```

2.Run the Application:

```
java --module-path mods -m com.example.app/com.example.app.Main
```