GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK
Approved by AICTE & Affiliated to AKTU

Department of Computer Science and Engineering

Object Oriented Programming with Java

(BCS-402)

Presentation Report

Session – 2023-24
(Even Semester)

Submitted To - Mr. Jitendra Singh	**Submitted By -**	
	Roll No.	**Name**
Date of Presentation: 21-06-2024 **Mode of Presentation: Chalk and Board**	**2201920100154**	**Kamakshi Verma**
	2201920100155	**Kanhiya Sharma**
	2201920100156	**Karan Gupta**
	2201920100157	**Kartik Verma**

Index

Java Records:-

Introduction:

Java records are a new language feature introduced in Java 14 (released in March 2020). They provide a concise and efficient way to define classes that primarily focus on holding data.

Key Benefits of Records:

- **Reduced Boilerplate Code:** Records automatically generate common methods like getters, constructors, `toString()`, `equals()`, and `hashCode()` based on the declared fields. This saves developers significant time and effort.

- **Improved Readability:** The clean syntax of records makes code easier to understand and maintain. Since getters and other methods are implicit, the focus is on the data itself.

- **Immutability by Default:** Records are immutable by default, meaning their fields cannot be changed after creation. This promotes thread safety and simplifies reasoning about program state. (Custom mutable records can be created if needed).

- **Value Objects:** Records are well-suited for representing value objects, which are essentially data holders with well-defined behavior.

When to Use Records:

- **Data Transfer Objects (DTOs):** Records are ideal for transferring data between different parts of your application. Their conciseness and immutability make them a perfect fit for this use case.

- **Domain Objects (Simple Cases):** For domain objects that primarily hold data, records can offer a cleaner and more efficient alternative to traditional classes.

- **Value Objects:** As mentioned earlier, records excel at representing value objects with well-defined behavior based on their data.

A simple example of java record usage:

```
record Person(String name, int age) {
// Getters, toString, equals, and hashCode are automatically generated
}
```

Explanation of above code:

This record defines a `Person` with two fields: `name` (String) and `age` (int). All the necessary methods for working with this data are implicitly generated by the compiler.

Things to consider while using records:-

- **Immutability:** While immutability is generally a good thing, it might not be suitable for all use cases. If you need to modify the data within a record, you'll need to create a new instance with the updated values.

- **Inheritance:** Records have limitations with inheritance compared to traditional classes. Carefully consider if inheritance is necessary for your record design.

Overall, Java records are a valuable addition to the language, promoting cleaner, more concise, and maintainable code for data-centric objects. They are not a replacement for traditional classes but offer a powerful alternative for specific scenarios.

Example program for using record:-

```java
record Product(String name, String description, double price) {
    // Optional method to format price with currency symbol
    public String getFormattedPrice() {
      return String.format(format:"$%.2f", price); // Format price with two decimal places
    }
  }

public class Main {
    Run | Debug
    public static void main(String[] args) {
      // Create a Product instance
      Product laptop = new Product(name:"Laptop", description:"Powerful laptop for work and
      play", price:799.99);

      // Call the getFormattedPrice() method to get the formatted price
      String formattedPrice = laptop.getFormattedPrice();

      // Print the formatted price
      System.out.println("Price: " + formattedPrice);
    }
  }
```

Output:

```
[Running] cd "c:\Users\Kartik Verma\OneDrive
Price: $799.99

[Done] exited with code=0 in 1.363 seconds
```

Sealed Classes in Java

Sealed classes, introduced in Java 17 (released in September 2021), provide a mechanism to restrict the set of classes that can inherit from a specific class or implement an interface. This feature enhances code maintainability, security, and type safety in several ways.

Key benefits of sealed classes:

- **Controlled Inheritance:** Sealed classes prevent unintended subclasses from being created, promoting better code organization and reducing the risk of unexpected behavior.
- **Exhaustiveness Checking:** When used with pattern matching for switch expressions (another Java 17 feature), sealed classes enable the compiler to verify that all possible subtypes are handled within a switch block. This improves code reliability and prevents potential runtime errors.
- **Secure Library Design:** Library developers can leverage sealed classes to create secure hierarchies where only permitted implementations are allowed. This strengthens the integrity and predictability of libraries.

How Sealed Classes Work:

- **sealed Keyword:** A class or interface is declared as sealed to restrict its extensibility.
- **permits Clause:** The `sealed` declaration can optionally include a permits clause that specifies a list of allowed subclasses or implementing classes. These permitted classes must be in the same module as the sealed class.
- **Subclasses:** Permitted subclasses can be declared as final, sealed, or non-sealed.
    - final: The class cannot have any further subclasses.
    - sealed: The class can have further permitted subclasses within the same module.
    - non-sealed: The class allows further subclasses from any module (not recommended for most cases with sealed classes).

Example code:

```
sealed interface Shape permits Circle, Rectangle {
  double getArea();
}

final class Circle implements Shape {
  // Implementation for Circle
}

class Rectangle implements Shape {
  // Implementation for Rectangle
}
```

When to Use Sealed Classes:

- **Domain Modeling:** Sealed classes can effectively model closed sets of related concepts in your domain, ensuring that only expected implementations exist.
- **Library Design:** As mentioned earlier, sealed classes are valuable for creating secure and predictable library hierarchies.
- **Exhaustive Pattern Matching:** When using pattern matching for switch expressions, sealed classes help guarantee that all possible subtypes are handled within a switch block.

Things to Consider with Sealed Classes:

- **Limited Inheritance:** Sealed classes restrict extensibility, so carefully consider if inheritance is truly necessary for your design.
- **Module System Dependence:** Permitted subclasses must be in the same module as the sealed class. This might require adjustments if you're working with modular projects.

Overall, sealed classes are a powerful addition to the Java language, promoting well-defined class hierarchies, improved type safety, and enhanced code maintainability. They offer a valuable tool for creating robust and predictable software.

Example program for using sealed classes:

```java
// Define a sealed class
abstract sealed class Vehicle permits Car, Truck, Motorcycle {

    public abstract String type();
}
// Define a final class that extends the sealed class
final class Car extends Vehicle {
    private final int numberOfDoors;

    public Car(int numberOfDoors) {
        this.numberOfDoors = numberOfDoors;
    }

    @Override
    public String type() {
        return "Car with " + numberOfDoors + " doors";
    }
}
// Define a final class that extends the sealed class
final class Truck extends Vehicle {
    private final double loadCapacity;

    public Truck(double loadCapacity) {
        this.loadCapacity = loadCapacity;
    }

    @Override
    public String type() {
        return "Truck with load capacity of " + loadCapacity + " tons";
    }
}
```

```java
// Define a final class that extends the sealed class
final class Motorcycle extends Vehicle {
    private final boolean hasSidecar;

    public Motorcycle(boolean hasSidecar) {
        this.hasSidecar = hasSidecar;
    }

    @Override
    public String type() {
        return "Motorcycle" + (hasSidecar ? " with a sidecar" : "");
    }
}
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Vehicle car = new Car(numberOfDoors:4);
        Vehicle truck = new Truck(loadCapacity:15);
        Vehicle motorcycle = new Motorcycle(hasSidecar:true);

        System.out.println(car.type());
        System.out.println(truck.type());
        System.out.println(motorcycle.type());
    }
}
```

Ouput:

```
[Running] cd "c:\Users\Kartik Verma\OneDrive\Desktop\"
Car with 4 doors
Truck with load capacity of 15.0 tons
Motorcycle with a sidecar
```