Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 21

Lecture 21

- Functional Interfaces
- Lambda Expression

Functional Interfaces

- A functional interface is an interface that contains only one abstract method.
- They can have only one functionality to exhibit.
- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
- A functional interface can have any number of default methods.
- Runnable, ActionListener, and Comparable are some of the examples of functional interfaces.

- Functional Interface is additionally recognized as Single Abstract Method Interfaces.
- In short, they are also known as SAM interfaces.
- Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.
- Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward.

Functional interfaces are interfaces that ensure that they include precisely only one abstract method.

Functional interfaces are used and executed by representing the interface with an annotation called @FunctionalInterface.

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.

```
class Test {
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable() {
            @Override public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

// functional interface using lambda expressions

```
class Test {
    public static void main(String args[])
    {
        // lambda expression to create the object
        new Thread(() -> {
            System.out.println("New thread created");
        }).start();
    }
}
```

Lambda Expression in Java

Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value.

Lambda Expression Syntax

lambda operator -> body

There are three Lambda Expression Parameters are mentioned below:

- Zero Parameter
- Single Parameter
- Multiple Parameters

1. Lambda Expression with Zero parameter

```
() -> System.out.println("Zero parameter lambda");
```

2. Lambda Expression with Single parameter

```
(p) -> System.out.println("One parameter: " + p);
```

3. Lambda Expression with Multiple parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Example of Lambda Expression with one parameter


```
//functional interface
interface NumericTest{
boolean test(int n);
}
public class LambdaDemo1 {
  public static void main(String args[]) {

  NumericTest isEven= (n) -> (n%2)==0;

  if(isEven.test(10)) System.out.println("10 is
even");
  if(!isEven.test(9)) System.out.println("9 is not
even");
  }
}
```

Passing Lambda Expression as arguments

To pass a lambda expression as a method parameter in java, the type of method parameter that receives must be of **functional interface** type.

```
interface TestInterface {
        boolean test(int a);
    }
class Test {
      // as first argument in check() method
      static boolean check(TestInterface i, int b) {
         return i.test(b);
       }
}
```

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method.

In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}
class Test1 {
    public static void main(String args[])
    {
        int a = 5;
        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;
        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }}
```

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface.

These interfaces are as follows

- **Runnable –>** This interface only contains the run() method.
- **Comparable –>** This interface only contains the compareTo() method.
- **ActionListener –>** This interface only contains the actionPerformed() method.
- **Callable –>** This interface only contains the call() method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

1. Consumer

- The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument.
- The consumer interface has no return value. It returns nothing.
- There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer.
- These variants accept primitive values as arguments.

.

Bi-Consumer

- Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer
- Bi-Consumer is the most exciting variant of the Consumer interface.
- Bi-Consumer interface takes two arguments.
- Both, Consumer and Bi-Consumer have no return value.
- It also returns nothing just like the Consumer interface.
- It is used in iterating through the entries of the map.

Syntax / Prototype of Consumer Functional Interface –

```
Consumer<Integer> consumer = (value) -> System.out.println(value);
```

2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate.

Syntax of Predicate Functional Interface –

```
public interface Predicate<T>
 {
    boolean test(T t);
 }
```

Bi-Predicate

Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.

Example of the implementation of the Predicate functional interface is

```
Predicate predicate = (value) -> value != null;
```

3. Function

- A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing.
- There are many versions of Function interfaces because a primitive type can't imply a general type argument, so we need these versions of function interfaces.

Bi-Function

The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

syntax of Bi-Function

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{
   R apply(T t, U u);
 }
```

T and U are the inputs, and there is only one output which is R.

Unary Operator and Binary Operator

- There are also two other functional interfaces which are named Unary Operator and Binary Operator.
- They both extend the Function and Bi-Function, respectively.
- In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

The prototype of the Unary Operator and Binary Operator is mentioned below

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, U>
{
    ……...
}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, U, R>
{
    ……...
}
```

4. Supplier

- The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output.
- This type of functional interface is generally used in the lazy generation of values.
- Supplier functional interfaces are also used for defining the logic for the generation of any sequence.

For example –

The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.

// lambda expression to create object

```
Predicate<String> p = (s) -> s.startsWith("G");
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 22

Lecture 22

- Method References
- Stream API
- Default Methods

Method References

- Java provides a new feature called method reference in Java 8.
- Method reference is used to refer method of functional interface.
- It is compact and easy form of lambda expression.
- Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

There are following types of method references in java:

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.

1) Reference to a Static Method

You can refer to static method defined in the class.

Syntax

ContainingClass::staticMethodName

Defined a functional interface and referring a static method

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

Using predefined functional interface Runnable to refer static method

```
public class MethodReference2 {
    public static void ThreadStatus(){
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        Thread t2=new Thread(MethodReference2::ThreadStatus);
        t2.start();
    }
}
```

We can also use predefined functional interface to refer methods

```
import java.util.function.BiFunction;
class Arithmetic{
public static int add(int a, int b){
return a+b;
} }
public class MethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}
```

We can also overload static methods by referring methods

```
import java.util.function.BiFunction;
class Arithmetic{
public static int add(int a, int b){
return a+b;
}
public static float add(int a, float b){
return a+b;
}
public static float add(float a, float b){
return a+b;
} }
```

```
public class MethodReference4 {
public static void main(String[] args)
{
BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
int result1 = adder1.apply(10, 20);
float result2 = adder2.apply(10, 20.0f);
float result3 = adder3.apply(10.0f, 20.0f);
System.out.println(result1);
System.out.println(result2);
System.out.println(result3);
} }
```

2) Reference to an Instance Method

like static methods, we can also refer instance methods.

Syntax

containingObject::instanceMethodName

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
}
```

```
class MyInstanceMethodReference
{
   public static void main(String[] args) {
      InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
      // Referring non-static method using reference
         Sayable sayable = methodReference::saySomething;
      // Calling interface method
         sayable.say();
         // Referring non-static method using anonymous object
         Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
         // Calling interface method
         sayable2.say();
   }
}
```

Here we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

```
public class InstanceMethodReference2
{
   public void printnMsg(){
      System.out.println("Hello, this is instance method");
   }
   public static void main(String[] args) {
    Thread t2=new Thread(new InstanceMethodReference2()::print
nMsg);
      t2.start();
   }
}
```

BiFunction interface. It is a predefined interface and contains a functional method apply().

```
import java.util.function.BiFunction;
class Arithmetic{
public int add(int a, int b){
return a+b;
} }
public class InstanceMethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder =
 new Arithmetic()::add;
int result = adder.apply(10, 20);
System.out.println(result);
} }
```

3) Reference to a Constructor

We can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

ClassName::new

```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    } }
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

Java 8 Stream

- Java provides a new additional package in Java 8 called java.util.stream.
- This package consists of classes and interfaces allows functional-style operations on the elements.
- We can use stream by importing java.util.stream package.
- Stream does not store elements.
- It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.

- Stream is functional in nature.
- Operations performed on a stream does not modify it's source.

For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

- The elements of a stream are only visited once during the life of a stream.
- Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.

Filtering Collection by using Stream

```
import java.util.*;
import java.util.stream.Collectors;
class Product{
   int id;
   String name;
   float price;
   public Product(int id, String name, float price)
 {
       this.id = id;
       this.name = name;
       this.price = price;
   }
}
```

```
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList2 =productsList.stream()
                                .filter(p -> p.price > 30000)// filtering data
                                .map(p->p.price)       // fetching price
                                .collect(Collectors.toList()); // collecting as list

         System.out.println(productPriceList2);
    }
}
```

Java Stream Iterating Example

```
import java.util.stream.*;
public class JavaStreamExample {
    public static void main(String[] args){
        Stream.iterate(1, element->element+1)
        .filter(element->element%5==0)
        .limit(5)
        .forEach(System.out::println);
    }
}
```

Filtering and Iterating Collection

```
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // This is more compact approach for filtering data
        productsList.stream()
                    .filter(product -> product.price == 30000)
                    .forEach(product -> System.out.println(product.name));
    }
}
```

Java Stream Example: count() Method

```
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // count number of products based on the filter
        long count = productsList.stream()
                .filter(product->product.price<30000)
                .count();
        System.out.println(count);
    }
}
```

Java Default Methods

- Java provides a facility to create default methods inside the interface.
- Methods which are defined inside the interface and tagged with default are known as default methods.
- These methods are non-abstract methods.

Java Default Method Example

```
1. interface Sayable{
2.     // Default method
3.     default void say(){
4.         System.out.println("Hello, this is default method");
5.     }
6.     // Abstract method
7.     void sayMore(String msg);
8. }
9. public class DefaultMethods implements Sayable{
10.    public void sayMore(String msg){
11. // implementing abstract method
12.        System.out.println(msg);
13.    }
```

```
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){
        // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
       dm.sayMore("Work is worship");
       // calling abstract method


    }
}
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 23

Lecture 23

- Static Method
- Base64 Encode and Decode

Static method in Interface in Java

- Static Methods in Interface are those methods, which are defined in the interface with the keyword static.
- Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

- Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes.
- To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

```
interface NewInterface {
   // static method
    static void hello()
    {
        System.out.println("Hello, New Static Method Here");
    }
   // Public and abstract method of Interface
    void overrideMethod(String str);
}
```

```
// Implementation Class
public class InterfaceDemo implements NewInterface {
    public static void main(String[] args)
    {
        InterfaceDemo interfaceDemo = new InterfaceDemo();
        // Calling the static method of interface
        NewInterface.hello();
        // Calling the abstract method of interface
        interfaceDemo.overrideMethod("Hello, Override Method here");
    }
   // Implementing interface method
     @Override
    public void overrideMethod(String str)
    {
        System.out.println(str);
    }
}
```

To demonstrate Scope of Static method

- The scope of the static method definition is within the interface only.
- If same name method is implemented in the implementation class then that method becomes a static member of that respective class.

Program to Demonstrate scope of static method in Interface.

```
interface PrintDemo {
   // Static Method
    static void hello()
    {
        System.out.println("Called from Interface PrintDemo");
    }
}
```

```
public class InterfaceDemo implements PrintDemo {
    public static void main(String[] args)
   {
      // Call Interface method as Interface
      // name is preceding with method
      PrintDemo.hello();
      // Call Class static method
      hello();
   }
   // Class Static method is defined
   static void hello()
   {
      System.out.println("Called from Class");
   }
}
```

Java Base64 Encode and Decode

- Java provides a class Base64 to deal with encryption.
- We can encrypt and decrypt data by using provided methods.
- We need to import java.util.Base64 in source file to use its methods.
- It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations.
- The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

Nested Classes of Base64

Class	Description
Base64.Decoder	This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
Base64.Encoder	This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Base64 Methods

Methods	Description
public static Base64.Decoder getDecoder()	It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
public static Base64.Encoder getEncoder()	It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
public static Base64.Decoder getUrlDecoder()	It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
public static Base64.Decoder getMimeDecoder()	It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.

Base64.Decoder Methods

Methods	Description
public byte[] decode(byte[] src)	It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into a newly-allocated output byte array. The returned byte array is of the length of the resulting bytes.
public byte[] decode(String src)	It decodes a Base64 encoded String into a newly-allocated byte array using the Base64 encoding scheme.

Base64.Encoder Methods

Methods	Description
public byte[] encode(byte[] src)	It encodes all bytes from the specified byte array into a newly-allocated byte array using the Base64 encoding scheme. The returned byte array is of the length of the resulting bytes.
public int encode(byte[] src, byte[] dst)	It encodes all bytes from the specified byte array using the Base64 encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0.

Basic Encoding and Decoding

```
import java.util.Base64;
public class Base64BasicEncryptionExample {
    public static void main(String[] args) {
        // Getting encoder
        Base64.Encoder encoder = Base64.getEncoder();
        // Creating byte array
        byte byteArr[] = {1,2};
        // encoding byte array
        byte byteArr2[] = encoder.encode(byteArr);
        System.out.println("Encoded byte array: "+byteArr2);
        byte byteArr3[] = new byte[5];
        int x = encoder.encode(byteArr,byteArr3);
        System.out.println("Encoded byte array written to another array: "+byteArr3);
        System.out.println("Number of bytes written: "+x);
     }}
```

Output

Encoded byte array: [B@15db9742

Encoded byte array written to another array: [B@6d06d69c

Number of bytes written: 4

Basic Encoding and Decoding

```
import java.util.Base64;
public class Base64BasicEncryptionExample1 {
    public static void main(String[] args) {
        // Getting encoder
        Base64.Encoder encoder = Base64.getEncoder();
        // Encoding string
        String str = encoder.encodeToString("ABES Engineering College".getBytes());
        System.out.println("Encoded string: "+str);
        // Getting decoder
        Base64.Decoder decoder = Base64.getDecoder();
        // Decoding string
        String dStr = new String(decoder.decode(str));
        System.out.println("Decoded string: "+dStr);
    }}
```

Output

Encoded string: QUJFUyBFbmdpbmVlcmluZyBDb2xsZWdl

Decoded string: ABES Engineering College

URL Encoding and Decoding

```
import java.util.Base64;
public class Base64URLEncryptionExample {
    public static void main(String[] args) {
        // Getting encoder
        Base64.Encoder encoder = Base64.getUrlEncoder();
        // Encoding URL
        String eStr =
encoder.encodeToString("https://abes.ac.in/".getBytes());
        System.out.println("Encoded URL: "+eStr);
        // Getting decoder
        Base64.Decoder decoder = Base64.getUrlDecoder();
        // Decoding URl
        String dStr = new String(decoder.decode(eStr));
        System.out.println("Decoded URL: "+dStr);
    }
}
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 24

Lecture 24

- ForEach Method
- Try-with resources
- Type Annotations, Repeating Annotations

forEach loop

- Java provides a new method forEach() to iterate the elements.
- It is defined in Iterable and Stream interface.
- It is a default method defined in the Iterable interface.
- Collection classes which extends Iterable interface can use forEach loop to iterate elements.

forEach() Signature in Iterable Interface

```
default void forEach(Consumer<super T>action)
```

Java 8 forEach() example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
   public static void main(String[] args) {
      List<String> gamesList = new ArrayList<String>();
      gamesList.add("Football");
      gamesList.add("Cricket");
      gamesList.add("Chess");
      gamesList.add("Hocky");
      gamesList.forEach(System.out::println);
   } }
```

Java Stream forEachOrdered() Method Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachOrderedExample {
   public static void main(String[] args) {
      List<String> gamesList = new ArrayList<String>();
      gamesList.add("Football");
      gamesList.add("Cricket");
      gamesList.add("Chess");
      gamesList.add("Hocky");
  System.out.println("------Iterating by passing lambda expression--------------");
      gamesList.stream().forEachOrdered(games -> System.out.println(games));
   System.out.println("------Iterating by passing method reference--------------");
      gamesList.stream().forEachOrdered(System.out::println);
   } }
```

Try-with-resources

- Try-with-resources statement is a try statement that declares one or more resources in it.
- A resource is an object that must be closed once your program is done using it.
- For example, a File resource or a Socket connection resource.
- The try-with-resources statement ensures that each resource is closed at the end of the statement execution.
- If we don't close the resources, it may constitute a resource leak and also the program could exhaust the resources available to it.

- We can pass any object as a resource that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable.
- We don’t need to add an extra finally block for just passing the closing statements of the resources.
- The resources will be closed as soon as the try-catch block is executed.

Java Type Annotations

- Java 8 has included two new features repeating and type annotations in its prior annotations topic.
- In early Java versions, you can apply annotations only to declarations.
- After releasing of Java SE 8 , annotations can be applied to any type use.
- It means that annotations can be used anywhere.

For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

@NonNull String str;

Following are the examples of type annotations:

1. @NonNull List<String>
2. List<@NonNull String> str
3. Arrays<@NonNegative Integer> sort
4. @Encrypted File file
5. @Open Connection connection
6. void divideInteger(int a, int b) throws @ZeroDivisor ArithmeticException

Repeating Annotations

- Java allows you to repeating annotations in your source code.
- It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.
- For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler

In order for the compiler to do this, two declarations are required in your code.

- Declare a repeatable annotation type
- Declare the containing annotation type

1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation.

In the following example, we have defined a custom @Game repeatable annotation type.

```
@Repeatable(Games.class)
@interfaceGame{
   String name();
   String day();
}
```

2) Declare the containing annotation type

Containing annotation type must have a value element with an array type.

The component type of the array type must be the repeatable annotation type.

In the following example, we are declaring Games containing annotation type:

```
@interfaceGames{
   Game[] value();
}
```

Repeating Annotations Example

```
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
// Declaring repeatable annotation type
@Repeatable(Games.class)
@Retention(RetentionPolicy.RUNTIME)
@interface Game {
    String name();
    String day();
}
// Declaring container for repeatable annotation type
@Retention(RetentionPolicy.RUNTIME)
@interface Games {
    Game[] value();
}
```

```
// Repeating annotation
@Game(name = "Cricket",  day = "Sunday")
@Game(name = "Hockey",   day = "Friday")
@Game(name = "Football", day = "Saturday")
public class RepeatingAnnotationsExample {
    public static void main(String[] args) {
        // Getting annotation by type into an array
       Game[] games =
RepeatingAnnotationsExample.class.getAnnotationsByType(Game.class);
        for (Game game : games) {    // Iterating values
            System.out.println(game.name() + " on " +
game.day());
        }
    }
}
```

Output

Cricket on Sunday

Hockey on Friday

Football on Saturday

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 25

Lecture 25

- Java Module System
- Diamond Syntax with 08 Inner Anonymous Class

Java 9 Module System

Java Module System is a major change in Java 9 version.

Java added this feature to collect Java packages and code into a single unit called module.

- In earlier versions of Java, there was no concept of module to create modular Java applications, that why size of application increased and difficult to move around.
- Even JDK itself was too heavy in size, in Java 8, rt.jar file size is around 64MB.
- To deal with situation, Java 9 restructured JDK into set of modules so that we can use only required module for our project.

module system includes various tools and options

- It Includes various options to the Java tools **javac, jlink and java** where we can specify module paths that locates to the location of module.
- Modular JAR file is introduced. This JAR contains module-info.class file in its root folder.
- JMOD format is introduced, which is a packaging format similar to JAR except it can include native code and configuration files.
- The JDK and JRE both are reconstructed to accommodate modules. It improves performance, security and maintainability.
- Java defines a new URI scheme for naming modules, classes and resources.

Java 9 Module

Module is a collection of Java programs or software.

To describe a module, a Java file module-info.java is required.

This file also known as module descriptor and defines the following

- Module name
- What does it export
- What does it require

Module Name

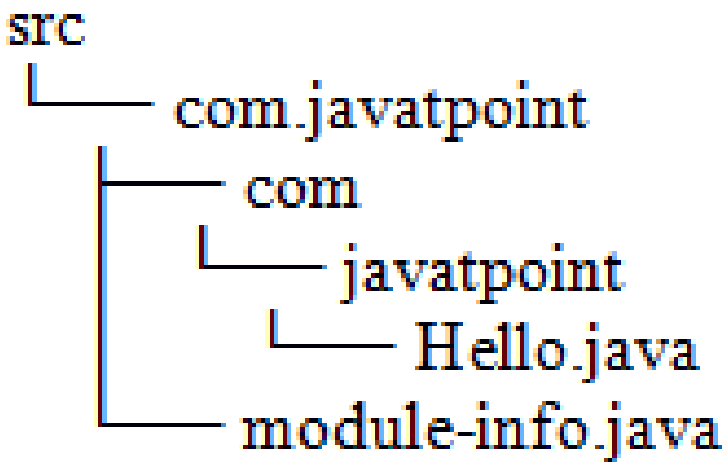
It is a name of module and should follow the reverse-domain-pattern. Like we name packages, e.g. com.javatpoint.

How to create Java module

Creating Java module required the following steps.

- Create a directory structure
- Create a module declarator
- Java source code

Create a Directory Structure



Note: The name of the directory containing a module's sources should be equal to the name of the module, e.g. com.javatpoint.

Java Source Code

```
class Hello{
    public static void main(String[] args){
        System.out.println("Hello from the Java module");
    }
}
```

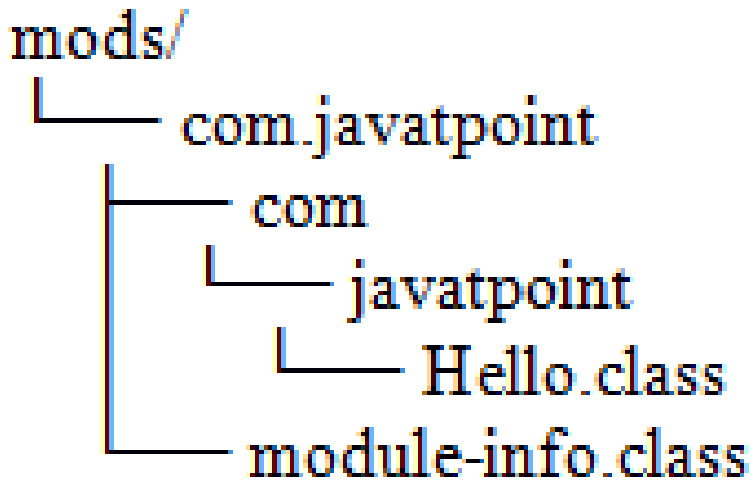
Save this file inside **src/com.javatpoint/com/javatpoint/** with **Hello.java** name.

Compile Java Module

To compile the module use the following command.

javac -d mods --module-source-path src/ --module com.javatpoint

After compiling, it will create a new directory that contains the following structure.



Run Module

java --module-path mods/ --module com.javatpoint/com.javatpoint.Hello

Output:

Hello from the Java module

Diamond Operator

- Diamond operator was introduced in Java 7 as a new feature.
- The main purpose of the diamond operator is to simplify the use of generics when creating an object.
- It avoids unchecked warnings in a program and makes the program more readable.
- The diamond operator could not be used with Anonymous inner classes in JDK 7.
- In JDK 9, it can be used with the anonymous class as well to simplify code and improves readability.

- Before JDK 7, we have to create an object with Generic type on both side of the expression like

List<String> ABES= new ArrayList<String>();

- When Diamond operator was introduced in Java 7, we can create the object without mentioning generic type on right side of expression like:

List<String> ABES = new ArrayList<>();

Problem with Diamond Operator in JDK 7

The problem is it will only work with normal classes. Suppose if we want to use the diamond operator for anonymous inner class then compiler will throw error message.

Java 9 Anonymous Inner Classes

```
abstract class ABCD<T>{
   abstract T show(T a, T b);
}
public class TypeInferExample {
   public static void main(String[] args) {
      ABCD<String> a = new ABCD<>()
     {
         String show(String a, String b)
         {
            return a+b;
         }
      };
      String result = a.show("Java","9");
      System.out.println(result);
   } }
```

Output:

Java9

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 26

Lecture 26

- Local Variable Type Inference
- Switch Expressions
- Yield Keyword

Local Variable Type Inference

- Local variable type inference is a feature in Java 10 that allows the developer to skip the type declaration associated with local variables (those defined inside method definitions, initialization blocks, for-loops, and other blocks like if-else), and the type is inferred by the JDK.
- It will, then, be the job of the compiler to figure out the datatype of the variable.

// declaration using LVTI

```
// Java code for local variable
import java.util.ArrayList;
import java.util.List;
class A {
    public static void main(String ap[])
    {
        var data = new ArrayList<>();
    }
}
```

Cases where you can declare variables using LVTI

```
// block using LVTI in Java 10
class A {
    static
    {
        var x = "Hi there";
        System.out.println(x)'
    }
    public static void main(String[] ax)
    {
    }
}
```

As iteration variable in enhanced for-loop

```
public class MyMain {
public static void main(String[] args) {
int[] arr ={ 1, 2, 3 };
for (var x : arr)
System.out.println(x);
}
}
```

As looping index in for-loop

```
public class MyMain {
public static void main(String[] args)
{
int[] arr = { 1, 2, 3 };
for (var x = 0; x < 3; x++)
System.out.println(arr[x]);
}
}
```

As a return value from another method

```
public class MyMain {
int ret()
{
return 1;
}
public static void main(String a[])
{
var x = new MyMain().ret();
System.out.println(x);
}
}
```

As a return value in a method

```
public class MyMain {
int ret()
{
var x = 1;
return x;
}
public static void main(String a[])
{
System.out.println(new MyMain().ret());
}
}
```

There are cases where declaration of local variables using the keyword ‘var’ produces an error.

1. **Not permitted in class fields**

```
class A {
    var x;
}
```

2. **Not permitted for uninitialized local variables**
3. **Not allowed as parameter for any methods**
4. **Not permitted in method return type.**

```
public var show()
```

5. **Not permitted with variable initialized with ‘NULL’**

Switch Expressions

- Until Java 7 only integers could be used in switch case and this had been the standard for a long time.
- In Java 8 strings & enum were introduced in case values and switch statements started to evolve.

```
public class MyMain {
public static void main(String a[])
{
String day = "Tuesday";
switch (day) {
case "Monday":
System.out.println("Week day");
break;
case "Tuesday":
.
.
break;
case "Friday":
System.out.println("Week day");
break;
case "Saturday":
System.out.println("Weekend");
break;
case "Sunday":
System.out.println("Weekend");
break;
default:
System.out.println("Unknown");
}}}
```

```
public class MyMain {
public static void main(String a[])
{
enum DAYS {
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
DAYS days = DAYS.MONDAY;
switch (days) {
case MONDAY:
System.out.println("Weekdays");
..
break;
case FRIDAY:
System.out.println("Weekdays");
break;
case SATURDAY:
System.out.println("Weekends");
break;
case SUNDAY:
System.out.println("Weekends");
break;
default:
System.out.println("Unknown");
}}}
```

Java 12 : Switch Statement

- Java 12 further enhanced the switch statement and introduced switch expressions as a *preview* feature.

It introduced a flurry of new features:

- We can return values from a switch block and hence switch statements became ***switch expressions***
- We can have multiple values in a case label
- We can return value from a switch expression through the arrow operator or through the “break” keyword

switch expressions Example

```
public class MyMain {
public static void main(String[] args) {
String day = "Wednesday";
String category = getDayCategory(day);
System.out.println(day + " is a " + category + ".");
}
public static String getDayCategory(String day) {
return switch (day) {
case "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday" -> "Weekday";
case "Saturday", "Sunday" -> "Weekend";
default -> "Unknown";
};
}}
```

Output : Wednesday is a Weekday.

Return value through break keyword

```
return  switch (day) {
    case "Monday":
        break "Weekday";
    case "Tuesday":
        break "Weekday";
     case "Friday":
        break "Weekday";
    case "Saturday":
        break "Weekend";
    case "Sunday":
        break "Weekend";
    default:
        break "Unknown";
};
```

The word break was replaced by “yield” later in Java 13.

```
return  switch (day) {
    case "Monday":
        yield  "Weekday";
    case "Tuesday":
        yield "Weekday";
    case "Sunday":
        yield "Weekend";
    default:
        yield "Unknown";
};
```

Return value through arrow operator :

```
return  switch (day)
{   case "Monday"-> "Week day";
    case "Tuesday"-> "Week day";
    case "Wednesday"->"Week day";
    case "Thursday"->"Week day";
    case "Friday"->"Week day";
    case "Saturday"-> "Weekend";
    case "Sunday"-> "Weekend";
    default->"Unknown";
};
```

Multiple case labels :

```
return  switch (day) {
            case
"Monday","Tuesday","Wednesday","Thursday","Friday"
-> "Week day";
            case "Saturday", "Sunday" -> "Weekend";
            default->"Unknown";
        };
```

Java 17 : Switch Statement / Expression :

Java 17 LTS is the latest long-term support release for the Java SE platform and it was released on September 15, 2021.

Switch expression features

- Pattern matching
- Gaurded pattern
- Null cases

Pattern Matching :

We can pass objects in switch condition and this object can be checked for different types in switch case labels.

```
return switch (obj) {
    case Integer i -> "It is an integer";
    case String s -> "It is a string";
    case Employee s -> "It is a Employee";
     default -> "It is none of the known data types";
};
```

Gaurded Patterns :

```
case Employee emp:
if(emp.getDept().equals("IT")) {
yield "This is IT Employee";
}

return switch (obj) {
case Integer i -> "It is an integer";
case String s -> "It is a string";
case Employee employee &&
employee.getDept().equals("IT") -> "IT Employee";
default -> "It is none of the known data types";
};
```

Null Cases

We can never pass a null value to switch statements prior to Java 17 without a Null pointer exception being thrown.

Java 17 allows you to handle it this way

```
case null -> "It is a null object";
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 26

Lecture 26

- Text Blocks
- Records
- Sealed Classes

Text Blocks in Java 15

- In earlier releases of the JDK, embedding multi-line code snippets required a tangled mess of explicit line terminators, string concatenations, and delimiters.
- Text blocks eliminate most of these obstructions, allowing you to embed code snippets and text sequences more or less as-is.
- A text block is an alternative form of Java string representation that can be used anywhere a traditional double-quoted string literal can be used.

Text blocks begin with a “””” (3 double-quote marks) observed through non-obligatory whitespaces and a newline.

```
/ Using a literal string
String text1 = "Geeks For Geeks";

// Using a text block
String text2 = """"
                Geeks For Geeks"""";
```

Example of text blocks

```
public class MyMain {
public static void main(String[] args) {
String text1="""
                hii
                      how are you
Welcome to ABES Engineering College""";
System.out.println(text1);
}
}
```

The object created from text blocks is java.lang.String with the same properties as a regular string enclosed in double quotes.

This includes the presentation of objects and the interning.

Example of Text Blocks

```
public class MyMain {
public static void main(String[] args) {
// Using a literal string
String text1 = "ABES Engineering COllege";
// Using a text block
String text2 = """
ABES Engineering COllege""";
// Both text1 and text2 are strings of equal
value
System.out.println(text1.equals(text2)); // true
System.out.println(text1==text2);
}
}
```

Records

Records are a better choice than classes in situations where you are primarily storing data and not defining any behavior.

Why Records are good for storing data

- With a Record, you can define the data fields in one line of code, instead of having to define a constructor and getter/setter methods for each field in a class. This makes your code shorter, easier to read, and less prone to errors.
- Records have a built-in equals() and hashCode() method, which makes it easy to compare two instances of a Record based on their values

Data Transfer Objects (DTOs)

Records are a good fit for DTOs, which are used to transfer data between different parts of an application.

With records, you can define DTOs with just a few lines of code, reducing the amount of boilerplate code you need to write.

```
public record PersonDTO(String firstName, String lastName, int age) {}
```

Immutable objects

- Records are immutable by default, making them a good choice for classes that should not be modified after instantiation.
- With records, you don’t need to write any code to make the class immutable — it’s done for you automatically.

```
public record Temperature(double value, String unit) {}
```

Person Record

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

Sealed Class in Java

- A sealed class is a technique that limits the number of classes that can inherit the given class.
- This means that only the classes designated by the programmer can inherit from that particular class, thereby restricting access to it.
- when a class is declared sealed, the programmer must specify the list of classes that can inherit it.
- The concept of sealed classes in Java was introduced in Java 15.

Steps to Create a Sealed Class

- Define the class that you want to make a seal.
- Add the “sealed” keyword to the class and specify which classes are permitted to inherit it by using the “permits” keyword.

Example of Sealed Class

```
sealed class Human permits Manish, Vartika, Anjali
{
   public void printName()
   {
      System.out.println("Default");
   }
}
```

```
non-sealed class Manish extends Human
{
public void printName()
   {
      System.out.println("Manish Sharma");
   }
}
sealed class Vartika extends Human
{
   public void printName()
   {
      System.out.println("Vartika Dadheech");
   }
}
```

```
final class Anjali extends Human
{
   public void printName()
   {
      System.out.println("Anjali Sharma");
   }
}
```

Child classes of a sealed class must be sealed, non-sealed, or final.