Unit- 3 Java New Features:

Java Functional Interfaces

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example 1

```
@FunctionalInterface
interface sayable
{
   void say(String msg);
}
public class FunctionalInterfaceExample implements sayable
{
   public void say(String msg)
{
      System.out.println(msg);
   }
   public static void main(String[] args)
{
      FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
      fie.say("Hello there");
   }
}
```

Test its Now

Output:

```
Hello there
```


Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation @*FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

```
1. () -> {
2. //Body of no parameter lambda
3. }
```

One Parameter Syntax

```
1. (p1) -> {
2. //Body of single parameter lambda
3. }
```

Two Parameter Syntax

```
1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }
```

Let's see a scenario where we are not implementing Java lambda expression. Here, we are implementing an interface without using lambda expression.

Without Lambda Expression

```
interface Drawable
{
   public void draw();
}
public class LambdaExpressionExample
 {
   public static void main(String[] args)
{
      int width=10;

      //without lambda, Drawable implementation using anonymous class
       Drawable d=new Drawable()
{
         public void draw()
{
System.out.println("Drawing "+width);
```

```
    }
        };
        d.draw();
    }
}
```

Test it Now

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Test it Now

Output:

```
Drawing 10
```

Unit- 3 Java New Features:

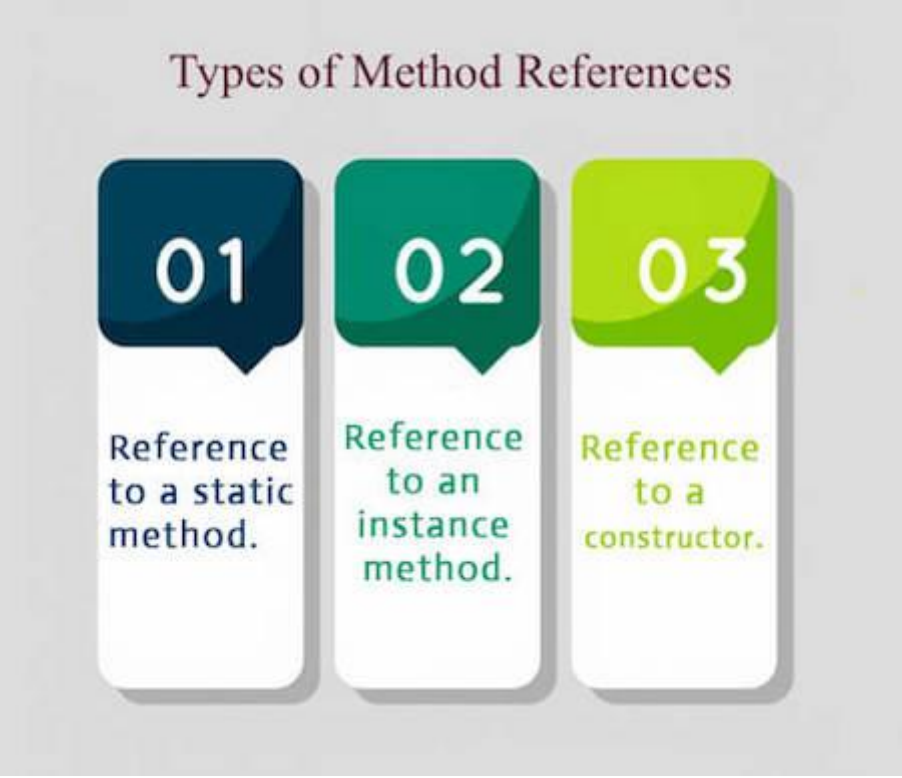
Java Method References

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. In this tutorial, we are explaining method reference concept in detail.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
1. interface Sayable{
2.     void say();
3. }
4. public class MethodReference {
5.     public static void saySomething(){
6.         System.out.println("Hello, this is static method.");
7.     }
8.     public static void main(String[] args) {
9.         // Referring static method
10.        Sayable sayable = MethodReference::saySomething;
11.        // Calling interface method
12.        sayable.say();
13.    }
14. }
```

Test it Now

Output:

```
Hello, this is static method.
```

Java 8 Stream

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

Stream provides following features:

- 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.
- Stream is lazy and evaluates code only when required.
- 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.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

Java Stream Interface Methods

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.
boolean anyMatch(Predicate<? super T> predicate)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

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

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

```
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){       // implementing abstract method
11.        System.out.println(msg);
12.    }
13.    public static void main(String[] args) {
14.        DefaultMethods dm = new DefaultMethods();
15.        dm.say();   // calling default method
16.        dm.sayMore("Work is worship");  // calling abstract method
17.
18.    }
19.}
```

Output:is default method

Unit- 3 Java New Features:

```
Work is worship
```

Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

```
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.      // static method
9.      static void sayLouder(String msg){
10.         System.out.println(msg);
11.     }
12. }
13. public class DefaultMethods implements Sayable{
14.     public void sayMore(String msg){     // implementing abstract method
15.         System.out.println(msg);
16.     }
17.     public static void main(String[] args) {
18.         DefaultMethods dm = new DefaultMethods();
19.         dm.say();                    // calling default method
20.         dm.sayMore("Work is worship");      // calling abstract method
21.         Sayable.sayLouder("Helloooo...");   // calling static method
22.     }
23. }
```

Output:

```
Hello there
Work is worship
Helloooo...
```

Unit- 3 Java New Features:

Java Base64 Encode and Decode

Java provides a class Base64 to deal with encryption. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level. You can use these methods at the following levels.

Basic Encoding and Decoding

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.

URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 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.

MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Unit- 3 Java New Features:

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.
public static Base64.Encoder getMimeEncoder()	It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)	It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

public static Base64.Encoder getUrlEncoder()	It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding 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.
public int decode(byte[] src, byte[] dst)	It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into the given output byte array, starting at offset 0.
public ByteBuffer decode(ByteBuffer buffer)	It decodes all bytes from the input byte buffer using the Base64 encoding scheme, writing the results into a newly-allocated ByteBuffer.
public InputStream wrap(InputStream is)	It returns an input stream for decoding Base64 encoded byte stream.

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.

Unit- 3 Java New Features:

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.
public String encodeToString(byte[] src)	It encodes the specified byte array into a String using the Base64 encoding scheme.
public ByteBuffer encode(ByteBuffer buffer)	It encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the Base64 encoding scheme. Upon return, the source buffer's position will be updated to its limit; its limit will not have been changed. The returned output buffer's position will be zero and its limit will be the number of resulting encoded bytes.
public OutputStream wrap(OutputStream os)	It wraps an output stream for encoding byte data using the Base64 encoding scheme.
public Base64.Encoder withoutPadding()	It returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data.

Java Base64 Example: Basic Encoding and Decoding

```
1. import java.util.Base64;
2. publicclass Base64BasicEncryptionExample {
3.     publicstaticvoid main(String[] args) {
4.         // Getting encoder
5.         Base64.Encoder encoder = Base64.getEncoder();
6.         // Creating byte array
7.         bytebyteArr[] = {1,2};
8.         // encoding byte array
9.         bytebyteArr2[] = encoder.encode(byteArr);
10.        System.out.println("Encoded byte array: "+byteArr2);
11.        bytebyteArr3[] = newbyte[5];            // Make sure it has enough size to
    store copied bytes
12.        intx = encoder.encode(byteArr,byteArr3);   // Returns number of bytes written
```


Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
13.         System.out.println("Encoded byte array written to another array: "+byteAr
    r3);
14.         System.out.println("Number of bytes written: "+x);
15.
16.         // Encoding string
17.         String str = encoder.encodeToString("JavaTpoint".getBytes());
18.         System.out.println("Encoded string: "+str);
19.         // Getting decoder
20.         Base64.Decoder decoder = Base64.getDecoder();
21.         // Decoding string
22.         String dStr = new String(decoder.decode(str));
23.         System.out.println("Decoded string: "+dStr);
24.     }
25. }
```

Output:

```
Encoded byte array: [B@6bc7c054
Encoded byte array written to another array: [B@232204a1
Number of bytes written: 4
Encoded string: SmF2YVRwb2ludA==
Decoded string: JavaTpoint
```

Java For-each Loop | Enhanced For Loop

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Advantages

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

```
for(data_type variable : array | collection){
//body of for-each loop
}
```

How it works?

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.

For-each loop Example: Traversing the array elements

```
//An example of Java for-each loop
class ForEachExample1{
  public static void main(String args[]){
   //declaring an array
   int arr[]={12,13,14,44};
   //traversing the array with for-each loop
   for(int i:arr){
     System.out.println(i);
   }
 }
}
```


Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

12.

Test it Now

Output:

```
12
12
14
44
```

Let us see another of Java for-each loop where we are going to total the elements.

```
1. class ForEachExample1{
2.   public static void main(String args[]){
3.    int arr[]={12,13,14,44};
4.    int total=0;
5.    for(int i:arr){
6.      total=total+i;
7.    }
8.    System.out.println("Total: "+total);
9.  }
10. }
```

Output:

```
Total: 83
```

The try-with-resources statement

In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The try-with-resources statement ensures that each resource is closed at the end of the statement execution.

You can pass any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable.

Unit- 3 Java New Features:

The following example writes a string into a file. It uses an instance of FileOutputStream to write data into the file. FileOutputStream is a resource that must be closed after the program is finished with it. So, in this example, closing of resource is done by itself try.

Try-with-resources Example 1

```
import java.io.FileOutputStream;
public class TryWithResources {
public static void main(String args[]){
      // Using try-with-resources
try(FileOutputStream fileOutputStream =newFileOutputStream("/java7-new-features/src/abc.txt")){
String msg = "Welcome to javaTpoint!";
byte byteArray[] = msg.getBytes(); //converting string into byte array
fileOutputStream.write(byteArray);
System.out.println("Message written to file successfuly!");
}catch(Exception exception){
      System.out.println(exception);
}
}
}
```

Output:

```
Message written to file successfuly!
```

Output of file

```
Welcome to javaTpoint!
```

Java Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations

There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Understanding Built-In Annotations

Let's understand the built-in annotations first.

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

```
class Animal{
```

Unit- 3 Java New Features:

```
void eatSomething(){System.out.println("eating something");}
}

class Dog extends Animal{
@Override
void eatsomething(){System.out.println("eating foods");}//should be eatSomet
hing
}

class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Test it Now

```
Output:Comple Time Error
```

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2
{
@SuppressWarnings("unchecked")
public static void main(String args[])
{
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");

for(Object obj:list)
System.out.println(obj);
```

```
}
}
```

Test it Now

```
Now no warning at compile time.
```

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A
{
void m()
{
System.out.println("hello m");
}

@Deprecated
void n()
{
System.out.println("hello n");
}
}

class TestAnnotation3
{
public static void main(String args[])
{

A a=new A();
a.n();
```

```
    }
    }
```

Test it Now

At Compile Time:

```
Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.
```

At Runtime:

```
hello n
```

Java Custom Annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

1. **@interface** MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

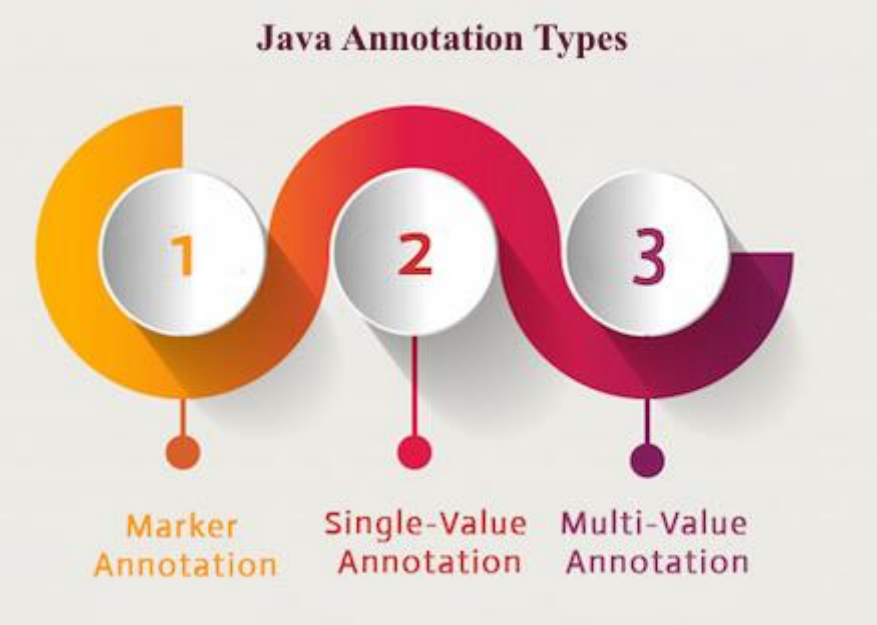
Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation

Unit- 3 Java New Features:

3. Multi-Value Annotation



1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
1. @interface MyAnnotation{}
```

The @Override and @Deprecated are marker annotations.

---

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
1. @interface MyAnnotation{
2. int value();
3. }
```

We can provide the default value also. For example:

```
1. @interface MyAnnotation{
2. int value() default 0;
3. }
```

Unit- 3 Java New Features:

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

```
1. @MyAnnotation(value=10)
```

The value can be anything.

---

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
1. @interface MyAnnotation{
2. int value1();
3. String value2();
4. String value3();
5. }
6. }
```

We can provide the default value also. For example:

```
1. @interface MyAnnotation{
2. int value1() default 1;
3. String value2() default "";
4. String value3() default "xyz";
5. }
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

```
1. @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")
```

Built-in Annotations used in custom annotations in java

- @Target

Unit- 3 Java New Features:

- @Retention
- @Inherited

@Doc ava 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.

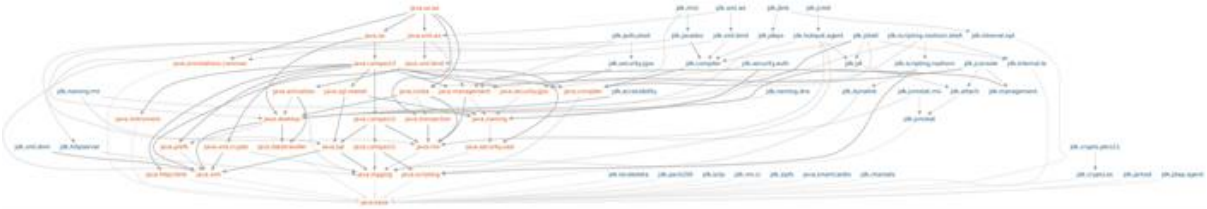
Apart from JDK, Java also allows us to create our own modules so that we can develop module based application.

The module system includes various tools and options that are given below.

- 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.

Unit- 3 Java New Features:

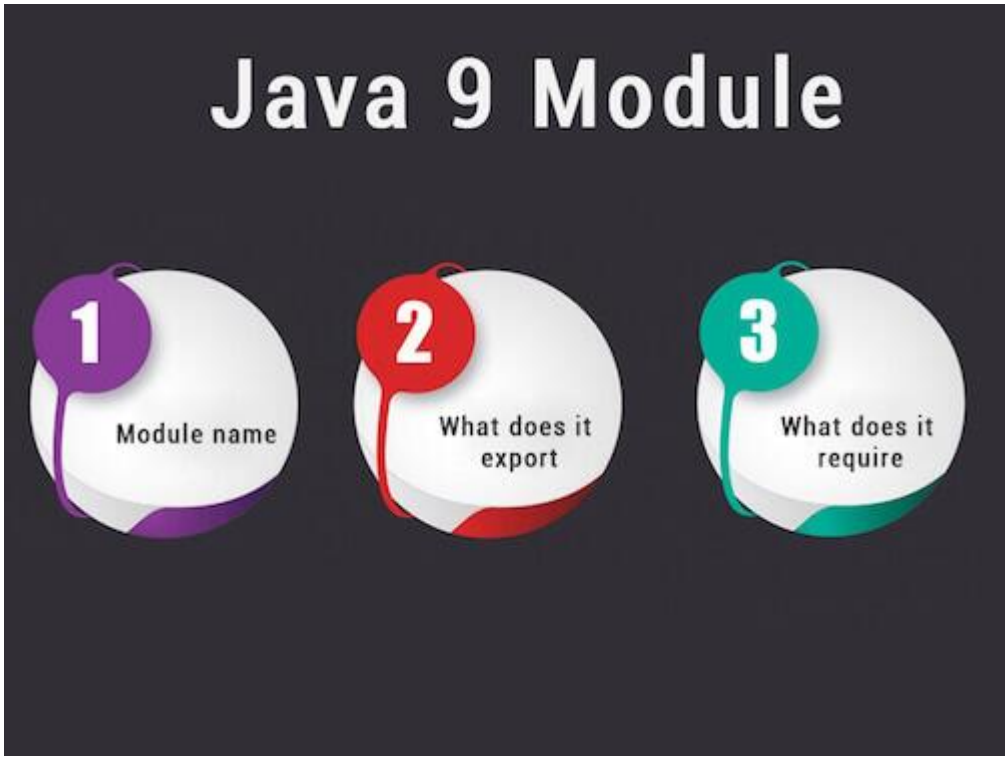
Java 9 Modularized JDK



Java 9 Module

Module is a collection of Java programs or softwares. 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



Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

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

To create module, it is recommended to follow given directory structure, it is same as reverse-domain-pattern, we do to create packages / project-structure in Java.

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

```
src
└── com.javatpoint
    ├── com
    │   └── javatpoint
    │       └── Hello.java
    └── module-info.java
```

Create a file **module-info.java**, inside this file, declare a module by using **module** identifier and provide module name same as the directory name that contains it. In our case, our directory name is com.javatpoint.

```
1. module com.javatpoint{
2.
3. }
```

Leave module body empty, if it does not has any module dependency. Save this file inside **src/com.javatpoint** with **module-info.java** name.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

Java Source Code

Now, create a Java file to compile and execute module. In our example, we have a **Hello.java** file that contains the following code.

```
1. class Hello{
2.     public static void main(String[] args){
3.         System.out.println("Hello from the Java module");
4.     }
5. }
```

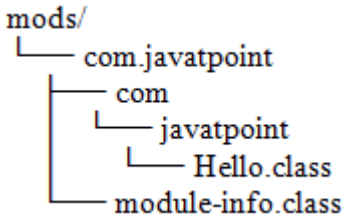
Save this file inside **src/com.javatpoint/com/javatpoint/** with **Hello.java** name.

Compile Java Module

To compile the module use the following command.

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

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



Now, we have a compiled module that can be just run.

Run Module

To run the compiled module, use the following command.

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

Output:

```
Hello from the Java module
```

Well, we have successfully created, compiled and executed Java module.

Unit- 3 Java New Features:

Look inside compiled Module Descriptor

To see the compiled module descriptor use the following command.

1. javap mods/com.javatpoint/module-info.class

This command will show the following code to the console.

```
1. Compiled from "module-info.java"
2. module com.javatpoint {
3.   requires java.base;
4. }
```

See, we created an empty module but it contains a **java.base** module. Why? Because all Java modules are linked to java.base module and it is default module.

- umented

Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

TestAnonymousInner.java

```
1. abstract class Person{
```

```
2.    abstract void eat();
3.  }
4.  class TestAnonymousInner{
5.   public static void main(String args[]){
6.    Person p=new Person(){
7.    void eat(){System.out.println("nice fruits");}
8.    };
9.    p.eat();
10.  }
11. }
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.

Unit- 3 Java New Features:

- The case value can have a *default label* which is optional.

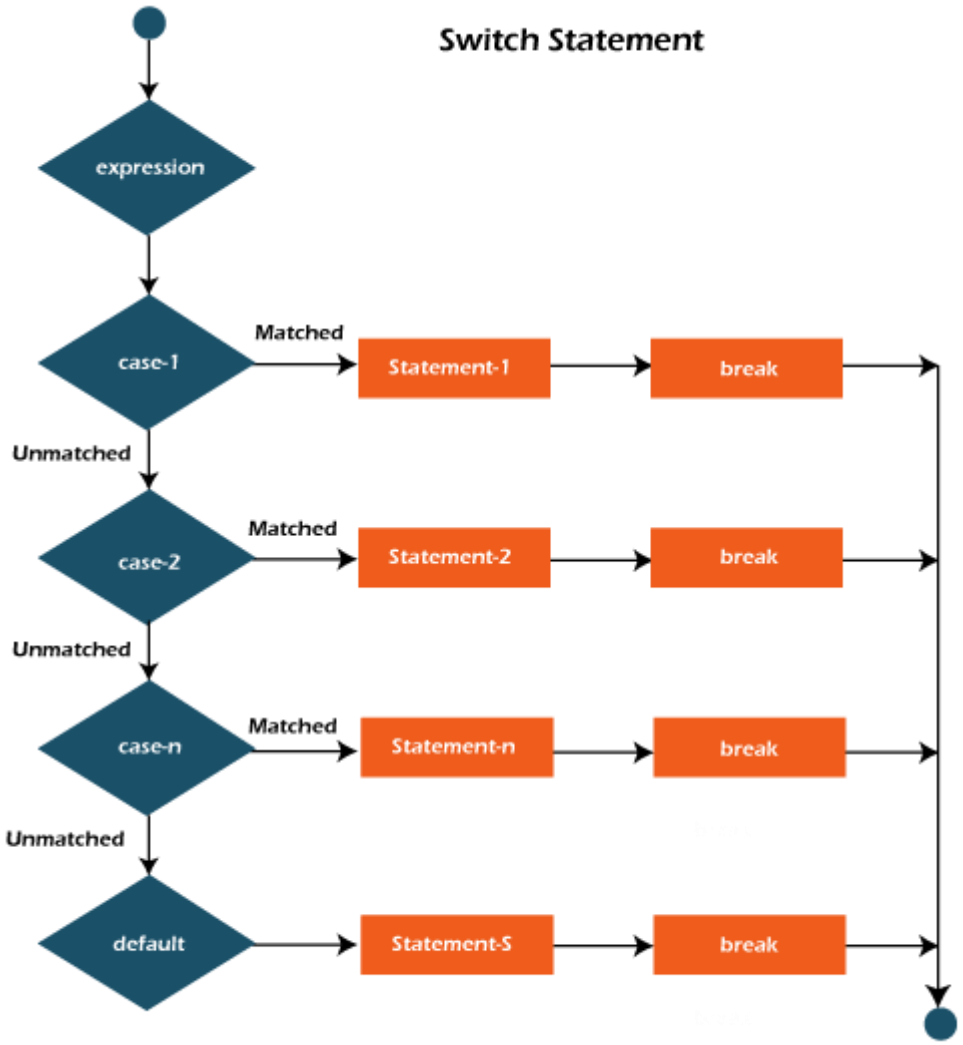
Syntax:

```
1. switch(expression){
2. case value1:
3.  //code to be executed;
4.  break;  //optional
5. case value2:
6.  //code to be executed;
7.  break;  //optional
8. ......
9.
10. default:
11.  code to be executed if all cases are not matched;
12. }
```

Flowchart of Switch Statement

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:



Example:

SwitchExample.java

```
1. public class SwitchExample {
2. public static void main(String[] args) {
3.     //Declaring a variable for switch expression
4.     int number=20;
5.     //Switch expression
6.     switch(number){
7.     //Case statements
8.     case 10: System.out.println("10");
```

Unit- 3 Java New Features:

```
9.      break;
10.    case 20: System.out.println("20");
11.    break;
12.    case 30: System.out.println("30");
13.    break;
14.    //Default case statement
15.    default:System.out.println("Not in 10, 20 or 30");
16.    }
17. }
18. }
```

Test it Now

Output:

```
20
```

Finding Month Example:

SwitchMonthExample.javaHTML

```
1.  //Java Program to demonstrate the example of Switch statement
2.  //where we are printing month name for the given number
3.  public class SwitchMonthExample {
4.  public static void main(String[] args) {
5.      //Specifying month number
6.      int month=7;
7.      String monthString="";
8.      //Switch statement
9.      switch(month){
10.     //case statements within the switch block
11.     case 1: monthString="1 - January";
12.     break;
13.     case 2: monthString="2 - February";
14.     break;
15.     case 3: monthString="3 - March";
16.     break;
```

Unit- 3 Java New Features:

```
17.    case 4: monthString="4 - April";
18.    break;
19.    case 5: monthString="5 - May";
20.    break;
21.    case 6: monthString="6 - June";
22.    break;
23.    case 7: monthString="7 - July";
24.    break;
25.    case 8: monthString="8 - August";
26.    break;
27.    case 9: monthString="9 - September";
28.    break;
29.    case 10: monthString="10 - October";
30.    break;
31.    case 11: monthString="11 - November";
32.    break;
33.    case 12: monthString="12 - December";
34.    break;
35.    default:System.out.println("Invalid Month!");
36.    }
37.    //Printing month of the given number
38.    System.out.println(monthString);
39. }
40. }
```

Test it Now

Output:

```
7 - July
```

Java Thread yield() method

The **yield()** method of thread class causes the currently executing thread object to temporarily pause and allow other threads to execute.

Unit- 3 Java New Features:

Syntax

1. **public static void** yield()

Return

This method does not return any value.

Example

```
1.  public class JavaYieldExp extends Thread
2.  {
3.      public void run()
4.      {
5.          for (int i=0; i<3 ; i++)
6.              System.out.println(Thread.currentThread().getName() + " in control");
7.      }
8.      public static void main(String[]args)
9.      {
10.         JavaYieldExp t1 = new JavaYieldExp();
11.         JavaYieldExp t2 = new JavaYieldExp();
12.         // this will call run() method
13.         t1.start();
14.         t2.start();
15.         for (int i=0; i<3; i++)
16.         {
17.             // Control passes to child thread
18.             t1.yield();
19.             System.out.println(Thread.currentThread().getName() + " in control");
20.         }
21.     }
22. }
```

Test it Now

Output:

```
main in control
main in control
main in control
Thread-0 in control
```


Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

```
Thread-0 in control
Thread-0 in control
Thread-1 in control
Thread-1 in control
Thread-1 in control
```

Ü

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
2.                  ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.

Unit- 3 Java New Features:

7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
1. import java.sql.*;
2. class FetchRecord{
3. public static void main(String args[])throws Exception{
4.
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
   "system","oracle");
7. Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,Resul
   tSet.CONCUR_UPDATABLE);
8. ResultSet rs=stmt.executeQuery("select * from emp765");
9.
10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
13.
14. con.close();
15. }}
```


Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit- 3 Java New Features:

3 Sealed Classes

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

For background information about sealed classes and interfaces, see JEP 409.

One of the primary purposes of inheritance is code reuse: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug) them yourself.

However, what if you want to model the various possibilities that exist in a domain by defining its entities and determining how these entities should relate to each other? For example, you're working on a graphics library. You want to determine how your library should handle common geometric primitives like circles and squares. You've created a `Shape` class that these geometric primitives can extend. However, you're not interested in allowing any arbitrary class to extend `Shape`; you don't want clients of your library declaring any further primitives. By sealing a class, you can specify which classes are permitted to extend it and prevent any other arbitrary class from doing so.

Declaring Sealed Classes

To seal a class, add the `sealed` modifier to its declaration. Then, after any `extends` and `implements` clauses, add the `permits` clause. This clause specifies the classes that may extend the sealed class.
For example, the following declaration of `Shape` specifies three permitted subclasses, `Circle`, `Square`, and `Rectangle`:

Figure 3-1 Shape.java

```
Copy
public sealed class Shape
    permits Circle, Square, Rectangle {
}
```

Define the following three permitted subclasses, `Circle`, `Square`, and `Rectangle`, in the same module or in the same package as the sealed class:

Figure 3-2 Circle.java

Unit- 3 Java New Features:

```
Copy
public final class Circle extends Shape {
   public float radius;
}
```

Figure 3-3 Square.java

Square is a *non-sealed class*. This type of class is explained in Constraints on Permitted Subclasses.

```
Copy
public non-sealed class Square extends Shape {
  public double side;
}
```

Figure 3-4 Rectangle.java

```
Copy
public sealed class Rectangle extends Shape permits FilledRectangle {
   public double length, width;
}
```

Rectangle has a further subclass, FilledRectangle:

Figure 3-5 FilledRectangle.java

```
Copy
public final class FilledRectangle extends Rectangle {
   public int red, green, blue;
}
```

Alternatively, you can define permitted subclasses in the same file as the sealed class. If you do so, then you can omit the permits clause:

```
Copy
package com.example.geometry;

public sealed class Figure
   // The permits clause has been omitted
   // as its permitted classes have been
   // defined in the same file.
{}

final class Circle extends Figure {
   float radius;
}
non-sealed class Square extends Figure {
   float side;
```

Unit- 3 Java New Features:

```
}
sealed class Rectangle extends Figure {
   float length, width;
}
final class FilledRectangle extends Rectangle {
   int red, green, blue;
}
```