Java New Features

Unit-3

Java New Features: Functional Interfaces, Lambda Expression, Method References, Stream API, Default Methods, Static Method, Base64 Encode and Decode, ForEach Method, Try-with-resources, Type Annotations, Repeating Annotations, Java Module System, Diamond Syntax with Inner Anonymous Class, Local Variable Type Inference, Switch Expressions, Yield Keyword, Text Blocks, Records, Sealed Classes

Meaning: To include or show something as a special or important part of something, or to be included as an important part

New Features in Java

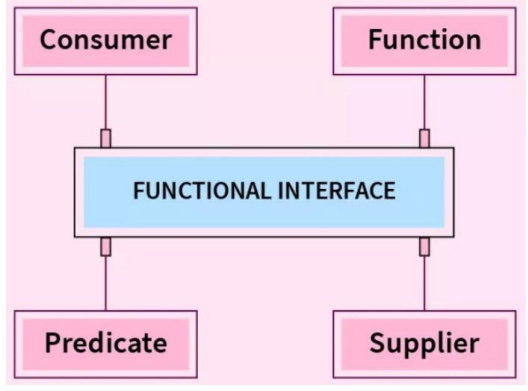
Feature w features that have been added in java. There are major enhancement made in Java5, Java6, Java7 and Java8 like **auto-boxing**, **generics**, **var-args**, **java annotations**, **enum**, **premain method** , **lambda expressions**, **functional interface**, **method references** etc.

Java 8 Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

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.

Java Functional Interfaces



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");
    }
}
```

Output:

```
Hello there
```

Invalid Functional Interface

```
interface sayable{
    void say(String msg);   // abstract method
}
@FunctionalInterface
interface Doable extends sayable{
    // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
    void doIt();
}
```

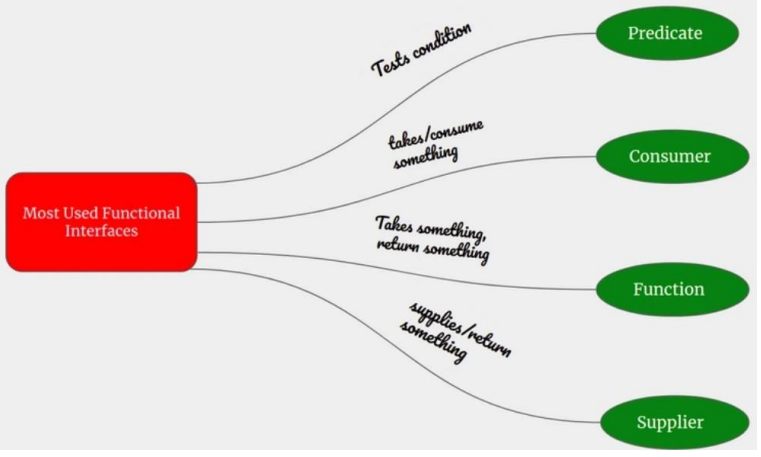
Output:

```
compile-time error
```

Java Predefined-Functional Interfaces

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

You can also define your own custom functional interface. Following is the list of functional interface which are placed in java.util.function package.



Interface	Description
BiConsumer<T,U>	It represents an operation that accepts two input arguments and returns no result.
Consumer<T>	It represents an operation that accepts a single argument and returns no result.
Function<T,R>	It represents a function that accepts one argument and returns a result.
Predicate<T>	It represents a predicate (boolean-valued function) of one argument.
BiFunction<T,U,R>	It represents a function that accepts two arguments and returns a a result.
BinaryOperator<T>	It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands.
BiPredicate<T,U>	It represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	It represents a supplier of boolean-valued results.

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

```
(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

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax

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

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();
    }
}
```

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();
   }
}
```

Output:

```
Drawing 10
```

Java Lambda Expression Example: No Parameter

A lambda expression can have zero or any number of arguments. Let's see the examples:

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {

   Sayable s=()->{
      return "I have nothing to say.";
   };
   System.out.println(s.say());
}
}
```

Output:

```
I have nothing to say.
```

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.

Types of Method References

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

```
ContainingClass::staticMethodName
```

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

```
containingObject::instanceMethodName
```

Example

```
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()::printnMsg);
        t2.start();
    }
}
```

Output:

```
Hello, this is instance method
```

Example

```
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");
    }
}
```

Output:

```
Hello
```

3) Reference to a Constructor

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

Syntax

```
ClassName::new
```

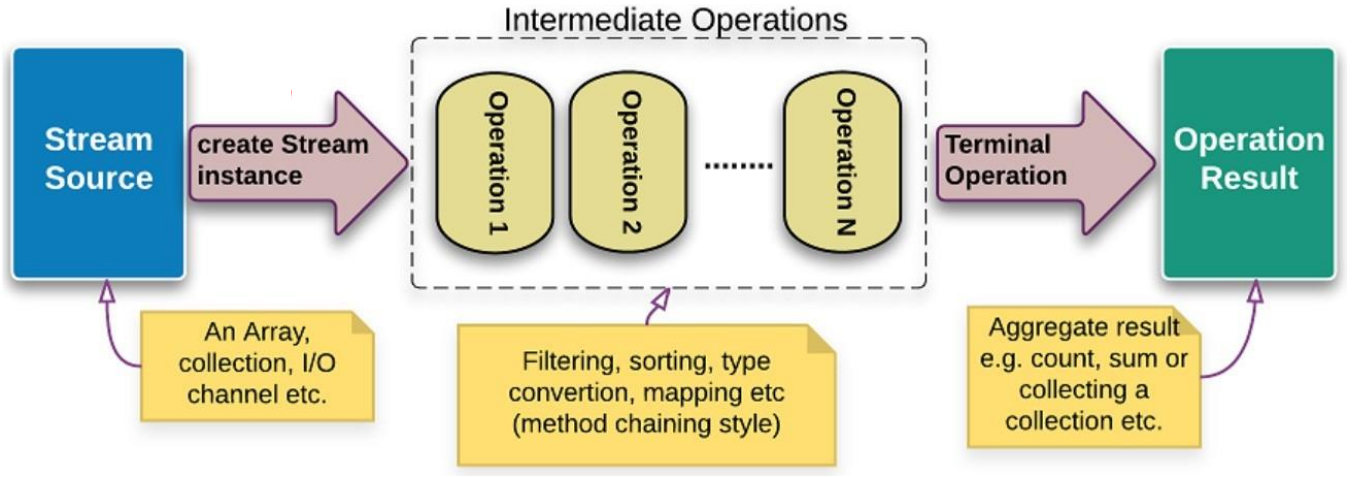
Stream API

Collection: It means **single unit of objects.**

Stream: If we want **to process the object from a collection,** then we should use a **Stream.**

Collection to Stream: `Stream s = collection.stream();`

Package: `java.util.stream`



Stream API is a newly added feature to the Collections API in Java 8. A stream represents a sequence of elements & supports different operations (Filter, Sort, Map, and Collect) from a collection.

In Stream API, operations are classified into two operation:

1. Intermediate Operations:

Transform a stream into another stream.

Examples are: `filter, map, distinct, sorted, limit` etc.

2. Terminal Operations:

It produce a result and terminate the stream

Examples are: `forEach, collect, reduce, count` etc.

Filter:

- It is used to filter the data from stream.
- And create a new stream.
- filter takes predicate as an argument, which returns Boolean types (true/false)
- It is intermediate operation.

Syntax:

```
Stream filteredStream = originalStream.filter(element -> /* predicate */);
```

Map

- Map is used to transform each element of Stream.
- And returns a new Stream
- Map takes function as an argument, the return type based on the types of data.
- It is intermediate operation.

Syntax:

```
Stream mappedStream = originalStream.map(element -> /* transformation function */);
```

Usage of Stream API Methods:

Using stream API, we can obtain the stream by calling the .stream() method on the languages array list and printing it in the following way:

```
Languages.stream().forEach(System.out::println)
```

After getting the stream, we called the forEach() method & pass the action we wanted to take on each Element, then printed the member value on the console using the System.out.println method.

After getting a stream from a collection, we can use that stream to process the collection's elements.

```
languages.stream().sorted().forEach(System.out::println);
```

The above method would help sort the elements in alphabetical order.

stream.sort():

We can sort a stream by calling the sort() method.

stream.collect():

Stream API provides a collect() method for processing on the stream interface. When the collect() method is invoked, filtering and mapping will occur, and the object obtained from those actions will be collected. Let's take the previous example and obtain a new list of languages in uppercase, as shown in the below example:

List<String>

```
case language =
languages.stream().map(item>item.toUpperCase()).collect(collectors.toList()));
```

System.out.println(ucaselanguage);

First, it creates a stream, adds a map to convert the strings to uppercase, and collects all objects in a new list.

stream.min() & max():

The Stream API provides the min() and max() methods to find the min & max values in streams. These methods are used to find min & max values in different streams like streams of chars, strings, dates, etc. We must change the parameter we pass in this method based on the stream type.

stream.count(): In Stream API, the Count method returns the number of elements in the stream after filtering. So, let's take the previous example to get the count of languages starting with 'G.'

```
long count = languages.stream().filter(item->item.getName().startsWith('G')).count();
```

count() method returns a long, which is the count of elements matching the filter criteria.

Advantages of Stream API:

1. Stream conveys elements from a source, such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.

2. It's functional in nature, and an operation on a stream produces a result but doesn't modify its source. For example, when we are filtering a stream obtained from a collection, it has a new Stream without the filtered elements.

3. Consumable: Elements of a stream are visited once during the life of a stream. Like an Iterator, a new stream should be created to revisit the same source elements.

Conclusion:

Here we learned how to use Stream API and Lambda functions in selenium WebDriver code, and it helps us to write code in a functional programming style, which is more readable in nature. Streams are also helpful in working with the list of WebElements, where we can easily collect & filter data with a stream.

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.

```
interface Sayable{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
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

    }
}
```

Output:

```
Hello, this is default method
Work is worship
```

Static Methods

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?

```
interface Sayable{
    // default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
    // static method
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
```

```
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
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

Output:

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

Static method is a static member to the Interface, cant be overridden (as with the class), default method is the default implementation of a method which might be overridden.

Difference between Static Interface Method and Default Interface Method

Sl.No	Static Interface Method	Default Interface Method
1.	It is a static method which belongs to the interface only. We can write implementation of this method in interface itself	It is a method with default keyword and class can override this method
2.	Static method can invoke only on interface class not on class.	It can be invoked on interface as well as class
3.	Interface and implementing class, both can have static method with the same name without overriding each other.	We can override the default method in implementing class
4.	It can be used as a utility method.	It can be used to provide common functionality in all implementing classes

Java Base64 Encode and Decode

Base 64 is an encoding scheme that converts binary data into text format, so that encoded textual data can be easily transported over network, un-corrupted and without any data loss.

For many years, java has provided support for base 64 via a non-public class (therefore non-usable) **java.util.prefs.Base64** an undocumented class **sun.misc.BASE64Encoder**. This class has also very limited information in public domain.

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.

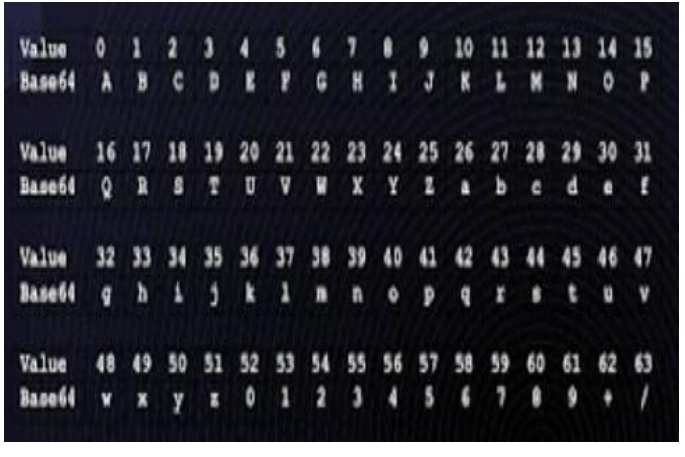
MIME

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.

- **Simple** – Output is mapped to a set of characters lying in A-Za-z0-9+/. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
- **URL** – Output is mapped to set of characters lying in A-Za-z0-9+_. Output is URL and filename safe.
- **MIME** – Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters.

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.



Java Base64 Example: Basic Encoding and Decoding

```
import java.util.Base64;
publicclass Base64BasicEncryptionExample {
   publicstaticvoid main(String[] args) {
      // Getting encoder
      Base64.Encoder encoder = Base64.getEncoder();
      // Creating byte array
      bytebyteArr[] = {1,2};
      // encoding byte array
      bytebyteArr2[] = encoder.encode(byteArr);
      System.out.println("Encoded byte array: "+byteArr2);
      bytebyteArr3[] = newbyte[5];            // Make sure it has enough size to store copied bytes
      intx = encoder.encode(byteArr,byteArr3);   // Returns number of bytes written
      System.out.println("Encoded byte array written to another array: "+byteArr3);
      System.out.println("Number of bytes written: "+x);
      // Encoding string
      String str = encoder.encodeToString("JavaTpoint".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 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 Base64 Example: URL Encoding and Decoding

```
import java.util.Base64;
publicclass Base64BasicEncryptionExample {
   publicstaticvoid main(String[] args) {
      // Getting encoder
      Base64.Encoder encoder = Base64.getUrlEncoder();
      // Encoding URL
      String eStr = encoder.encodeToString("http://www.javatpoint.com/java-tutorial/".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);
   }
}
```

Output:

```
Encoded URL:
Decoded URL:
```

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.

Java Base64 Example: MIME Encoding and Decoding

```
package Base64Encryption;
import java.util.Base64;
publicclass Base64BasicEncryptionExample {
    publicstaticvoid main(String[] args) {
        // Getting MIME encoder
        Base64.Encoder encoder = Base64.getMimeEncoder();
        String message = "Hello, \nYou are informed regarding your inconsistency of work";
        String eStr = encoder.encodeToString(message.getBytes());
        System.out.println("Encoded MIME message: "+eStr);
        // Getting MIME decoder
        Base64.Decoder decoder = Base64.getMimeDecoder();
        // Decoding MIME encoded message
        String dStr = new String(decoder.decode(eStr));
        System.out.println("Decoded message: "+dStr);
    }
}
```

Output:

```
Encoded MIME message: SGVsbG8sIApZb3UgYXJlIGluZm9ybWVkIHJlZ2FyZGluZyB5b3VyIGluY29uc2lzdGVuY3kgb2Yg
d29yaw==
Decoded message: Hello,
You are informed regarding your inconsistency of work
```

ForEach Method

The Java forEach() method is a utility function to iterate over a collection such as (list, set or map) and stream. It is used to perform a given action on each element of the collection.

The forEach() method has been added in following places:

1. **Iterable interface –** This makes Iterable.forEach() method available to all collection classes except Map
2. **Map interface –** This makes forEach() operation available to all map classes.
3. **Stream interface –** This makes forEach() and forEachOrdered() operations available to all types of stream.

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.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

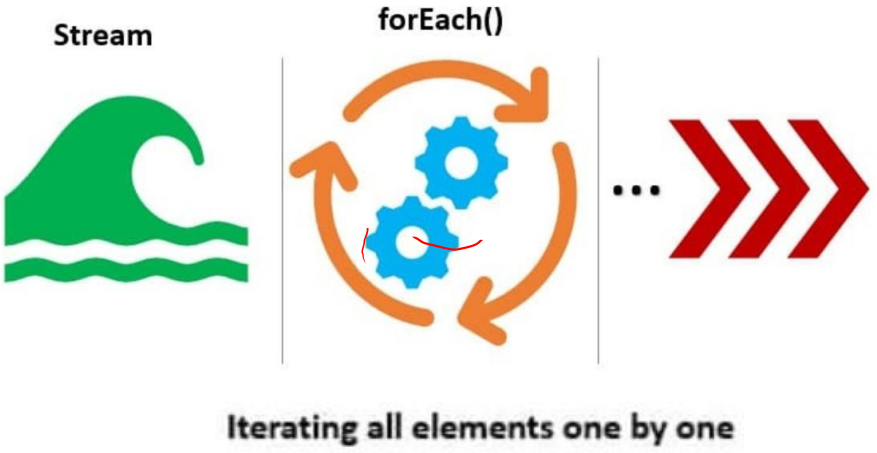
forEach() Signature in Iterable Interface

```
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");
        System.out.println("------------Iterating by passing lambda expression--------------");
        gamesList.forEach(games -> System.out.println(games));

    }
}
```



Output:

```
------------Iterating by passing lambda expression--------------
Football
Cricket
Chess
Hocky
```

try with resources

- Until java 1.6 version it is highly recommended to write finally block to close resources which are open as a part of try block.
- The problems in this approach are: -
  - Programmer is required to close the resources inside finally block. It increases complexity of programming.
  - We have to write finally block compulsory and hence it increases length of the code
- To overcome above problems Java introduced try with resources in Java 1.7 version
- The main advantage of try with resources is whatever resources we open as a part of try block will be closed automatically once control reaches end of try block either normally or abnormally and hence we are not required to close explicitly so that complexity of programming will be reduced
- We are not required to write finally block so that length of the code will be reduced.

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

Java 1.6

```
try
{
        use resource
}
catch ( )
{
        handle exception
}
finally
{
        close resources
}
```

Java 1.7

```
try(resource)
{
        use resource
}
catch ( )
{
        handle exception
}
```

Try-with-resources Example

```
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!
```

Multiple Resources

We can declare multiple resources but these resources should be separated with semicolon

```
try (Resource1 ; resource2 ; resource3)
{
}
```

All resources should be AutoCloseable resources.

A resources said to be auto closeable if and only if corresponding class implement java.lang.AutoCloseable Interface

What is a Java Annotation?

Java Annotation is a kind of a tag that represents the metadata or information attached with class, interface, methods, or fields to show some additional information that Java compiler and JVM can use.

Though Annotations are not a part of the Java code they allow us to add metadata information into our source code. Java introduced Annotations from JDK 5.

There is no direct effect of Annotations on the operation of the code they annotate; they do not affect the execution of the program. Annotations provide supplemental information about a program.

Some points about Annotations are:

- They start with '@'.
- They do not change the action or execution of a compiled program.
- Annotations help to associate metadata or information to the elements of the program like classes, instance variables, interfaces, constructors, methods, etc.
- We cannot consider Annotations as pure comments as they can change the way a compiler treats a program

Example of Java Annotation

```
class Base {
  public void display() {
    System.out.println("Base class display() method");
  }
}
public class Derived extends Base {@Override
  public void display(int x) {
    System.out.println("Derived class display(int) method")
  }
  public static void main(String args[]) {
    Derived obj = new Derived();
    obj.display();
  }
}
```

Output:

error: method does not override or implement a method from a supertype
@Override

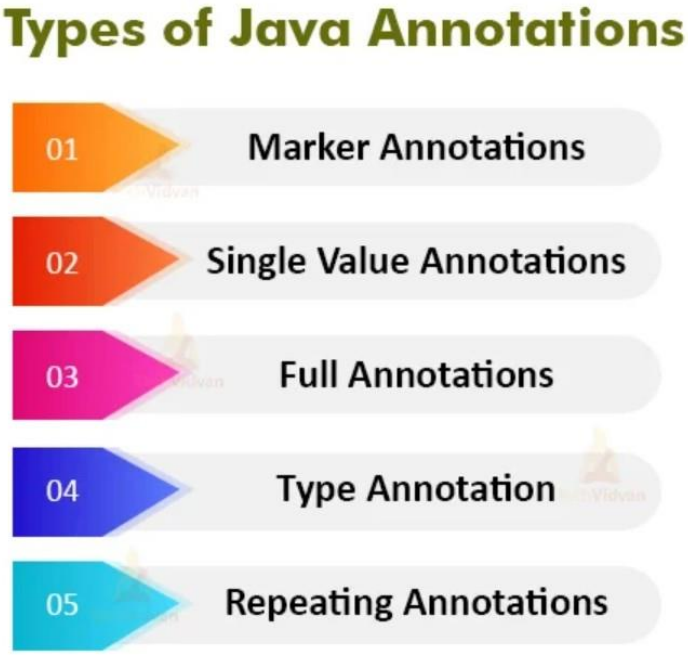
If we remove parameter (int x) from the method or if we remove the @override annotation from the code, then the program compiles fine. The output will be:

Base class display() method

Types of Java Annotations

There are five types of Java Annotations which are:

- Marker Annotations
- Single Value Annotations
- Full Annotations
- Type Annotation
- Repeating Annotation



Type Annotations in Java

The type annotations are applicable to any place where there is a use of a type. For example, if we want to annotate the return type of a method, we can declare these annotations with @Target annotation.

Code to demonstrate a Type Annotation

```
 import java.lang.annotation. * ;@Target(ElementType.TYPE_USE)@interface
TypeAnnoDemo {}

 public class MyClass {
   public static void main(String[] args) {@TypeAnnoDemo String s = "Hello,I am
annotated with a type annotation";
     System.out.println(s);
     myMethod();
   }

   static@TypeAnnoDemo int myMethod() {
     System.out.println("There is a use of annotation with the return type of the
function");
                return 0;
         }
 }
```

Output:

I am annotated with a type annotation

There is a use of annotation with the return type of the function");

Repeating Annotations in java

Repeating Annotations are the annotations that we apply to a single item more than once. The repeating annotations must be annotated with the @Repeatable annotation, which is present in the java.lang.annotation package. The value of this annotation specifies the container type for the repeatable annotation.

There is a container specified as an annotation whose value field is an array of the repeatable annotation type. Hence, to create a repeatable annotation, firstly we need to create the container annotation, and then specify the annotation type as an argument to the @Repeatable annotation.

Code to demonstrate a repeatable annotation

```
 import java.lang. *
;@Retention(RetentionPolicy.RUNTIME)@Repeatable(MyRepeatedAnnos.class)@interface
MyWords {
   String word()
 default "Hello World";
   int value()
 default 0;
 }
```

```java
// Creating a container annotation
@Retention(RetentionPolicy.RUNTIME)@interface MyRepeatedAnnotations {
  MyWords[] value();
}
public class MyClass {

  @MyWords(word = "Data", value = 1)@MyWords(word = "Flair", value = 2)
  public static void myMethod() {
    MyClass obj = new MyClass();

    try {
      Class < ?>c = obj.getClass();

      Method m = c.getMethod("myMethod");

      Annotation a = m.getAnnotation(MyRepeatedAnnotations.class);
      System.out.println(anno);
    }
    catch(NoSuchMethodException e) {
      System.out.println(e);
    }
  }
  public static void main(String[] args) {
    myMethod();
  }
}
```

Output:

@MyRepeatedAnnotations(value={@MyWords(value=1, word="Data"), @Words(value=2, word="Flair")})

Predefined/ Standard Annotations in java

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

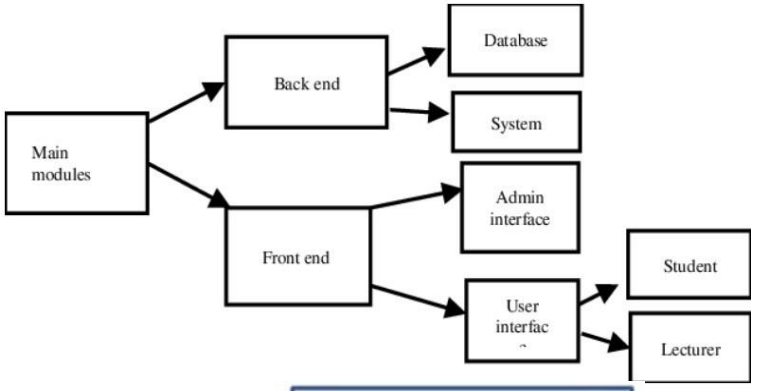
Built-In Java Annotations used in other annotations

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

Java 9 Module System

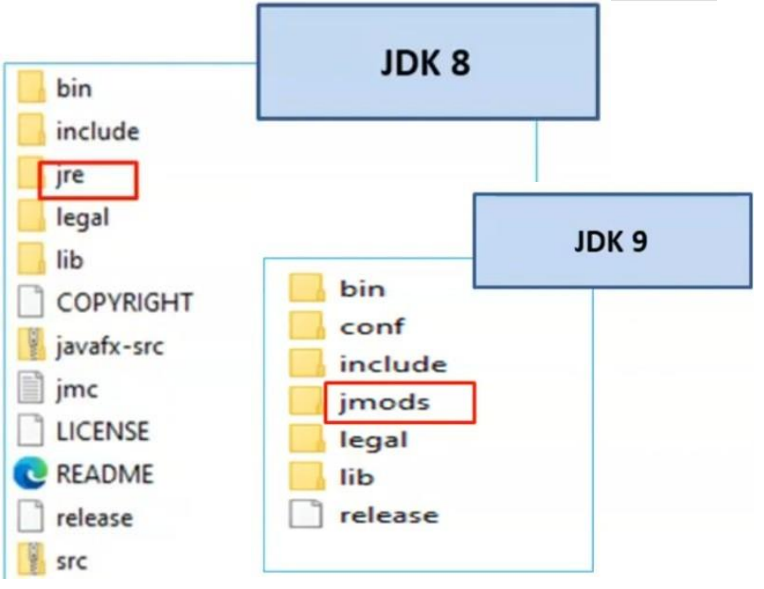
What is module system?

- Java added this feature to collect Java packages and code into a single unit called *module*.
- A Module is a set of related Packages, Types (classes, abstract classes, interfaces etc) with Code & Data and Resources.



Why module system?

- JDK 8 or earlier versions are too heavy in size.
- Jar files like rt.jar etc are too big to use in small devices and applications.



Java 9 restructured JDK into set of modules so that we can use only required module for our project.

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.

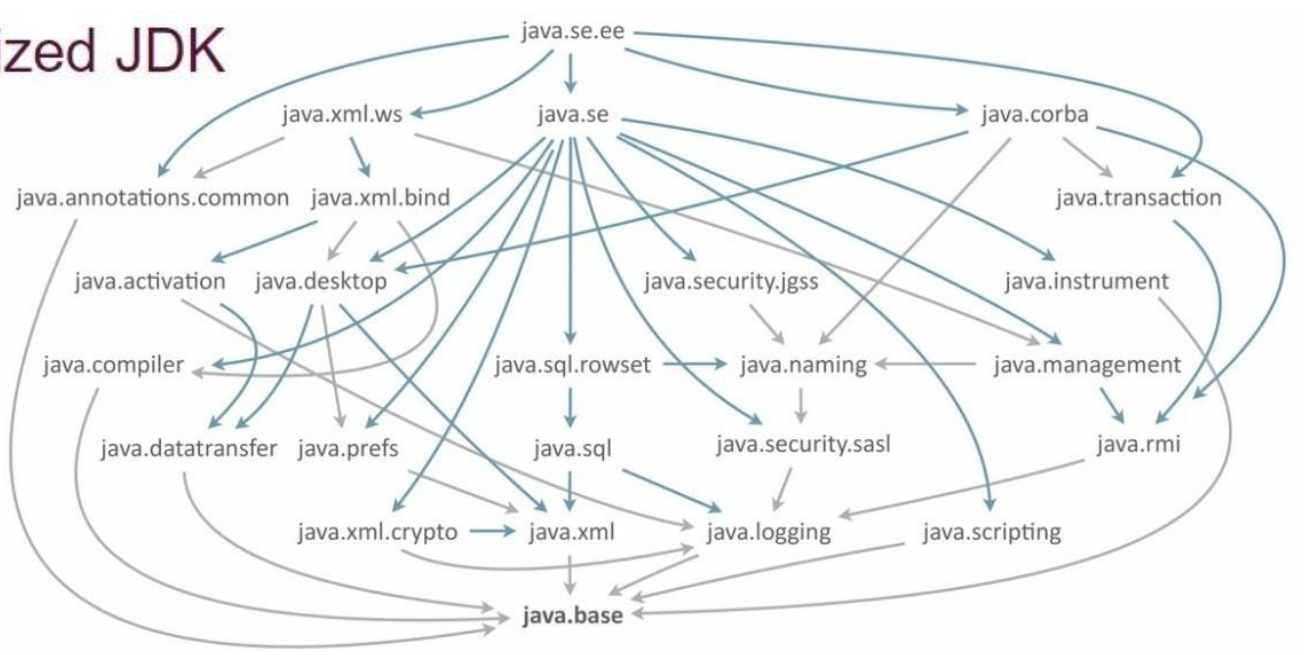
Apart from JDK, Java also allows us to create our own modules so that we can develop module based application.

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.

Java 9 Modularized JDK

This is how JDK now looks like. On the bottom, we have "**java.base**" module that every other module implicitly or explicitly depends on. As you can see, this dependency graph is a DAG which means no circular dependency allowed.



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

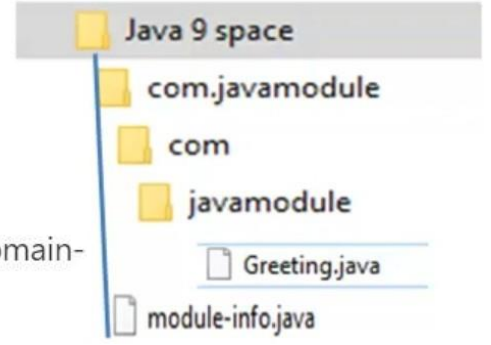
Module Name

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

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.

Create a module Descriptor

module-info.java

```
module com.javamodule{

}
```

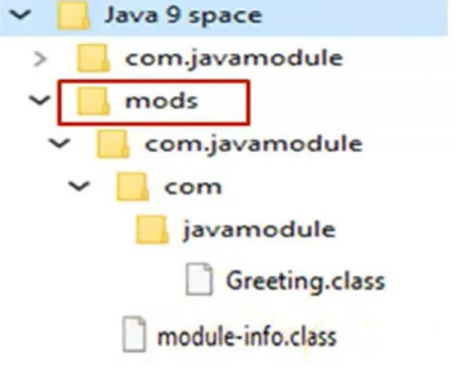
Java source code

Greeting.java

```
package com.javamodule;
public class Greeting
{
    public static void main(String[]ar)
    {
        System.out.println("Hello java world!!");
    }
}
```

Compile java module

```
P:\Java 9 space>javac -d mods --module-source-path . --module com.javamodule
```



Run java module

```
P:\Java 9 space>java --module-path mods/ --module com.javamodule/com.javamodule.Greeting
Hello java world!!
```

Diamond Syntax with Inner Anonymous Class

Inner Anonymous Class , It 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 overriding methods of a class or interface, without having to actually subclass a class.

```
Test t = new Test()
{
   // data members and methods
   public void test_method()
   {
      ........
      ........
    }
};
```

The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code. Anonymous inner classes are generic created via below listed two ways as follows:

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

Types of Anonymous Inner Class

Based on declaration and behavior, there are 3 types of anonymous Inner classes:

1. Anonymous Inner class that extends a class
2. Anonymous Inner class that implements an interface
3. Anonymous Inner class that defines inside method/constructor argument

Diamond syntax, sometimes known as the diamond operator, It was added to Java 7 as just a new feature. The diamond operator **makes it easier to employ generics while building an object**.

Diamond Operator <>

Before Java 7	After Java 7	After Java 9
//Need to mention type on both sides List<String> list = new ArrayList<String>(); Set<Integer> set = new HashSet<Integer>(); Map<Integer, String> map = new HashMap<Integer, String>();	//No need to mention type on right side List<String> list = new ArrayList< >(); Set<Integer> set = new HashSet< >(); Map<Integer, String> map = new HashMap< >();	//No need to mention type on right side List<String> list = new ArrayList< >(); Set<Integer> set = new HashSet< >(); Map<Integer, String> map = new HashMap< >();

//Need to mention type on both sides for anonymous classes also
 	
Addition<Integer> integerAddition = new Addition<Integer>() {
@Override
void add(Integer t1, Integer t2)
{
System.out.println(t1+t2);
}
};
	//But, need to mention type on both sides for anonymous classes
	
	Addition<Integer> integerAddition = new Addition<Integer>() {
	@Override
	void add(Integer t1, Integer t2)
	{
	System.out.println(t1+t2);
	}
	};
		//No need to mention type on right side for anonymous classes also
		
		Addition<Integer> integerAddition = new Addition< >() {
		@Override
		void add(Integer t1, Integer t2)
		{
		System.out.println(t1+t2);
		}
		};

What is a diamond operator?

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

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

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

Problem with the diamond operator while working with Anonymous Inner classes

Java 7 allowed us to use diamond operator in normal classes but it didn't allow us to use them in anonymous inner classes. Lets take an example:

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

Output:

We got a compilation error when we ran the above code in Java SE 8.

Java 9 – Diamond operator enhancements

Java 9 improved the use of diamond operator and allows us to use the diamond operator with anonymous inner classes. Lets take the same example that we have seen above.

Running this code in Java SE 9

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

Output:

```
201
```

Local Variable Type Inference

What is type inference?

Type inference refers to the automatic detection of the datatype of a variable, done generally at the compiler time.

What is 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.

Why has this feature been introduced?

Till Java 9, to define a local variables of class type, the following was the only correct syntax: `Class_name variable_name=new Class_name(arguments);`

For example:

```
class A {
    public static void main(String a[])
    {
        String s = " Hi there";
    }
}
```

It looks fine, right? Yeah, because this is how things have been since the inception of Java. But there is one issue: It's pretty obvious that if the type of the object is clearly mentioned at the right side of the expression, mentioning the same thing before the name of the variable makes it redundant. Plus, in the second example, you can see that it's obvious that after the '=' sign, it's clearly a string as nothing except for a string can be enclosed in double inverted commas. Therefore, there arose a need to eliminate this redundancy and make variable declaration shorter, and more convenient

How to declare local variables using LVTI:

Instead of mentioning the variable datatype on the left-side, before the variable, LVTI allows you to simply put the keyword 'var'. For example,

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

Can be re-written as:

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

Use Cases

Here are the cases where you can declare variables using LVTI:

1. **In a static/instance initialization block**

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

2. **As a local variable**

```java
// Declaration of a local variable in java 10 using LVTI
class A {
    public static void main(String a[])
    {
        var x = "Hi there";
        System.out.println(x)
    }
}
```

3. As iteration variable in enhanced for-loop

```
class A {
    public static void main(String a[])
    {
        int[] arr = new int[3];
        arr = { 1, 2, 3 };
        for (var x : arr)
            System.out.println(x + "\n");
    }
}
```

4. As looping index in for-loop

```
class A {
    public static void main(String a[])
    {
        int[] arr = new int[3];
        arr = { 1, 2, 3 };
        for (var x = 0; x < 3; x++)
            System.out.println(arr[x] + "\n");
    }
}
```

5. As a return value from another method

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

6. As a return value in a method

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

Error cases: There are cases where declaration of local variables using the keyword 'var' produces an error. They're mentioned below:

1. **Not permitted in class fields**
2. **Not permitted for uninitialized local variables**
3. **Not allowed as parameter for any methods**
4. **Not permitted in method return type**
5. **Not permitted with variable initialized with 'NULL'**

Switch Expressions

To better understand the need for this syntax improvement, let's first examine how this task would have been formulated with the old syntax. Afterwards, we'll look at the advantages provided by the new syntax of switch.

Let's start with the implementation of mapping weekdays of type DayOfWeek to their textual length using the older syntax:

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters;
switch (day)
{
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    numOfLetters = 6;
    break;
  case TUESDAY:
    numOfLetters = 7;
    break;
  case THURSDAY:
  case SATURDAY:
    numOfLetters = 8;
    break;
  case WEDNESDAY:
    numOfLetters = 9;
    break;
  default:
    numOfLetters = -1;
}
```

Modern Switch Expressions

Let's have a critical look at the source code. First of all, the shown construct does not appear elegant and is also quite long. The multiple specifications of values need accustoming, too. Even worse, a break is needed so that the processing runs without surprise and there is no fall-through. Moreover, we need to set the (artificial) auxiliary variable numOfLetters correctly in each branch. In particular, despite the actually complete coverage of the enum values, the default is necessary. Otherwise, the compiler complains that the variable numOfLetters may not be initialized – unless you have already assigned a value to it initially. So how is it better?

Syntax of the new Switch Expressions

With the new "Switch Expressions", expressing case distinctions is made much easier and provides an intuitive notation:

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
  case MONDAY, FRIDAY, SUNDAY -> 6;
  case TUESDAY                -> 7;
  case THURSDAY, SATURDAY     -> 8;
  case WEDNESDAY              -> 9;
};
```

From this example, we notice some syntactic innovations: In addition to the obvious arrow instead of the colon, multiple values can now be specified after the case. Conveniently, there is no more need for break: The statements after the arrow are only executed specifically for the case and no fall-through exists with this syntax. Besides, the switch can now return a value, which avoids the need to define auxiliary variables. Instead of just stating a value after the arrow, it is also possible to specify expressions such as assignments or method calls without any problems. And even better, of course, still without the need for a break. Furthermore, this is no longer allowed in the new syntax after the arrow.

Conclusion

The Java releases up to and including 13 are rather manageable in terms of their innovations. This is true even for Java 11 as an LTS version. Fortunately, Java 14 brings a good slew of useful enhancements:
On the one hand, there are the convenient syntax changes in switch.

Let me conclude: The new syntax of switch seems to be just a small change, but it has an enormous effect on readability and ease of use.

yield keyword

Introduced in Java 13 as part of the enhancements in Project Amber, the '*yield*' keyword aims to simplify code, making switch expressions more concise and expressive. Let us learn about '*yield*' keyword, its purpose, syntax, and see some practical examples.

The 'yield' keyword enhances the switch expressions by allowing the expression to produce a result directly. Let us understand with an example.

In the following program, the *switch* expression evaluates the variable '*day*' value.

- If 'day' matches any of the weekday (MON through FRI), it prints "It is WeekDay" to the console and sets the boolean result to **`true`**.
- If 'day' matches SAT or SUN, it prints "It is Weekend" to the console and sets the boolean result to **`false`**.
- The **`true`** and **`false`** values are implicitly returned as results for the corresponding cases.

Program without 'yield' Keyword

```
Boolean result = switch(day) {

    case MON, TUE, WED, THUR, FRI -> {
        System.out.println("It is WeekDay");
        true;
    }
    case SAT, SUN -> {
        System.out.println("It is Weekend");
        false;
    }
};

System.out.println("Result is " + result);
```

Let us rewrite the above **program using the *yield* keyword** again, and notice the difference.

```
Program with 'yield' Keyword

Boolean result = switch(day) {

    case MON, TUE, WED, THUR, FRI -> {
        System.out.println("It is WeekDay");
        yield true;
    }
    case SAT, SUN -> {
        System.out.println("It is Weekend");
        yield false;
    }
};

System.out.println("Result is " + result);
```

In this version, the only difference is the explicit use of the '*yield*' keyword which serves two purposes:

- **It provides a value for the *switch* expression.**
- **It acts as a terminator, indicating that the control flow should exit the *switch* expression.**

You can notice that both versions of the program achieve the same functionality. However, the '*yield*' keyword in the second version adds explicitness to the *return* statement within the case blocks, thus making it clear that a value is being returned and terminating the *switch* expression.

Difference between ‘yield’ and ‘return’ Keywords

The ‘yield’ and ‘return’ keywords in Java serve distinct purposes and are used in different contexts.

- **A *return* statement returns control to the invoker of a method or constructor.**
- **A *yield* statement transfers control by causing an enclosing `switch` expression to produce a specified value.**

Let us list down the differences in more detail:

Feature	Yield	Return
Used In	Methods, constructors, loops, lambda expressions	Specifically within *switch* expressions
Termination	Terminates the execution of the entire method.	Terminates the execution of the *switch* expression
Scope	General-purpose keyword used in various contexts.	Specialized for use within switch expressions and is not applicable elsewhere.

Java Text Blocks

- A text block is a multi-line string literal
- Avoids the need for most escape sequences, automatically formats the string in a predictable way.
- Gives developers control over the format when desired

Since Java 15, text blocks are available as a standard feature. With Java 13 and 14, we needed to enable it as a preview feature.

Text blocks start with a """(three double-quote marks) followed by optional whitespaces and a newline. The most simple example looks like this:

```
String example = """
    Example text""";
```

Note that the result type of a text block is still a *String*. Text blocks just provide us with another way to write *String* literals in our source code.

Inside the text blocks, we can freely use newlines and quotes without the need for escaping line breaks. It allows us to include literal fragments of HTML, JSON, SQL, or whatever we need, in a more elegant and readable way.

Enhances readability of multi-line strings that contains non-java language code such as HTML, JSON

Features of Text Blocks:

Escape Sequences

Concatenation with String Literals

Incidental White Spaces \s

Line Terminators \

Comparison with String Literals

Text blocks are added for supporting multi-line strings in Java, thereby avoiding the need for escape characters or concatenation

```
public String getBlockOfHtml() {
    return """
            <html>

                <body>
                    <span>example text</span>
                </body>
            </html>""";
}
```

Java Record

A *Java Record* is a special kind of Java class which has a concise syntax for defining immutable data-only classes. Java Record instances can be useful for holding records returned from a database query, records returned from a remote service call, records read from a CSV file, or similar types of use cases.

A Java Record consist of one or more data fields which corresponds to member variables in a regular Java class. The Java compiler auto generates getter methods, toString(), **hashcode() and equals() methods** for these data fields, so you don't have to write that boilerplate code yourself. Since a Java Record is immutable, no setter methods are generated.

Java Record Syntax

The Java Record syntax is quite simple. Here is an example of a Java Record modeling a Vehicle:

```
public record Vehicle(String brand, String licensePlate) {}
```

Notice how the example uses the `record` instead of `class`. The `record` keyword is what tells the Java compiler that this type definition is a record.

Notice also how the record defined in the example has no explicit **Java field definitions**. The record is defined solely by what looks like a regular **Java constructor** . That constructor is actually enough to define a Java Record. The two parameters defined in the Record constructor tells the Java compiler that the record type has two fields - one field per parameter in the constructor. The Java compiler then generates the corresponding fields, getter methods and a hashCode() and equals() method.

Using a Java Record

You use a Java Record just like you use other Java classes - by creating instances of the record type using the Java `new` keyword. Here is an example of using the Java Record type Vehicle defined in the previous section:

```
public class RecordsExample {

  public static void main(String[] args) {

    Vehicle vehicle = new Vehicle("Mercedes", "UX 1238 A95");

    System.out.println( vehicle.brand() );
    System.out.println( vehicle.licensePlate() );

    System.out.println( vehicle.toString() );

  }
}
```

Notice how the Java compiler has generated a `brand()` method, a `licensePlate()` method and a `toString()` method for us. The output printed from the above example would be:

```
Mercedes
UX 1238 A95
Vehicle[brand=Mercedes, licensePlate=UX 1238 A95]
```

A Record is Final

A Record type definition is final, meaning you cannot create subclasses (subrecords) of a Java Record type.

Sealed Class in Java

In programming, security and control flow are the two major concerns that must be considered while developing an application. There are various controlling features such as the use of final and protected keyword restricts the user to access variables and methods. Java 15 introduces a new **preview feature** that allows us to control the inheritance. In this section, we will discuss the **preview feature, the concept of sealed class, and interface** with proper examples.

Java 15 introduced the concept of **sealed** classes. It is a preview feature. Java sealed classes and interfaces restrict that which classes and interfaces may extend or implement them.

In other words, we can say that the class that cannot be inherited but can be instantiated is known as the sealed class. It allows classes and interfaces to have more control over their permitted subtypes. It is useful both for general domain modeling and for building a more secure platform for libraries.

Note that the concept of sealed classes is a **preview feature,** not a permanent feature.

Uses of Sealed Class

Sealed classes work well with the following:

- Java Reflection API
- Java Records
- Pattern Matching

Advantages of Sealed Class and Interface

- It allows permission to the subclasses that can extend the sealed superclass.
- It makes superclass broadly accessible but not broadly extensible.
- It allows compilers to enforce the type system on the users of the class.
- Developer of a superclass gets control over the subclasses. Hence, they can define methods in a more restricted way.

Defining a Sealed Class

The declaration of a sealed class is not much complicated. If we want to declare a class as sealed, add a **sealed** modifier to its declaration. After the class declaration and extends and implements clause, add **permits** clause. The clause denotes the classes that may extend the sealed class.

It presents the following modifiers and clauses:

- **sealed:** It can only be extended by its permitted subclasses.
- **non-sealed:** It can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this.
- **permits:** It allows the subclass to inherit and extend.
- **final:** The permitted subclass must be final because it prevents further extensions.

Example of Sealed Class

```
public class SealedClassExample
{
public static void main(String args[])
{
//creating an instance of GrandFather class
Person grandfather = new GrandFather(87, "Albert");
grandfather.name = "Albert";
System.out.println("The age of grandfather is: "+getAge(grandfather));
}
// getting the age of the Person
public static int getAge(Person person)
{
//if the person is an instance of the Father class, returns the age of the father
if (person instanceof Father)
{
//cast the person class to Father class and get the age
return ((Father) person).getFatherAge();
}
//if the person is an instance of the GrandFather class, returns grandfather age
else if (person instanceof GrandFather)
{
return ((GrandFather) person).getGrandFatherAge();
}
//returns nothing if does not match with any of the above conditions
return -1;
}
}
//the class person extends only Father and GrandFather class
abstract sealed class Person permits Father, GrandFather
{
String name;
String getName()
{
return name;
}
}
//non-sealed class extends unknown subclass (Person)
non-sealed class GrandFather extends Person
{
int age;
GrandFather(int age, String name)
{
this.age = age;
this.name = name;
}
int getGrandFatherAge()
{
return age;
}
}
```

Output:

```
The age of grandfather is: 87
```