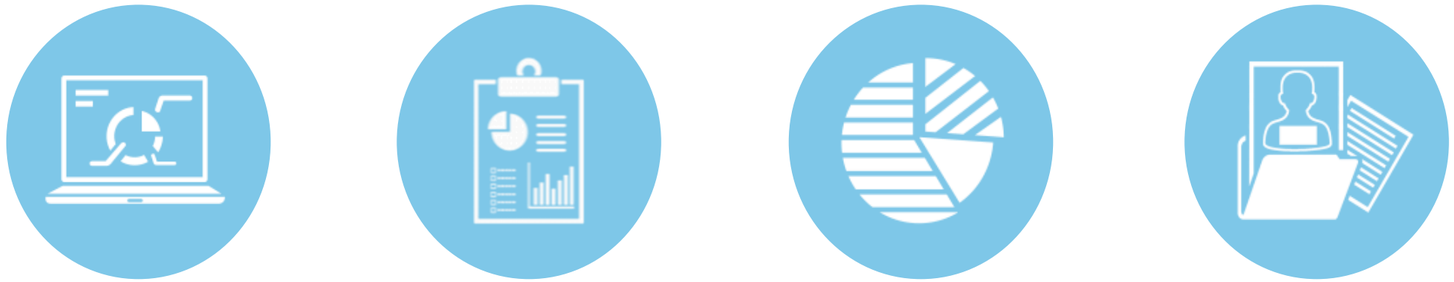
Core Java

Lesson 13—Introduction to Java 8



Learning Objectives

At the end of this lesson, you should be able to:

- Explain the key features of Java 8

Core Java

Topic 1—Key Features of Java 8

Key Features of Java 8

The key features of Java that will be discussed in this lesson include:

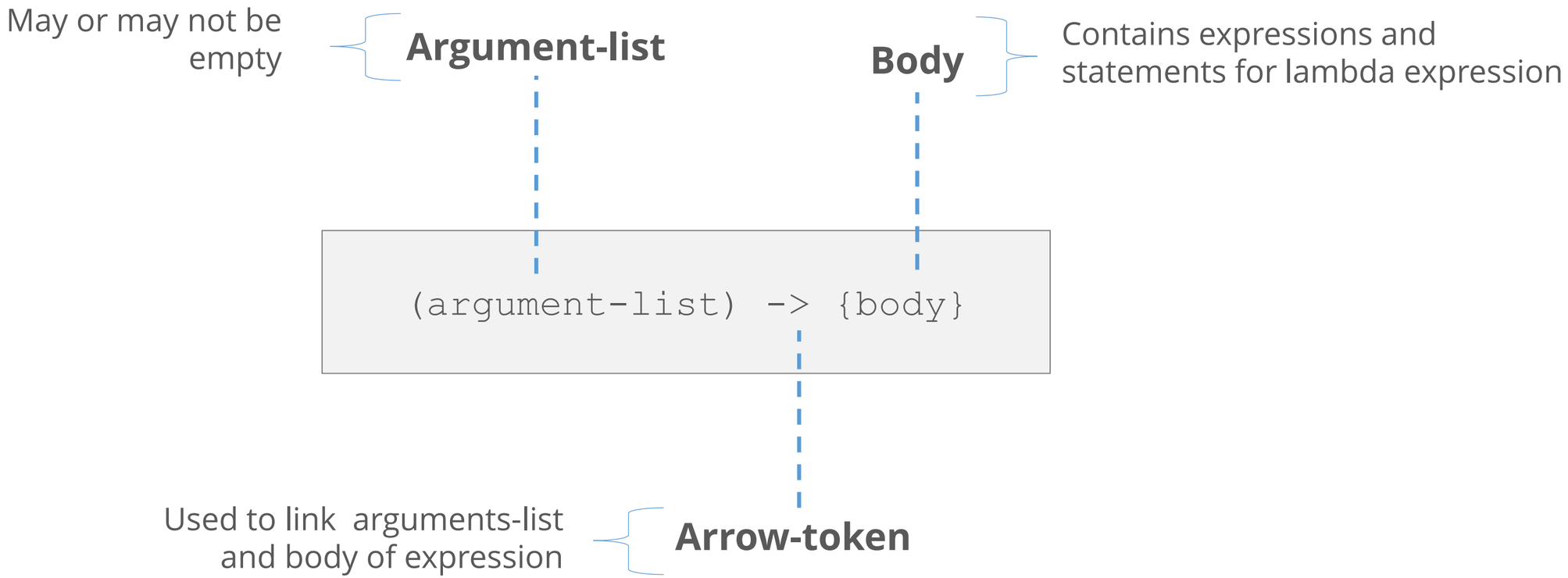
- Lambda Expressions
- Method References
- Functional Interfaces
- Stream Application Programming Interface (API)
- Default Methods
- Base64 Encode Decode
- Optional Class
- Collectors Class
- forEach() Method
- Parallel Array Sorting
- Java Nashorn
- Type and Repeating Annotations
- IO Enhancements
- Concurrency Enhancements
- JDBC Enhancements

Lambda Expression

- An important feature of Java SE8 that allows developers to write code in the functional style
- Implements Functional Interface (an interface that has only one abstract method)
- Provides a clear and concise method to implement Single Abstract Method (SAM) interface by using an expression
- Requires less coding
- Useful in collection library—helps to iterate, filter, and extract data

Lambda Expression (Contd.)

Lambda Expression has of three components:



Lambda Expression (Contd.)

Example:

```
1.  interface Drawable
2. {
3.       public void draw();
4.  }
5.   public class LambdaExpressionExample {
6.     public static void main(String[] args) {
7.         int width=10;

8. //with lambda
9.         Drawable d2=()->{
10.            System.out.println("Drawing "+width);
11.        };
12.        d2.draw();
13.    }
14. }
```

Method Reference

- A new feature of Java 8, used to refer method of functional interface
- A compact and easy form of lambda expression

Each time you use a lambda expression to refer a method, you can replace your lambda expression with method reference.

Method Reference (Contd.)

There are four types of method references:

01 Reference to a static method

02 Reference to an instance method of a particular object

03 Reference to an instance method of an arbitrary object of a particular type

04 Reference to a constructor

Method Reference (Contd.)

Example:

```
1.interface Sayable
2.{
3.    void say();
4.}
5.public class MethodReference {
6.    public static void saySomething(){
7.        System.out.println("Hello, this is static method.");
8.    }
9.    public static void main(String[] args) {
10.        // Referring static method
11.        Sayable sayable = MethodReference::saySomething;
12.        // Calling interface method
13.        sayable.say();
14.    }
15.}
```

Functional Interface

- An interface that can contain a number of default, static methods, but only one abstract method
- Also known Single Abstract Method Interface
- Can also declare methods of object class
- Requires helps to achieve functional programming approaches and less coding

Functional Interface (Contd.)

Example:

```
1.interface sayable
2.{
3.    void say(String msg);
4.}
5.public class FunctionalInterfaceExample implements sayable{
6.    public void say(String msg){
7.        System.out.println(msg);
8.    }
9.    public static void main(String[] args) {
10.         FunctionalInterfaceExamplefile= new FunctionalInterfaceExample();
11.         fie.say("Hello there");
12.     }
13.}
```

Stream Application Programming Interface

- Provides a new package in Java 8 called java.util.stream (contains classes, interfaces, and enum) that allows functional-style operations on the elements
- Execute only when it is required
- Can be used to filter, collect, print, and convert one data structure to other

Java enum or enumerated type is a special Java type that defines collections of constants. It contains constants, method, etc.

Stream Application Programming Interface (Contd.)

Characteristics of Steam API:

01 Does not store elements.

02 Functional in nature

03 Evaluates code only when required

04 Elements of a stream are only visited once during the life of a stream

Stream Application Programming Interface (Contd.)

Example:

```
1. public class JavaStreamExample
2.{
3.    public static void main(String[] args) {
4.        List<Product> productsList = new ArrayList<Product>();
5.        //Adding Products
6.        productsList.add(new Product(1,"HP Laptop",25000f));
7.        productsList.add(new Product(2,"Dell Laptop",30000f));
8.        productsList.add(new Product(3,"Lenevo Laptop",28000f));
9.        productsList.add(new Product(4,"Sony Laptop",28000f));
10.        productsList.add(new Product(5,"Apple Laptop",90000f));
11.        List<Float> productPriceList = new ArrayList<Float>();
12.        for(Product : productsList){   // filtering data of list
13.         if(product.price<30000){
14.           productPriceList.add(product.price);  // adding price to a productPriceList
 }
15.        }
16.        System.out.println(productPriceList);   // displaying data
17.    }
18.}
```

Default Methods

- Non-abstract methods defined inside the interface and tagged as default

In Java 8, you can create default methods inside the interface.

Default Methods (Contd.)

Example:

```
1.interface Sayable
2.{                                    // Default method
3.    default void say(){
4.        System.out.println("Hello, this is default method");
5.    }                                                    // Abstract method
6.    void sayMore(String msg);
7.}
8.public class DefaultMethods implements Sayable{
9.    public void sayMore(String msg){                     // implementing abstract method
10.        System.out.println(msg);
11.    }
12.    public static void main(String[] args) {
13.        DefaultMethods dm = new DefaultMethods();
14.        dm.say();                                  // calling default method
15.        dm.sayMore("Example");             // calling abstract method
16.
17.    }
18.}
```

Output: Hello, this is default method Example

Base64 Encode and Decode

- Base64 deal with encryption
- Provides three different encoders and decoders to encrypt information at each level
  - Basic Encoding and Decoding
  - URL and Filename Encoding and Decoding
  - Multi-purpose Internet Mail Extension (MIME)

Data can be encrypted and decrypted using the provided methods. Import java.util.Base64 in your source file to use its methods.

Base64 Encode and Decode (Contd.)

Basic Encoding and Decoding

- Uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations
- Encoder does not add any line separator character
- Decoder rejects data that contains characters outside the base64 alphabet

URL and Filename Encoding and Decoding

- Uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations
- Encoder does not add any line separator character
- Decoder rejects data that contains characters outside the base64 alphabet

Base64 Encode and Decode (Contd.)

Basic Encoding and Decoding

- Uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations
- No line separator is added to the end of the encoded output
- Rejects line separators or other characters not found in the base64 alphabet table

Base64 Encode and Decode (Contd.)

```
1. import java.util.Base64;
2.  publicclass Base64BasicEncryptionExample {
3.    publicstaticvoid main(String[] args) {                  // Getting encoder
4.        Base64.Encoder encoder = Base64.getEncoder();      // Creating byte array
5.        bytebyteArr[] = {1,2};                            // encoding byte array
6.        bytebyteArr2[] = encoder.encode(byteArr);
7.        System.out.println("Encoded byte array: "+byteArr2);
8.        bytebyteArr3[] = newbyte[5];               // Make sure it has enough size to store copied bytes
9.        intx = encoder.encode(byteArr,byteArr3);   // Returns number of bytes written
10.       System.out.println("Encoded byte array written to another array: "+byteArr3);
11.       System.out.println("Number of bytes written: "+x);
12.       String str = encoder.encodeToString(“Sample".getBytes());     // Encoding string
13.       System.out.println("Encoded string: "+str);
14.       Base64.Decoder decoder = Base64.getDecoder();               // Decoding string
15.       String dStr = new String(decoder.decode(str));
16.       System.out.println("Decoded string: "+dStr);
17.    }
18.}
```

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: Sample
```

Optional Class

- A new, public final class in Java 8 that deals with NullPointerException in Java application
- Provides methods to check the presence of value for a particular variable

You must import java.util package to use this class

Optional Class (Contd.)

Example:

```
1.import java.util.Optional;
2.public class OptionalExample {
3.    public static void main(String[] args) {
4.        String[] str = new String[10];
5.        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";     // Setting value for 5th index
6.        Optional<String> checkNull = Optional.ofNullable(str[5]);
7.        checkNull.ifPresent(System.out::println);  // printing value by using method reference
8.        System.out.println(checkNull.get());      // printing value by using get method
9.        System.out.println(str[5].toLowerCase());
10.    }
11.}
```

Output:
JAVA OPTIONAL CLASS EXAMPLE
JAVA OPTIONAL CLASS EXAMPLE
java optional class example

Collectors Class

- A new, final class in Java 8 that extends object class
- Provides reduction operations, such as gathering elements into collections and summarizing them according to various criteria
- Provides various methods to deal with elements

Collectors Class (Contd.)

Example:

```
1.public class CollectorsExample
2.{
3.    public static void main(String[] args) {
4.        List<Product> productsList = new ArrayList<Product>();
5.        //Adding Products
6.        productsList.add(new Product(1,"HP Laptop",20000f));
7.        productsList.add(new Product(2,"Dell Laptop",35000f));
8.        productsList.add(new Product(3,"Lenevo Laptop",30000f));
9.        productsList.add(new Product(4,"Sony Laptop",30000f));
10.        productsList.add(new Product(5,"Apple Laptop",60000f));
11.        List<Float> productPriceList =
12.                productsList.stream()
13.                            .map(x->x.price)                // fetching price
14.                            .collect(Collectors.toList());  // collecting as list
15.        System.out.println(productPriceList);
16.    }
17.}
```

Output:

[20000.0, 35000.0, 30000.0, 30000.0, 60000.0]

forEach() Method

- A default method defined in the Iterable interface that can be used to iterate the elements
- Takes a single parameter that is a functional interface

Since this method takes a single parameter, you can pass lambda expression as an argument.

forEach() Method (Contd.)

Example:

```
1.import java.util.ArrayList;
2.import java.util.List;
3.public class ForEachExample {
4.    public static void main(String[] args) {
5.        List<String> gamesList = new ArrayList<String>();
6.        gamesList.add("Football");
7.        gamesList.add("Cricket");
8.        gamesList.add("Chess");
9.         System.out.println("-------Iterating by passing lambda expression---------");
10.        gamesList.forEach(games -> System.out.println(games));
11.
12.    }
13.}
```

Output:

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

Parallel Array Sorting

- A new method in Java 8 that has been added to java.util.Arrays class
- Uses the JSR 166 Fork/Join parallelism common pool to provide sorting of array elements
- Also called “parallelSort()” method

Parallel Array Sorting (Contd.)

Example:

```
1.import java.util.Arrays;
2.public class ParallelArraySorting {
3.    public static void main(String[] args) {
4.        // Creating an integer array
5.        int[] arr = {5,8,1,0,6,9};          // Iterating array elements
6.        for (int i : arr) {
7.            System.out.print(i+" ");
8.        }
9.         Arrays.parallelSort(arr);        // Sorting array elements parallel
10.        System.out.println("\nArray elements after sorting");
11.         for (int i : arr) {
12.            System.out.print(i+" ");    // Iterating array elements
13.        }
14.    }
15.}
```

Output:

5 8 1 0 6 9

Array elements after sorting

0 1 5 6 8 9

Java Nashorn

- A JavaScript engine
- Provides a command-line tool, **jjs**, that allows you to execute JavaScript code dynamically at JVM

JavaScript code can be executed by using jjs command-line tool and by embedding Java Nashorn into Java source code.

Java Nashorn (Contd.)

Example:

```
1.import javax.script.*;
2.import java.io.*;
3.public class NashornExample {
4.    public static void main(String[] args) throws Exception{   // Creating script engine
5.        ScriptEngine ee = new ScriptEngineManager().getEngineByName("Nashorn is JavaScript engine
");
6.        ee.eval(new FileReader("js/hello.js"));                // Reading Nashorn file
7.    }
8.}
```

Output:

Hello Nashorn is JavaScript engine

Type and Repeating Annotations

New features included in Java 8 annotations topic

Earlier you could apply annotations only to declarations. After the release of Java SE 8, annotations can be applied to any type use.

Examples:

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

Type and Repeating Annotations (Contd.)

Types of Annotations:
Repeatable Annotation Type
Containing Annotation Type

Type and Repeating Annotations (Contd.)

Types of Annotations:

Repeatable Annotation Type

Repeatable annotation type must be declared using the @Repeatable meta-annotation.

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

```
1.@Repeatable(Games.class)
2.@interfaceGame{
3.     String name();
4.     String day();
5.}
```

Containing Annotation Type

Type and Repeating Annotations (Contd.)

Types of Annotations:
Repeatable Annotation Type
Containing Annotation Type

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

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

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

```
1.@interfaceGames{
2.    Game[] value();
3.}
```

IO Enhancements

- Several improvements to the standard (java.nio.charset.Charset) and extended charset (character set) implementations
- They include:
  - The /dev/poll SelectorProvider which continues to be the default
    To use the Solaris event port mechanism, run it with the system property java.nio.channels.spi.Selector set to the value sun.nio.ch.EventPortSelectorProvider
  - Decrease the size of <JDK_HOME>/jre/lib/charsets.jar file
  - Performance improvement for the java.lang.String(byte[], *) constructor and the java.lang.String.getBytes() method

Concurrency Enhancements

- New classes and interfaces in java.util.concurrent
- java.util.concurrent package contains two new interfaces and four new classes
  - Interface CompletableFuture.AsynchronousCompletionTask: A marker interface identifying asynchronous tasks produced by async methods
  - Interface CompletionStage<T>: A stage of a possibly asynchronous computation that performs an action or computes a value when another CompletionStage is completed

Concurrency Enhancements (Contd.)

- Class CompletableFuture<T>: A Future that may be explicitly completed (setting its value and status) and may be used as a CompletionStage, supporting dependent functions and actions that are triggered upon its completion
- Class ConcurrentHashMap.KeySetView<K,V>: A view of a ConcurrentHashMap as a Set of keys in which additions may optionally be enabled by mapping to a common value
- Class CountedCompleter<T>: A ForkJoinTask with a completion action performed when triggered and there are no pending actions
- Class CompletionException: Exception is thrown when an error or other exception is encountered in the course of completing a result or task

Concurrency Enhancements (Contd.)

New methods in java.util.concurrent.ConcurrentHashMap:

- The Collections Framework has undergone a major revision in Java 8 to add aggregate operations based on the newly added streams facility and lambda expressions. As a result, the ConcurrentHashMap class introduces over 30 new methods. These include various forEach methods (forEach, forEachKey, forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries), and a large number of reduction methods (reduce, reduceToDouble, reduceToLong).

New classes in java.util.concurrent.atomic:

- Maintaining a single count, sum, etc., that is updated by possibly many threads is a common scalability problem. This release introduces scalable, updatable, and variable support through a small set of new classes (DoubleAccumulator, DoubleAdder, LongAccumulator, LongAdder) that internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables. This is made possible by relaxing atomicity guarantees in a way that is acceptable in most applications.

Concurrency Enhancements (Contd.)

New methods in java.util.concurrent.ForkJoinPool

- A static commonPool() method is now available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use and reinstated upon subsequent use). Two new methods (getCommonPoolParallelism() and commonPool()) have been added. These return the targeted parallelism level of the common pool or the common pool instance, respectively.

New class java.util.concurrent.locks.StampedLock

- A new StampedLock class adds a capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes.

JDBC Enhancements

- Addition of REF_CURSOR support
- Addition of java.sql.DriverAction Interface
- Addition of security check on deregisterDriver Method in DriverManager Class
- Addition of the java.sql.SQLType Interface
- Addition of the java.sql.JDBCType Enum
- Add Support for large update counts
- Changes to the existing interfaces
- Rowset 1.2: Lists the enhancements for JDBC RowSet

Key Takeaways

- Lambda expression is an important feature of Java SE8 that allows developers to write code in the functional style.
- Method reference feature is used to refer method of functional interface.
- Functional interface is an interface that can contain a number of default, static methods, but only one abstract method.
- Stream application programming interface provides a new package in Java 8 called java.util.stream that allows functional-style operations on the elements.
- Default method is a non-abstract methods defined inside the interface and tagged as default.
- Optional class is a new, public final class in Java 8 that deals with NullPointerException in Java application.
- Collectors class is a new final class that extends object class.

Quiz

QUIZ 1

In Java 8, interfaces can have:

a. Static methods

b. Non-static public methods

c. Default methods with body

d. Private methods with body

QUIZ 1

In Java 8, interfaces can have:

a. Static methods

b. Non-static public methods

c. Default methods with body

d. Private methods with body

The correct answer is **a and c.**

In java8, we can have default and static methods with body under default classes.

QUIZ 2

Which of the following are new operators or expressions added to java 8?

a. &=

b. ->

c. ::

d. *=

QUIZ 2

Which of the following are new operators or expressions added to java 8?

a. &=

b. ->

c. ::

d. *=

The correct answer is **b and c.**

Option “b” is called lambda expression and option “c” is called method reference operator. These are new additions to java 8.

Thank You