CitiusTech

Introduction to Lambda Expressions

Session 1

Version 1.0

January 2019

Agenda

- **Imperative & Declarative Style of Programming**
- Lambda Expression & Functional Interfaces
- Traversing the Collections
- Method References
- References

Imperative Style of Programming in Java

- **Imperative style of programming in Java**
  - Java has provided imperative style of programming, since version 1.0.
  - In this style, we instruct VM on every step that what we want it to do and observe its behavior.
  - That's worked fine, but it's a bit low level.
  - The code tends to get verbose, and we often wish the Java language were a little more intelligent.
  - Let's start on familiar ground to see the imperative style in action in the following code snippet:

- Imperative version is noisy and low level
- Has several moving parts such as Initialize a smelly **boolean** flag named **found**

```
boolean found = false;
 for(String city : cities) {
   if(city.equals("Chicago")) {
            found = true;
            break;
    }
 }
    System.out.println("Found chicago?:" +
found);
```

Declarative Style of Programming in Java (1/2)

- The code can be quickly turn into something more concise and easier to read:
  - **System**.out.println(*"Found chicago?:"* + cities.contains(*"Chicago"*));
- **Declarative style** – the **contains** method helped us get directly to our business.
- The declarative function to check if an element is present in a collection has been around in Java for a very long time.
- The changes improved the code in the following ways:
  - No messing around with mutable variables
  - Iteration steps wrapped under the hood
  - Less clutter
  - Better clarity; retains our focus
  - Less impedance; code closely trails the business intent
  - Less error prone
  - Easier to understand and maintain

Declarative Style of Programming in Java (2/2)

- How to achieve declarative style of programming in Java?
  - The idea lead to use the **functional style of programming** along with Object style, a hybrid language with **declarative style** of programming instead of imperative style.
  - The Java language team has put in substantial time and effort to bring functional capabilities to the language and the JDK.
  - To reap the benefits, we have to pick up a few new concepts.
    - Be declarative
    - Promote immutability
    - Avoid side effects
    - Prefer expressions over statements
    - Design with higher-order functions

Declarative Style - Guidelines

Be declarative

- We saw how the declarative use of the **contains()** method—when used on an immutable collection—was far easier to work with than the imperative style. Immutability and declarative programming are the essence of the functional style of programming, and Java now makes them quite approachable.

Promote immutability

- By avoiding mutability we can create pure functions—that is, functions with no side effects.

Avoid side effects

- A function with no side effects honors immutability and does not change its input or anything in its reach. These functions are easier to understand, have fewer errors, and are easier to optimize. The lack of side effects removes any concerns of race conditions or updates. As a result we can also easily parallelize execution of such functions

Prefer expressions over statements

- Statements are stubborn and force mutation. Expressions promote immutability and function composition.

Design with higher-order functions

- A higher-order function takes the concept of reuse to the next level. Instead of solely relying on objects and classes to promote reuse, with higher-order functions we can easily reuse small, focused, cohesive, and well-written functions. With higher-order functions we can
  - Pass functions to functions
  - Create functions within functions
  - Return functions from functions

Agenda

- Imperative & Declarative Style of Programming
- **Lambda Expression & Functional Interfaces**
- Traversing the Collections
- Method References
- References

Java Lambda Expressions

- Java lambda expressions are new in Java 8.
- Java lambda expressions are Java's first step into functional programming.
- A Java lambda expression is thus a function which can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.
- Using lambda expressions, we can write code that’s is:
  - Elegant and concise
  - Less prone to errors
  - More efficient
  - Easier to optimize
  - Enhance
  - Parallelize.

Lexpression provides implementation of Functional Interfaces in Java Lambda 8.

Functional Interface

- A functional interface which has only one abstract method is called functional interface.
- Java provides an annotation named ***@FunctionalInterface***, which is used to declare an interface as functional interface.
- Following code snippet shows a definition of functional interface:

```
public interface FileFilter {
    boolean accept(File pathname);
}
```

Lambdas and Functional Interface (1/2)

- Functional programming is very often used to implement event listeners.
- Event listeners in Java are often defined as Java interfaces with a single method.

```
public interface StateChangeListener {
  public void onStateChange(State oldState, State newState);
 }
```

- In Java 7, we can implement this interface using an anonymous class implementation as shown:

```
StateOwner stateOwner = new StateOwner();
stateOwner.addStateListener(new StateChangeListener() {
    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
} });
```

- Here, in the code, a class named **StateOwner**, invokes a method **addStateListener** and implements an anonymous class for the interface implementation.

Lambdas and Functional Interface (2/2)

- In Java 8 you can add an event listener using a Java lambda expression, like this:

```
StateOwner stateOwner = new StateOwner();
stateOwner.addStateListener(
        (oldState, newState) -> System.out.println("State changed") );
);
```

- The Lambda expression in the code is:

```
(oldState, newState) -> System.out.println("State changed") );
```

Lambda Expression Syntax

- The syntax is as follows:

```
(argument-list) -> {body}
```

- Java lambda expression is consisted of three components.
  - **Argument-list:** It can be empty or non-empty as well.
  - **Arrow-token:** It is used to link arguments-list and body of expression.
  - **Body:** It contains expressions and statements for lambda expression.
- Java lambda expressions can only be used where the type they are matched against is a single method interface.

Structure of Lambda Expressions (1/2)

- A lambda expression can have zero, one or more parameters.
- The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
- Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
- Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a
- The body of the lambda expressions can contain zero, one or more statements.

Structure of Lambda Expressions (2/2)

- If body of lambda expression has single statement, curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
- When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.
- Some of the existing interfaces that are treated as functional interfaces in Java 8:
  - Runnable
  - Comparator
  - Callable
  - FileFilter

Java Code without Lambda Expression

- Let's implement the **FileFilter** interface present in java.io package in Java 8.
- The definition of the **FileFilter** interface is as follows:

```
/**
     A filter for abstract pathnames.
**/
@FunctionalInterface
public interface FileFilter {
      boolean accept(File pathname)
 }
```

FileFilter Interface (1/2)

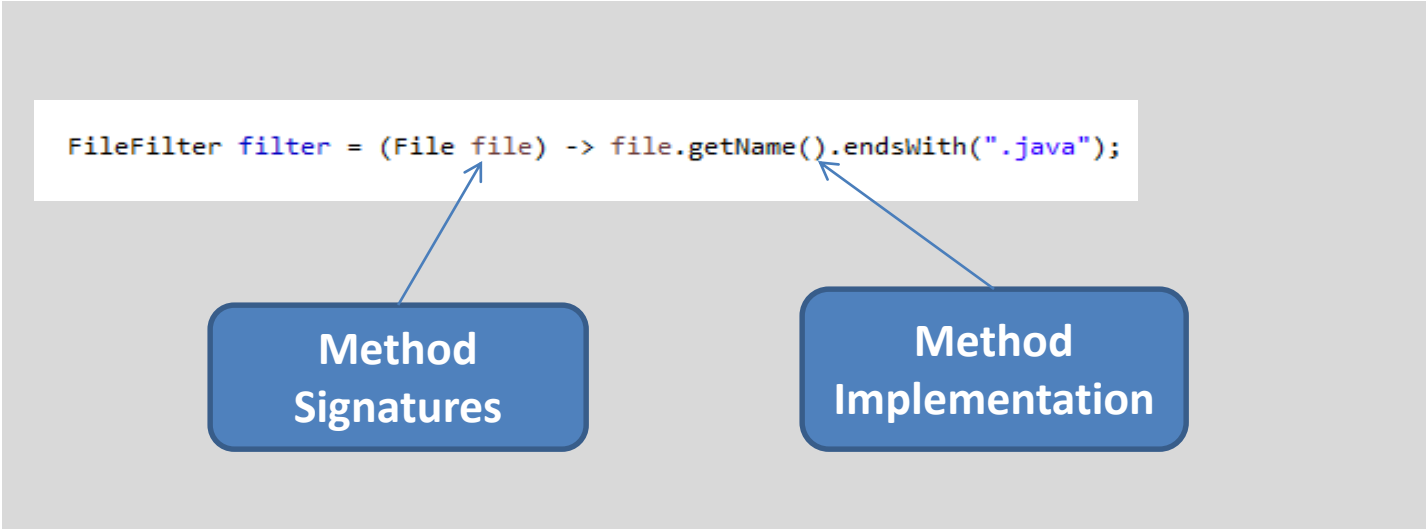
- The following anonymous class shows the implementation for **FileFilter** interface:

```
. . .
FileFilter fileFilter = new FileFilter() {
 @Override
  public boolean accept(File file)
   {
            return file.getName().endsWith(“.java”);
   }
};

. . .
File dir = new File(“d:/tmp”);
File[] javaFiles = dir.listFiles(fileFilter);
. . .
```

FileFilter Interface (2/2)

- The following code snippet shows the equivalent code for **FileFilter** in Java 8 using Lambda expressions:



Lambda Expression Types

- Few more functional interfaces with only one abstract method.
- These interfaces can be written as Lambda expressions.

```
public interface Runnable {
        void run();
}

public interface Comparator<T> {
    int compareTo(T t1, T t2);
}

public interface FileFilter {
   boolean accept(File pathname);
}
```

Class Assignment

- Write Lambda expressions for the Runnable and Comparator interfaces.

Agenda

- Imperative & Declarative Style of Programming
- Lambda Expression & Functional Interfaces
- **Traversing the Collections**
- Method References
- References

Lambda Expressions Parameters

- As Java lambda expressions are effectively just methods, lambda expressions can take parameters just like methods.
- The parameters passed in the Lambda expressions must match the parameters of the method in the single method interface.

Lambda Expression – Zero Parameter

- The following code snippet shows an interface whose method do not accept any parameters.

```
interface Sayable{
    public String say();
}
public class LambdaZeroParameter {
  public static void main(String[] args) {
            Sayable s = () -> "Nothing to say";

 // When more than one statement, then enclose the body part in the curly braces
            Sayable s1 = () -> {
                  return "Nothing to say";
          };

   // Invoke the method
            System.out.println(s.say());
   }
}
```

Lambda Expression – Single Parameter

- The following code snippet shows an interface whose method accept parameter.

```
interface Sayable{
    public String say(String name);
}
public class LambdaZeroParameter {
  public static void main(String[] args) {
            Sayable s = (name) -> "Nothing to say “ + name;

 // When more than one statement, then enclose the body part in the curly braces
            Sayable s1 = (name) -> {
                  return "Nothing to say “ +  name;
          };

   // Invoke the method
            System.out.println(s.say(“John”));
   }
}
```

Lambda Expressions – Multiple Parameters

- The following code snippet defines an interface with multiple parameters and shows the Lambda expression for the same:

```
interface Addable{
   int add(int a, int b);
}

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

      // Multiple parameters in lambda expression
      Addable ad1=(a, b) -> a + b;
      System.out.println(ad1.add(10,20));

      // Multiple parameters with data type in lambda expression
      Addable ad2=(int a, int b) ->  a + b;
      System.out.println(ad2.add(100,200));
   }
}
```

Lambda Expressions – Type Inference

- Type Inference means that the data type of any expression (e.g. method return type or parameter type) can be deduced automatically by the compiler.
- Java 8 Lambda expressions also support Type inference.
- The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. *The target type of an expression is the data type that the Java compiler expects depending on where the expression appears.*
- Before Java 8, the compiler often selected Object as the type argument (as in List<Object>), causing code not to compile when a more specific type was required.
- Compilers in Java 7 and earlier won't compile this code because they don't use target typing to infer types for method call arguments.
- **addAll()** expects to receive a java.util.Collection instance as its argument.
  Also, Arrays.asList() returns a java.util.List instance.
- Consider generics, addAll()'s target type is Collection<? extends String>,
  and Arrays.asList() returns a List<T> instance.
- In contrast, from target type Collection<? extends String>, the Java 8 compiler can infer that the value of type variable T is String.
- In situations where the Java compiler cannot infer types, you must explicitly specify values for type variables. You do so by using *type witnesses*, as demonstrated here:

```
birds.addAll(Arrays.<String>asList());
```

Agenda

- Imperative & Declarative Style of Programming
- Lambda Expression & Functional Interfaces
- Traversing the Collections
- **Method References**
- References

Method References

- Sometimes lambda expression does nothing but call an existing method.
- In those cases, it's often clearer to refer to the existing method by name.
- Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.
- The goal is to make your code more concise and more readable.
- You can use method references on four kinds of methods.
  - Static methods of any class,
  - instance methods of a particular object,
  - instance methods of an arbitrary object, in which case you would refer to it just like it were a static method,
  - References to constructor methods.
- This lambda expression:

```
Function<String, String> f = s -> s.toLowerCase();
```

- Can be written like that:

```
Function<String, String> f = String::toLowerCase;
```

Method References – Static Methods

- Sorting the data structure using the Collection.Sort method.The method signature of this invocation of sort is the following:
  - static <T> void sort(T[] a, Comparator<? super T> c)
- Define CompareAges static method in the Person class
  - public static int compareAges(Person person1, Person person2) { return Integer.compare(person1.getAge(), person2.getAge()); }
- Notice that the interface Comparator is a functional interface. Therefore, you could use a lambda expression instead of defining and then creating a new instance of a class that implements Comparator
  - Collections.sort(people, (p1, p2) -> Person.compareAges(p1, p2));
- Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:
  - Collections.sort(people, Person :: compareAges);
- The method reference Person::compareAges is semantically the same as the lambda expression (a, b) -> Person.compareByAge(a, b). Each has the following characteristics:
  - Its formal parameter list is copied from Comparator<Person>.compare, which is (Person, Person).
  - Its body calls the method Person. *compareAges*.

Method References – Instance Method of a Particular Object

- The following is an example of a reference to an instance method of a particular object:

```
class ComparisonProvider {
   public int compareByName(Person a, Person b) {
      return a.getName().compareTo(b.getName());
   }
public int compareByAge(Person a, Person b) {
      return a.getBirthday().compareTo(b.getBirthday());
   }
}
ComparisonProvider myComparisonProvider = new ComparisonProvider();
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);
```

- The method reference myComparisonProvider::compareByName invokes the method compareByName that is part of the object myComparisonProvider.
- The JRE infers the method type arguments, which in this case are (Person, Person).

Method References – Instance Method of an Arbitrary Object

- The following is an example of a reference to an instance method of an arbitrary object of a particular type:

```
String[] stringArray = { "Barbara", "James", "Mary", "John", "Patricia", "Robert",
"Michael", "Linda" };

Arrays.sort(stringArray, String::compareToIgnoreCase);
```

- The equivalent lambda expression for the method reference **String::compareToIgnoreCase** would have the formal parameter list (String a, String b), where a and b are arbitrary names used to better describe this example.
- The method reference would invoke the method **a.compareToIgnoreCase(b)**.

Method References – Constructor

- You can create references to the constructors. The only difference is in Constructor references the method name is "new".

```
public interface FuncIf  {
                        String strFunc(char[] chArray);
            }

            public class StringCreator {
               public static void main(String[] args) {
                     FuncIf fi = String::new;
                    char[] charArray = {'s','p','e','a','k','i','n','g','c','s'};
                    System.out.println(fi.strFunc(charArray));
                }
}
```

- In the program, the expression "String::new" creates a constructor reference to the String's constructor. Here the method StrFunc() takes a Character array as an argument, so the parameterized constructor String(char[] charArray) is referred here.
- The equivalent lambda expression is

  charArray -> new String(charArray);

Agenda

- Imperative & Declarative Style of Programming
- Lambda Expression & Functional Interfaces
- Traversing the Collections
- Method References
- **References**

References

- https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com