Lambda Expression

Lambda Expression

- Java lambda expressions are new in Java 8(March 2014).
- Java lambda expressions are commonly used to implement simple event listeners / callbacks, or in functional programming with the Java Streams API.

http://tutorials.jenkov.com/java/lambda-expressions.html

Java Lambdas and the Single Method Interface

- Event listeners in Java are often defined as Java **interfaces with a single method**. Here is a fictive single method interface example

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

Any interface with a SAM(Single Abstract Method) is a **functional interface**, and its implementation may be treated as lambda expressions

Java Lambdas and the Single Method Interface

- Imagine you have a class called StateOwner which can register state event listeners.

```
public class StateOwner {
  public void addStateListener(StateChangeListener listener) { ... }
}
```

- Before Java 8, you could add an event listener using an anonymous interface implementation, like this

```
StateOwner stateOwner = new StateOwner();
stateOwner.addStateListener(new StateChangeListener() {
      public void onStateChange(State oldState, State newState) {
          System.out.println("State changed");
      }
  }
);
```

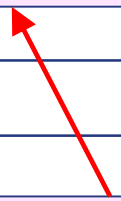
Java Lambdas and the Single Method Interface

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

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

```
public class StateOwner {
  public void addStateListener(StateChangeListener listener) { ... }
}
```

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



Lambda Parameters

- The parameters of a lambda expression have to match the parameters of the method on the single method interface

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

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

- If the parameter types can be inferred, you can omit them.

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

Lambda Parameters

- Zero Parameters

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

- One Parameter

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

- Multiple Parameters

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

- Parameter Types

```
(Car car) -> System.out.println("The car is: " + car.getName());
```

Lambda Function Body

- The body of a lambda expression is specified to the right of the -> in the lambda declaration

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

- If your lambda expression needs to consist of multiple lines, you can enclose the lambda function body inside the { } bracket

```
(oldState, newState) -> {
  System.out.println("Old state: " + oldState);
  System.out.println("New state: " + newState);
 }
```

Lambda Expression Examples

Use case	Examples of lambdas
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`

Consuming from an object	`(Apple a) -> {`
	`  System.out.println(a.getWeight());`
	`}`

Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`
Compare two objects	`(Apple a1, Apple a2) ->` `a1.getWeight().compareTo(a2.getWeight())`

Modern Java in Action: Lambda, streams, functional and reactive programming, 2018

Returning a Value

- You can return values from Java lambda expressions
- You just add a return statement to the lambda function body

```
(param) -> {
  System.out.println("param: " + param);
  return "return value";
}
```

- In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way

```
(a1, a2) -> { return a1 > a2; }

(a1, a2) -> a1 > a2
```

Variable Capture

- A Java lambda expression is capable of accessing variables declared outside the lambda function

- Java lambdas can capture the following types of variables:
  - Local variables
  - Instance variables
  - Static variables

Local Variable Capture

- A Java lambda can capture the value of a local variable declared outside the lambda body

```
public interface MyFactory {
  public String create(String message);
}
```

```
public class LocalVariableCapture {
  public static void main(String[] args) {
      String greeting = "Hello";
      MyFactory myFactory = (message) -> {
          return greeting + ":" + message;
      };
      System.out.println(myFactory.create("Java Lambda"));
  }
}
```

- This is possible if, and only if, **the variable being references is "effectively final"**, meaning it does not change its value after being assigned

Instance Variable Capture

- A lambda expression can also capture an instance variable in the object that creates the lambda

```
public class EventConsumerImpl {
   private String name = "MyConsumer";
   public void attach(MyEventProducer eventProducer) {
       eventProducer.listen( e -> {
           System.out.println(this.name);
       });
   }
}
```

Static Variable Capture

- A Java lambda expression can also capture static variables

```
public class EventConsumerImpl {
  private static String someStaticVar = "Some text";
  public void attach(MyEventProducer eventProducer) {
      eventProducer.listen(e -> {
          System.out.println(someStaticVar);
      });
  }
}
```

Function<T, R>

- Function interface takes an object of generic type T as input and returns an object of generic type R.

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

Function<T, R>

- You might use this interface when you need to define a lambda that maps information from an input object to an output

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

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
public class FunctionalIntefaceExample {
  public static void main(String[] args) {
      List<Integer> l = map(
              Arrays.asList("lambdas", "in", "action"), (String s) -> s.length() );
      System.out.println(l); // [7, 2, 6]
  }
  public static <T, R> List<R> map(List<T> list, Function<T, R> f) {
      List<R> result = new ArrayList<>();
      for ( T t: list ) result.add(f.apply(t));
      return result;
  }
}
```

Method References

- In the case where all your lambda expression does is to call another method with the parameters passed to the lambda, the Java lambda implementation provides a shorter way to express the method call

```
interface Print {
  void execute(String msg);
}
public class MRExample1 {
  private static void doSomething(String value, Print print) {
      print.execute(value);
  }
  public static void main(String[] args) {
      doSomething("Hello", (String msg) -> System.out.println(msg));
      doSomething("Java", System.out::println);
  }
}
```

Method References

```
public class FunctionalIntefaceExample {
  public static void main(String[] args) {
     List<Integer> l = map(
           Arrays.asList("lambdas", "in", "action"), (String s) -> s.length() );
     System.out.println(l); // [7, 2, 6]
  }
}
```

```
public class FunctionalIntefaceExample {
  public static void main(String[] args) {
     List<Integer> l = map(
           Arrays.asList("lambdas", "in", "action"), String::length );
     System.out.println(l); // [7, 2, 6]
  }
}
```

Method References

- Method references can be seen as shorthand for lambdas calling only a specific method.

Lambda	Method reference equivalent
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`
`(String s) -> this.isValidName(s)`	`this::isValidName`

```
public class MRExample2 {
  public static boolean isValidName(String s) {
     return s.length() > 4;
  }
  public static void main(String... args) {
     List<String> names = Arrays.asList("green", "blue", "red");
     List<String> validNames1 = filter(names, (String s) -> isValidName(s));
     System.out.println(validNames1);
     List<String> validNames2 = filter(names, MRExample::isValidName);
     System.out.println(validNames2);
  }

  public static <T> List<T> filter(List<T> list, Predicate<T> p) {
     List<T> result = new ArrayList<>();
     for ( T e: list ) {
        if ( p.test(e) ) {
           result.add(e);
        }
     }
     return result;
  }
}
```

Q&A