Lambda Expressions and Functional Interfaces (Java 8)

Table of Contents

Lambda Expressions

Introduction

- Allowes behavioural parameterization:
  - Flexible
  - Concise
- A clear and concise way to represent one method interface using an expression.

Example lambda usage

```
Repository<User> userRepo = new UserRepo<User>();
List<User> admins = someRepo.find( user -> user.isAdmin() );
```

Syntax

- (argument list) -> {body}
- Syntactical sugar for a method call.
- argument list:
  - Declaring the types of the arguments is optional
  - Using parentheses around the argments is optional if there is only one argument
- body:
  - a single expression or
  - a statement block.
  - In the expression form, the body is evaluated and returned.
  - In the block form, the body is evaluated like a method body and a return statement returns control to the caller of the anonymous method.

    - Using curly braces is optional (unless you need multiple statements).
    - The “return” keyword is optional if you have a single expression that returns a value.

Lambda syntax examples

```
// Basic sytanx:
(params) -> {body}

// Examples:
(int x, int y) -> x + y
() -> 42
(String s) -> { System.out.println(s); }

() -> System.out.println(this)
(String str) -> System.out.println(str)
str -> System.out.println(str)
(String s1, String s2) -> { return s2.length() - s1.length(); }
(s1, s2) -> s2.length() - s1.length()
```

Context capture

- Lambda expressions are compiled to anonymous inner classes.
- They can capture context just like with anonymous inner classes.
    - Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be effectively final (§4.12.4), or a compile-time error occurs where the use is attempted.
    - Any local variable used but not declared in a lambda body must be definitely assigned (§16 (Definite Assignment)) before the lambda body, or a compile-time error occurs.
    - Similar rules on variable use apply in the body of an inner class (§8.1.3). The restriction to effectively final variables prohibits access to dynamically-changing local variables, whose capture would likely introduce concurrency problems. Compared to the final restriction, it reduces the clerical burden on programmers.
    - The restriction to effectively final variables includes standard loop variables, but not enhanced-for loop variables, which are treated as distinct for each iteration of the loop (§14.14.2).
    - They do not have shadowing issues.
    - Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are in the enclosing environment.

Lambda syntax examples

```
public class LambdaScopeExample {

 private String string = "LambdaScopeExample level string";

 private class Inner{

  private String string = "OuterClass string";

  void printString( String externalString ){

   String string = "Lamdba string";

   Consumer<String> consumer = ( stringToPrint ) -> {
    System.out.println( "externalString: " + externalString );
    System.out.println( "stringToPrint: " + stringToPrint );
    System.out.println( "string: " + string );
    System.out.println( "this.string: " + this.string );
    System.out.println( "Inner.this.string: " + Inner.this.string );
    System.out.println( "LambdaScopeExample.this.string: " + LambdaScopeExample.this.string );
   };
   consumer.accept( externalString );
  }
 }

 // Output:
 // externalString: External string
 // stringToPrint: External string
 // string: Lamdba string
 // this.string: OuterClass string
 // Inner.this.string: OuterClass string
 // LambdaScopeExample.this.string: LambdaScopeExample level string

 public static void main(String[] args) {
  LambdaScopeExample example = new LambdaScopeExample();
  Inner outerClass = example.new Inner();
  outerClass.printString( "External string" );
 }
}
```

Target Typing

The main question:

- How do you determine the type of a lambda expression?

Lambda syntax examples

```
void broadcast( Supplier<String> msgProducer ){
 String msg = msgProducer.get();
 System.out.println( msg );
}

void broadcastHello(){
 broadcast( () ->"hello!" ); // How do we know that () -> "hello!" is Supplier<String>?
}
```

Target typing resolution

- The data type that these methods expect is called the target type.
- To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:
  - Variable declarations
  - Assignments
  - Return statements
  - Array initializers
  - Method or constructor arguments
  - Lambda expression bodies
  - Conditional expressions, `?:`
  - Cast expressions

Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution an d type argument inference.

Consider the following two functional interfaces (`java.lang.Runnable` and `java.util.concurrent.Callable<V>`):

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

public interface Callable<V> {
    V call();
}
```

The method `Runnable.run` does not return a value, whereas `Callable<V>.call` does.

Suppose that you have overloaded the method `invoke` as follows (see Defining Methods for more information about overloading methods):

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Which method will be invoked in the following statement?

```
String s = invoke(() -> "done");
```

The method `invoke(Callable<T>)` will be invoked because that method returns a value; the method `invoke(Runnable)` does not. In this case, the type of the lambda expression `() -> "done"` is `Callable<T>`.

Serialization

You can serialize a lambda expression if its target type and its captured arguments are serializable. However, like inner classes, the serialization of lambda expressions is strongly discouraged.

Functional Interface

Introduction

- A functional interface is an interface that only specifies one abstract method.
- Example functional interfaces from the JDK:

Example functional interfaces

```
// in java.lang package
interface Runnable { void run(); }

// in java.util package
interface Comparator<T> { boolean compare(T x, T y); }

// java.awt.event package:
interface ActionListener { void actionPerformed(ActionEvent e); }

// java.io package
interface FileFilter { boolean accept(File pathName); }
```

- Functional interfaces sometimes called Single Abstract Method (SAM)
- Functional interfaces can take generic parameters.
- Functional interfaces provide target types for lambda expressions and method references.
- They often represent abstract concepts. E.g. consumer, producer etc...
- It is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object".
- When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...",
  this is understood to mean a non-null reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.

@FunctionalInterface

- The @FunctionalInterface is used to indicate that an interface declaration is intended to be a functional interface
- See: https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html
- Abstract method:
  - A functional interface must have one abstract method.
  - A functional interface can have only one abstract method.
  - It can have any number of default method.
  - If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count
- Instances of functional interfaces can be created with
  - lambda expressions
  - method references
  - constructor references

Functional interface declaration

```
@FunctionalInterface
interface Hello {
 void sayHello(String name);
}
```

Functional interface creation with lambda expression

```
Hello hello = name -> System.out.println( "Hello " + name );
hello.sayHello( "Johny" );
```

The java.util.function package

- https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html#package.description
- The java.util.function package defines generic purpose functional interfaces.

Naming conventions

The functional interfaces in this package follow an extensible naming convention, as follows:

Function:

- Accept some parameters and return an object .
- From T to R

Examples:

- Function<T,R>
    - Represents a function that accepts one argument and produces a result.
- BiFunction<T,U,R>
    - Represents a function that accepts two arguments and produces a result.

java.util.function.BiFunction example

```
BiFunction<Integer, Integer, Integer> add = (op1,op2) -> op1 + op2;
Integer result = add.apply(5,4);
```

Consumer

- Accept some parameters and return nothing.
- From T to Void.

Examples:

- Consumer<T>
    - Represents an operation that accepts a single input argument and returns no result.
- BiConsumer<T,U>
    - Represents an operation that accepts two input arguments and returns no result.

Supplier

- Accept no parameters and return an object.

- Nillary function to R.

Examples:

- Supplier<T>
  - Represents a supplier of results.
- IntSupplier
  - Represents a supplier of `int`-valued results.

java.util.function.Consumer and java.util.Supplier example

```
Supplier<Date> currentDateProducer = () -> new Date();
Consumer<Date> datePrinter = ( date ) -> { System.out.println(date); };
Date currentDate = currentDateProducer.get();
datePrinter.accept( currentDate );
```

Predicate

- Accept some parameters and return with boolean.
- From T to boolean

Examples:

- Predicate<T>
  - Represents a predicate (boolean-valued function) of one argument.

java.util.function.Predicate example

```
Predicate<Car> isRedCar = (car) -> { return car.isRed(); };
List<Car> cars = new ArrayList<>();
@SuppressWarnings("unused") boolean allRed = cars.stream().allMatch( isRedCar );
```

Arity and naming conventions

- Basic shapes:
  - `Function` (unary function from `T` to `R`)
  - `Consumer` (unary function from `T` to `void`)
  - `Predicate` (unary function from `T` to `boolean`)
  - `Supplier` (nilary function to `R`)
- Derived shapes extend the basic shapes with arity prefix:
  - `BiFunction`(binary function from `T` and `U` to `R`
  - `etc...`
- `There are additional derived shapes that extend the basic functionality:`
  - `UnaryOperator` (extends `Function`)
  - `BinaryOperator` (extends `BiFunction`).
  - etc...
- Type parameters of functional interaces can be specialized to primitives with additional type prefixes:
  - `ToIntFunction`
  - `DoubleConsumer`
  - etc...

Function chaining

- The functional interfaces in the java.util.function package can be chained.
- They provide an andThen() default method for specifying the next operation.

Function chaining

```
private BiFunction<Integer,Integer,Integer> add(){
 return (op1,op2) -> {return op1 + op2;};
}

private Function<Integer,Integer> minus5(){
 return (op1) -> op1-5;
}

public void functionChaining() {
 add().andThen( minus5() ).apply(5, 6);
}
```

Method reference

- Method references allowes us to use existing methods in lamdba expressions.

Kinds of method references

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Reference to static method

```
Car comparatorWithStaticMethod() {
 return getTheBetterCar( new Car(),
       new Car(),
       MethodReferenceExamples::compareByColor );
}

static int compareByColor( Car car1, Car car2 ) {
 return car1.getColor().compareTo( car2.getColor() );
}
```

Reference to an instance mehod of a particular object

```
Car comparatorWithInstanceMethod() {
 return getTheBetterCar( new Car(),
         new Car(),
         this::compareByPrice );
}

int compareByPrice( Car car1, Car car2 ) {
 return car1.getPrice().compareTo( car2.getPrice() );
}
```

Reference to a constructor

```
<T, SRC extends Collection<T>, DEST extends Collection<T>>
 DEST copyElements( SRC source, Supplier<DEST> destinationFactory)
{
 DEST destination = destinationFactory.get();
 for ( T t : source ){
  destination.add( t );
 }
 return destination;
}

void useCopyWithLamdba(){
 copyElements( new ArrayList<Car>(), () -> new ArrayList<Car>() );
}

void useCopyWithConstructorReference(){
 copyElements( new ArrayList<Car>(), ArrayList::new );
}
```