Intro to Java (Page 6)

JUMP June 2019

Draft: 07/16/2019

Table of Contents

Lambdas

1. Intro
2. Block Lambda Expressions
3. Generic Functional Interfaces
4. Pass a Lambda Expression as an Argument
5. Lambda Expressions and Variable Capture
6. Throw an Exception from within a Lambda Expression
7. Lambda expression using a parameter that is an array
8. Method References
9. Constructors References
10. Predefined Functional Interfaces

Lambdas - Intro

- In computer programming, an anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier
- A “lambda” is a block of code that can be passed as an argument to a function call.
- JDK 8 introduces lambda expression
- Lambda is a new syntax in Java

Lambdas - Intro

- Lambda reshapes the Java language
- Lambda is the foundation for other new key features in Java (default method, method reference, etc.)
- Other programming languages have added lambda expressions such as C#, C++, Go, .NET Core, etc.
- JavaScript arrow function expression are also in many ways similar to lambda expression

Lambdas - Intro

Some of the Benefits of lambda expressions in Java

- Conciseness
- Reduction in code bloat
- Readability
- Encouragement of functional programming
- Code reuse
- Enhanced iterative syntax
- Simplified variable scope
- Less boilerplate code
- JAR file size reductions
- Parallel processing opportunities

Lambdas - Intro

Definition of:

1. The lambda expression
2. The functional interface

Lambdas - Intro

The lambda expression

1. an anonymous (unnamed) function which is not executed on its own, instead it is used to implement a method defined by a functional interface.
2. It results in a form of an anonymous class
3. Lambdas are commonly referred to as closures

Lambdas - Intro

The functional interface

1. An interface that contains only one abstract method
2. This method should be specific towards the intended purpose of the interface
3. Should represent a single action
4. Example: the Runnable interface is a functional interface as it only defines a single method run(). This implies that “run()” defines the action of Runnable.

Lambdas - Intro

The functional interface

1. A functional interface defines the ‘target type’ of a lambda expression.
2. A lambda expression can be used only in a context in which a target type is specified.
3. A functional interface is sometimes referred to as a SAM type (Single Abstract Method)

Lambdas - Intro

Lambda Expression Fundamentals

1. New syntax and Operator
2. Lambda operator or arrow operator “->”

   This operator divides a lambda expression in 2 parts

   - **Left side** specifies any parameters required by the lambda expression
   - **Right side** is the lambda body, this specifies the action(s) of the lambda expression
     - 2 types of lambda bodies
       - Single expression
       - Block of code

Lambdas - Intro

Lambda Expression Fundamentals (examples)

1. **Simplest expression**

```
() -> 98.6
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 98.6.

Note: The return type is inferred to be a double.

Non-Lambda equivalent could be as follows (with name)

```
double myMeth() { return 98.6; }
```

Lambdas - Intro

Lambda Expression Fundamentals (examples)

1. **another expression**

```
() -> Math.random() * 100
```

This lambda expression takes no parameters, thus the parameter list is empty. But it does an operation where it takes a random value and multiplies it by 100.

Lambdas - Intro

Lambda Expression Fundamentals (examples)

1. **another expression taking a single parameter**

```
(n) -> 1.0 / n
```

This lambda expression takes one single parameter, it returns the reciprocal values of parameter n, thus 4.0 would return 0.25;

Lambdas - Intro

Lambda Expression Fundamentals (examples)

1. **Expressions can return any data type**

```
(n) -> (n % 2)==0
```

What is interesting here is that the expression will return a boolean value of true or false based on n being an even or odd number.

Lambdas - Intro

Lambda Expression Fundamentals (examples)

1. **Parentheses are not required for single parameter lambda expression**

```
(n) -> (n % 2) == 0
```

and

```
n -> (n % 2) == 0
```

Are identical, valid lambda expressions

For the sake of consistency and clarity, it is best to use parentheses, in this training, we will use them.

Lambdas - Intro

Functional Interfaces

- A functional interface specifies only one abstract method
- Not all interface methods are abstract
- Since JDK8 we can not have an interface with one or more default methods
- Default methods are NOT abstract, neither are ‘static’ interface methods.
- Thus, an interface method is abstract ONLY if it does not specify an implementation

Lambdas - Intro

Functional Interfaces

- A functional interface can only have ONE SINGLE abstract method
- It can have many default and/or static methods
- Note: the keyword modifier ‘abstract’ is not required when defining a functional interface as it is implicit.

See live code in STS “FunctionalInterfaceDemo”

Lambdas - Intro

Functional Interfaces

- Interfaces are reusable, therefore we can create various forms of lambda expressions and keep the same interface

See live code in STS “FunctionalInterfaceDemo2”

Lambdas - Intro

Functional Interfaces

- All of our previous examples were using primitive types, but lambdas can easily use any type, there are no restrictions
- The following example is a lambda that determines if one string is contained within another.

See live code in STS “FunctionalInterfaceDemo3”

Lambdas - Intro

Legal vs Not Legal Lambda Syntax

When dealing with 2 or more parameters, you must explicitly declare them.

THIS IS LEGAL

```
(int n, int d) -> (n % d) == 0
```

THESE ARE NOT LEGAL

```
(int n, d) -> (n % d) == 0
(n, int d) -> (n % d) == 0
```

Lambdas - Intro

Recap of implementing a lambda

- Define an interface
- Define 1 single abstract method in an interface (the abstract modifier keyword is implicit)
- Use the interface and create the lambda expression
- Use the method to execute the lambda expression

Lambdas - Block Lambda Expressions

1. Everything covered so far has been lambdas with single statement expression.
2. Block lambdas are the natural progression in the creation of functions requiring multiple statements. We call that “block body”.
3. The key difference between a regular method and a lambda is that you must always explicitly return a value.

See live code in STS “BlockLambdaDemo”

Lambdas – Generic Functional Interfaces

1. A lambda expression, itself, cannot specify type parameters, this implies that a lambda expression cannot be generic.

2. The functional interface associated with a lambda expression can be generic.

3. The target type of the lambda expression is determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

See live code in STS "GenericFunctionalLambdaInterface"

Lambdas – Pass a Lambda Expression as an Argument

1. A lambda expression can be used in any context that provides a target type.
2. The target contexts used by the preceding examples are assignment and initialization.
3. Another one is when a lambda expression is passed as an argument.
4. In fact, passing a lambda expression as an argument is a common use of lambdas.
5. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method.
6. This greatly enhances the expressive power of Java.

See live code in STS "LambdaExpressionAsArgument"

Lambdas – Lambda Expressions and Variable Capture

- Variables defined by an enclosing scope of a lambda expression are accessible within the lambda expression.
- A lambda expression can use an instance variable or static variable defined by its enclosing class.
- A lambda expression also has access to "this" (both explicitly and implicitly), which refers to the invoking instance of the lambda expression’s enclosing class.
- A lambda expression can obtain or set the value of an instance variable or static variable and call a method defined by its enclosing class.
- When a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a variable capture.

Lambdas – Lambda Expressions and Variable Capture

- Variable capture is when a lambda expression can only use local variables that are effectively final.
- An effectively final variable is one whose value does not change after it is first assigned.
- There is no need to explicitly declare such a variable as final, although doing so would not be an error.
- (The "this" parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a "this" of their own.)
- It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression as it would be rendering it illegal for capture.

See live code in STS “LambdaExpressionVariableCapture”

Lambdas – Throw an Exception from within a Lambda Expression

- A lambda expression can throw an exception.
- If it throws a checked exception, however, then that exception must be compatible with the exception(s) listed in the throws clause of the abstract method in the functional interface.
- For example, if a lambda expression throws an IOException, then the abstract method in the functional interface must list IOException in a throws clause.

```
import java.io.*;

interface MyIOAction {
  boolean ioAction(Reader rdr) throws IOException;
}

class LambdaExceptionDemo {
  public static void main(String args[]) {
    double[] values = { 1.0, 2.0, 3.0, 4.0 };
    // This block lambda could throw an IOException.
    // Thus, IOException must be specified in a throws
    // clause of ioAction() in MyIOAction.
    MyIOAction myIO = (rdr) -> {
      int ch = rdr.read(); // could throw IOException
      // ...
      return true;
    };
  }
}
```

Lambdas – Lambda expression using a parameter that is an array

- When the type of the parameter is inferred, the parameter to the lambda expression is not specified using the normal array syntax.
- Instead the parameter is specified as a simple name, such as n, not as n[ ].
- The type of a lambda expression parameter will be inferred from the target context.
- If the target context requires an array, then the parameter's type will automatically be inferred as an array.
- Here's an example:

```
// A functional interface.
interface MyTransform<T> {
  void transform(T[] a);
}
```

- The parameter to the transform() method is an array of type T.

Lambdas – Lambda expression using a parameter that is an array

- Consider the following lambda expression using MyTransform to convert the elements of an array of Double values into their square roots.

```
MyTransform<Double> sqrts = (v) -> {
  for(int i=0; i < v.length; i++) v[i] = Math.sqrt(v[i]);
};
```

- Here, the type of a in transform( ) is Double[], because Double is specified as the type parameter for MyTransform when sqrts is declared.

- Therefore, the type of v in the lambda expression is inferred as Double[ ].

- It is not necessary (or legal) to specify it as v[ ].

- One last point: It is legal to declare the lambda parameter as Double[ ] v, because doing so explicitly declares the type of the parameter, but doing so gains nothing in this case.

Lambdas – Method References

- There is an important feature related to lambda expressions called the method reference.
- A method reference provides a way to refer to a method without executing it.
- It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface.
- When evaluated, a method reference also creates an instance of a functional interface.
- There are different types of method references.
  - **Method References to static Methods**
  - **Method References to Instance Methods**
  - **Class Name with Method Reference**
  - **Specify a method reference to a generic method**

Lambdas – Method References

Method References to static Methods

- A method reference to a static method is created by specifying the method name preceded by its class name, using this general syntax:
  `ClassName::methodName`
- Notice that the class name is separated from the method name by a double colon. The :: is a new separator that has been added to Java by JDK 8 expressly for this purpose.
- This method reference can be used anywhere in which it is compatible with its target type.

Lambdas – Method References

Method References to static Methods

- The following program demonstrates the static method reference. It does so by first declaring a functional interface called IntPredicate that has a method called test( ).
- This method has an int parameter and returns a boolean result. Thus, it can be used to test an integer value against some condition.
- The program then creates a class called MyIntPredicates, which defines three static methods, with each one checking if a value satisfies some condition.
- The methods are called isPrime( ), isEven( ), and isPositive( ), and each method performs the test indicated by its name.
- Inside MethodRefDemo, a method called numTest( ) is created that has as its first parameter, a reference to IntPredicate.
- Its second parameter specifies the integer being tested.
- Inside main( ), three different tests are performed by calling numTest( ), passing in a method reference to the test to perform.

See live code in STS “LambdaMethodReferenceStaticMethods”

Lambdas – Method References

Method References to Instance Methods

- A reference to an instance method on a specific object is create by this basic syntax:

  `objRef::methodName`

- As you can see, the syntax is similar to that used for a static method, except that an object reference is used instead of a class name.
- Thus, the method referred to by the method reference operates relative to objRef.

Lambdas – Method References

Method References to Instance Methods

- The following program demonstrates what the previous slide explains.
- It uses the same IntPredicate interface and test( ) method as the previous program.
- However, it creates a class called MyIntNum, which stores an int value and defines the method isFactor( ), which determines if the value passed is a factor of the value stored by the MyIntNum instance.
- The main( ) method then creates two MyIntNum instances.
- It then calls numTest( ), passing in a method reference to the isFactor( ) method and the value to be checked.
- In each case, the method reference operates relative to the specific object.

See live code in STS “LambdaMethodReferenceInstanceMethods”

Lambdas – Method References

ClassName with Method Reference

- It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class not just a specified object.
- In this case, you will create a method reference as shown here:

```
ClassName::instanceMethodName
```

- Here, the name of the class is used instead of a specific object, even though an instance method is specified.
- With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter (if any) specified by the method.

Lambdas – Method References

ClassName with Method Reference

- The following example reworks the previous one.
- First, it replaces IntPredicate with the interface MyIntNumPredicate.
- In this case, the first parameter to test( ) is of type MyIntNum.
- It will be used to receive the object being operated upon.
- This allows the program to create a method reference to the instance method isFactor( ) that can be used with any MyIntNum object.

See live code in STS “LambdaMethodClassnameMethodReference”

Lambdas – Method References

Specify a method reference to a generic method

- Because of type inference we don't need to explicitly specify a type argument to a generic method when obtaining its method reference.
- Java includes a syntax to handle those cases in which you do.

```
interface SomeTest<T> {
  boolean test(T n, T m);
}

class MyClass {
  static <T> boolean myGenMeth(T x, T y) {
    boolean result = false;
    // ...
    return result;
  }
}
```

- The following statement is valid:
  `SomeTest<Integer> mRef = MyClass::<Integer>myGenMeth;`
- Here, the type argument for the generic method myGenMeth is explicitly specified.
- The type argument occurs after the ::.
- This syntax can be generalized:
- When a generic method is specified as a method reference, its type argument comes after the :: and before the method name.
- In cases in which a generic class is specified, the type argument follows the class name and precedes the ::.

Lambdas – Constructors References

- Similar to the way that you can create references to methods, you can also create references to constructors.
- Here is the general form of the syntax that you will use:

  `classname::new`

- This reference can be assigned to any functional interface reference that defines a method compatible with the constructor.

See live code in STS "LambdaExpressionConstructorReference"

Lambdas – Constructors References

Declare a constructor reference that creates an array

- To create a constructor reference for an array, use this construct:

```
type[]::new
```

- Here, type specifies the type of object being created. For example, assuming the form of MyClass shown in the preceding example and given the MyClassArrayCreator interface shown here:

```
interface MyClassArrayCreator {
 MyClass[] func(int n);
}
```

- the following creates an array of MyClass objects and gives each element an initial value:

```
MyClassArrayCreator mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(3);
for(int i=0; i < 3; i++) {
 a[i] = new MyClass(i);
}
```

The call to func(3) causes a three-element array to be created. This example can be generalized.
Any functional interface that will be used to create an array must contain a method that takes a single int parameter and returns a reference to the array of the specified size.

As a point of interest, you can create a generic functional interface that can be used with other types of classes, as shown here:

```
interface MyArrayCreator<T> {
 T[] func(int n);
}
```

For example, you could create an array of five Thread objects like this:

```
MyArrayCreator<Thread> mcArrayCons = Thread[]::new;
Thread[] thrds = mcArrayCons.func(5);
```

Lambdas - Predefined Functional Interfaces

JDK 8 adds a new package called java.util.function that provides several predefined functional interfaces.

See live code in STS “BuiltInPredicateFunctionalInterfaceDemo”

Link to a list of predefined functional interfaces:
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html