Lambdas in Java 8

Start programming in a more functional style

Background

Who am I?

- Tobias Coetzee
- I'm a Technical Lead at BBD
- I present the Java Expert Level Certifications at BBD (EJB, JPA, etc.)
- I'm currently a banker
- Part of the JoziJug organising committee
- I like to run
- This presentation is based on a Simon Ritter course
- @tobiascode
- https://github.com/Jozi-JUG/lamdatraining

ASK QUESTIONS, SWAG FOR EVERYONE!!

Outcomes

What are the take away's?

- Why should we care about lambdas
- Apply lambdas to everyday problems
- Determine when to apply lambdas (and when not to!)
- How does the lambda syntax work?
- What are functional interfaces?
- Convert anonymous classes to lambda expressions
- Debug lambda expressions

Why Lambdas?

Why lambdas?

Why care about concurrency in Java?

- CPU's are getting much faster, but we are getting more cores to use.
- We need to parallelise our processing.
- The functional programming paradigm lends itself better to parallel processing as there is no external state.

Concurrency history in Java

- Java 1 had threads.
- Java 5 had java.util.concurrent (BlockingQueue, Executers, etc.).
- Java 7 had Fork/Join framework.
- Java 8 gives us Lambdas.

Demo: Intro to functional programming in Java 8

When do we use lambdas?

Usages

- Inject functionality into methods, the same way we inject values into methods.
- Look where methods are very similar except for 1 or 2 lines of code.
- Look at streams for clues, filtering, mapping, finding, matching, etc.
- Enhance library methods, look at new Collection methods.
- Small one line functions.
- To shorten your code, remove inner classes and anonymous classes.
- If you want to use streams.

When don’t we use lambdas?

Non-Usages

- Lambdas were introduced into Java for 2 reasons, to allow programming in a functional paradigm and to make code shorter and more concise.
- If it isn’t used for one or both these reasons, rethink the usage.
- Avoid huge lambdas, e.g. 20 lines of code.
- Because it is cool!

Lambda Syntax

Lambda Syntax

lambda operator

Basic Structure

```
(parameters) -> {lambda body}
```

Rules

- Lambdas can be single or multi-line
- Single line Lambdas
  - Do not need curly braces
  - Do not need an explicit return statement
- Lambdas with a single parameter do not need braces
- Lambdas with no parameters must have empty braces
- Body of the Lambda may throw exceptions

Lambda Syntax

Examples

- `() -> System.out.println("Hello Lambda")`
- `x -> x + 10`
- `(int x, int y) -> { return x + y; }`
- `(String x, String y) -> x.length() – y.length()`

-
  ```
  (String x) -> {
                 listA.add(x);
                 listB.remove(x);
                 return listB.size();
        }
  ```
- `(x, y) -> x.length() – y.length()`

Exercise: Lambda Functions

Functional Interfaces

Functional Interfaces

Lambda Expression Types

- A lambda expression is an anonymous function, it is not associated with a class.
- A lambda expression can be used wherever the type is a functional interface.
  - This is a single abstract method type.
  - The lambda expression provides the implementation of the abstract method.

Functional Interfaces

Definition

- An interface that has only one abstract method.
- Before Java 8 this was obvious, only one method.
- Java 8 introduced **default** methods
  - Multiple inheritance of behaviour for Java
- Java 8 also now allows static methods in interfaces
- **@FunctionalInterface** annotation
- Previously known as **SAM**'s, **S**ingle **A**bstract **M**ethods
- java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator, java.util.concurrent.Callable

True or False: Functional Interfaces

Functional Interfaces

Example Uses Of Lambda Expressions

- Method parameter

```
consumeStudent(s ->
System.out.println(s.getGrade()));
```

- Variable assignment

```
Consumer<Student> c = s ->
System.out.println(s.getGrade())
```

Functional Interfaces

java.util.function Package

- Well defined set of general purpose functional interfaces.
- All have only one abstract method.
- Lambda expressions can be used wherever these types are referenced.
- Used extensively in the Java class libraries.
- Especially with the Stream API.
- Includes several static methods to help with composing/chaining.
- Variations on number of arguments and specific for primitive types.

Functional Interfaces

Supplied Interfaces

Interface	Description
Consumer<T>	Operation that takes a single value and returns no result.
Supplier	A supplier of results.
Function<T,R>	A function that accepts one argument and returns a result.
UnaryOperator<T>	Specialised form of function, takes a single argument and returns a result of the same type.
BinaryOperator<T>	Specialised form of function, takes two arguments and returns a result of the same type
Predicate	A boolean valued function that takes one argument.

Demo: General Functional Interfaces

Method and Constructor References

Method&Constructor References

Usage

- Method references let us reuse a method as a lambda expression.

```
FileFilter x = File f -> f.canRead();
      FileFilter x = File::canRead;
```

- Format: **`target_reference::method_name`**
- Three kinds of method reference
  - Static method
  - Instance method of an arbitrary type
  - Instance method of an existing object

Method&Constructor References

Rules For Construction

```
(args) -> ClassName.staticMethod(args)
          ClassName::staticMethod

(arg0, rest) -> arg0.instanceMethod(rest)
          ClassName::instanceMethod

(args) -> expr.instanceMethod(args)
          expr::instanceMethod
```

Method&Constructor References

Constructor Reference

- Same concept as a method reference.

```
Factory<List<String>> f = () -> return
          new ArrayList<String>();

     Factory<List<String>> f =
      ArrayList<String>::new;
```

Referencing External Variables

Referencing External Variables

Effectively Final

- Lambda expressions can refer to **effectively final** local variables from the surrounding scope.

```
class DataProcessor {
  private int currentValue;

  public void process() {
    DataSet myData = myFactory.getDataSet();
    dataSet.forEach(d -> d.use(currentValue++));
  }
}
```

Referencing External Variables

Effectively Final

- Effectively final: A variable that meets the requirements for final variables (i.e., assigned once), even if not explicitly declared final.
- Closures on values, not variables.
- Closures allow you to use variables defined outside the scope of a function.
- For heap-based objects, e.g. arrays, modify values just not reference.

Referencing External Variables

'this' in a Lambda

- 'this' refers to the enclosing object, not the lambda itself.
- Remember the Lambda is an anonymous function.
- It is not associated with a class, therefore there can be no 'this' for a lambda.

```
class DataProcessor {
  private int currentValue;

  public void process() {
    DataSet myData = myFactory.getDataSet();
    dataSet.forEach(d -> d.use(this.currentValue++));
  }
}
```

Debugging Lambda Methods

Debugging Lambdas

Problem

- Lambda expressions do not compile to equivalent inner class.
- Compiled to **invokedynamic** call.
- Implementation decided at runtime.
- Better chance of optimisation, makes debugging harder.

Debugging Lambdas

Solution

- IDE should support debugging.
- If not, extract the code from a lambda expression into a separate method.
- Replace the lambda with a method reference for the new method.
- Implement the functional interface and pass a reference of the class.

Demo: Debugging a lambda function

New Lambda Methods in Java 8

New Lambda Methods

Why New Methods?

- Look at existing interfaces for new default or static methods, they are normally the new methods that accept lambdas.
- Use the new methods in Java 8 to eliminate the frequent need for loops.
- Fits in with using the functional paradigm.
- Remember that a lambda provides behaviour, not a value.
- Very useful for conditional uses of data.

Functional Interfaces

Some examples

Interface	Description
`Iterable.forEach(Consumer c)`	`myList.forEach(System.out::println);`
`Collection.removeIf(Predicate p)`	`myList.removeIf(s -> s.length() == 0);`
`List.replaceAll(UnaryOperator o)`	`myList.replaceAll(String::toUpperCase);`
`List.sort(Comparator c)` *Replaces Collections.sort(List l, Comparator c)*	`myList.sort((x, y) -> x.length() – y.length());`

Exercises

Exercises

How does it work?

- The exercises for this month are all small exercises that you will complete using the things you have learned during this presentation.
- There are 6 exercises to complete.
- For first month focus on using lambda expressions, method references and some of the new methods that have been added to existing classes.
- There is a template source file that you should use to create the answers to the exercises called Exercise_Test.java under the test folder.
- To simplify things the lists and maps you need for the exercises have already been created.
- You just need to focus on the code to solve the exercise.
- The solutions are in the Solution_Test.java file, but try all the exercises first before taking a peak.

Thank you!

JOZI JUG

bbd
combined intelligence