Functional Programming in Java

COMP2209 - Programming III

Dr Julian Rathke
Objects and Functions

- In Java we use objects as a means of aggregating related data as well as functionality for manipulating that data.
- Therefore, if we consider objects that don’t store data per se (i.e. no fields) but simply provide functionality then these begin to look something like functions.
- Reasonable questions to ask in this case then are -
  - What type of objects can be considered as “functions” in Java?
  - Can we do functional programming with them?
  - **Should** we do functional programming with them?
  - What language support should we provide?
- We’ll consider these questions during this lecture and show what has actually been done in this respect in Java 8
Functions as Objects

- We’ve already noted that a function has no state - i.e. no fields to manipulate, but the data related to a function is the code itself.

- So let’s consider the manipulations one does with a function:
  - Application.
  - Errr, that’s it.

- So a function should be an object with just a single method that represents application of the function to an argument.

- Java is a typed language so we would expect to use typed functions similar to Haskell.
  - A Haskell function \( f :: T \rightarrow U \), says if you apply this function to a value of type \( T \) we get a value of type \( U \) in return.
  - Hence, our single method should also have these types.
Example of a “function” object

• Consider an object that would represent a function that takes an Int, doubles it and returns that value.

• In Java we could use

```java
class Double {
    int apply(int x) { return 2*x ; }
}
```

• Any instance of this class would behave much like the Haskell function `double x = 2*x`, but let’s play spot the difference:

- The types in Java need to be explicitly written.
- The name “apply” is not needed in Haskell as it is standard ($$\)$).
- We don’t use return in Haskell (for pure values).
Functional Interfaces

• In simplest form we can define a functional interface in Java to be an interface with a single abstract method.
  • These used to be called SAM Types
• Remember that all objects in Java inherit several methods from Object
  • These don’t count.
  • Nor do static or default methods.
• Good practice would have you annotate the interface using the @FunctionalInterface annotation
  • It is not strictly necessary to do so as being a functional interface is a structural property checkable by the compiler.
Examples of functional interfaces

• We have loads of existing functional interfaces in Java!
• You’ll notice a few listeners in there.
• These are one of the chief use cases for functions in Java.
• Let’s look at a typical ActionListener and see how functions help
ActionListener as a Function

• Consider this typical ActionListener Java code snippet

```java
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anon Class");
    }
});
```

This is very bulky syntactically, and uses an anonymous inner class.

Remember the scoping rules for inner classes? You can access `final` variables from the containing class. Interplay with inherited members can cause naming confusion with enclosing class. These aren’t great.
Re-imagining the listener

• Imagine if we could rewrite that ActionListener function in a much neater syntax, lambda expressions say.
• We know from the interface the type and name of the single method so we don’t need to write that down again.
• Strictly speaking, we just need to provide the name of the bound variable and the actual code for the function.

```java
JButton testButton = new JButton("Test Button");
testButton.addActionListener(
    ae -> System.out.println("Click Detected by Anon Class")
);
```

The rather brilliant thing is that since Java 8 you can do exactly this!
Lambda Syntax in Java

• For **any** functional interface we can now use lambda notation in Java.

• A lambda expression implements the functional interface for which it is used by providing code for its single listed method.

• The syntax is `Args -> Body` where `Args` is an argument list

• An argument list is a single variable name, or a comma-separated list of (typed) variables names

  \[(T_1 \ x_1, \ T_2 \ x_2, \ ... , \ T_N \ x_N)\]

• You can omit the types! In Java! And you can have an empty list () for when there are no arguments.

• The Body is any Java expression, Statement or Block - you can omit writing “return” to return a value from an Expression.

• Obviously the argument types and return type of the Body must match the usage of the lambda expression in context.
Example Lambda Expressions

Single argument - no parentheses

s -> s.length()

No arguments

() -> 42

Two arguments with explicit types

(int x, int y) -> x+y

(x, y, z) ->

{ if (true) return x;
  else {
    int result = y;
    for (int i = 1; i < z; i++) result *= i;
    return result;
  }
}
Here is a code snippet that I used to safely initialise a GUI:

```java
public static void main(String[] args) {
    DDFrame ddf = new DDFrame("Digital Doilies");
    SwingUtilities.invokeLater(() -> ddf.init());
}
```

The method `invokeLater` is expecting a `Runnable` - this is a functional interface with sole method `run`. This lambda expression implements `Runnable` and overrides the `run` method to have method body `ddf.init()`.

This syntax replaces a lot of the tedious boilerplate code and makes Java a whole lot more readable.

But is it doing any more than that? Can we actually do functional programming with it?
General function types

• In order not to drift too far from Java’s design it was decided not to introduce a new function type in Java.
• Functional interfaces play this role to some extent.
• However, if we were to try write a higher-order function like, map, say then what would its type be?
• We can use generics to achieve the polymorphism required but we are still lacking a “function type”

interface Map<T,U> {  
  ArrayList<U> apply ( ??? T → U ??? , ArrayList<T> x)  
}

• However, if we had a pre-defined functional interface called Function<T,U> say, with a fixed name for its single method, apply, say, then this could do the job.
General function types

- In fact in package `java.util.function` we have exactly this.
- This package defines common general purpose functional interfaces (all with generic types).

<table>
<thead>
<tr>
<th>Interface</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function&lt;T,R&gt;</td>
<td>R apply (T x)</td>
</tr>
<tr>
<td>BiFunction&lt;T,U,R&gt;</td>
<td>R apply (T x, U y)</td>
</tr>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>boolean test (T x)</td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>void accept(T x)</td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>T get ()</td>
</tr>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T apply (T x)</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>T apply (T x, T y)</td>
</tr>
</tbody>
</table>

Their intended use is clear from the names of the interface and method.
Example using a general functional interface

```java
... int absApply(Function<Integer, Integer> f, int x) {
    return f.apply(x < 0 ? -x : x)
}

// example use
Function<Integer, Integer> absSqrt = x → Math.sqrt(x);
absApply(absSqrt, -25) // returns 5
absApply(x → 6-x, -4) // returns 2
...
```
Closures and lambda

• Recall the notion of closure in the lambda calculus?
• A closure is a pair of a lambda term and bindings for all of its free variables.
• In Java, inner classes do not form proper closures - this is why there is the restriction to using only final fields from the enclosing class.
• Also, the binding for the keyword this always refers to the inner class instance within the inner class
  • The enclosing this reference must be explicitly addressed.
• Lambda syntax however improves this situation by providing closures for Java.
• Let’s look at this using examples.
Closures by example

```
public class ClosureExample {
    int x = 0;
    Runnable r = new Runnable () {
        public void run() {
            System.out.println("Inner x is "+x);
            System.out.println(this);
        }
    };

    String toString() { return ("Outer x is "+x); }

    public static void main(String... args) {
        new ClosureExample().r.run();
    }
}
```

Won’t compile due to non-final access of x
Closures by example

public class ClosureExample {
    final int x = 0;
    Runnable r = new Runnable () {
        public void run() {
            System.out.println("Inner x is " + x);
            System.out.println(this);
        }
    };

    String toString() { return ("Outer x is " + x); }
    public static void main(String... args) {
        new ClosureExample().r.run();
    }
}
Closures by example

```java
class ClosureExample {
    final int x = 0;
    Runnable r = () -> {
        System.out.println("Inner x is " + x);
        System.out.println(this);
    }

    String toString() { return ("Outer x is " + x); }

    public static void main(String... args) {
        ClosureExample ce1 = new ClosureExample();
        ClosureExample ce2 = new ClosureExample();
        ce1.r.run();
        ce2.x++;
        ce2.r.run();
    }
}
```

prints Inner is 0, Outer is 0
prints Inner is 0, Outer is 1
Closures by example

```java
public class ClosureExample {
    final int x = 0;
    Runnable r = () -> {
        System.out.println("Inner x is "+ x);
        x++;
        System.out.println(this);
    }

    String toString() { return "Outer x is "+ x; }

    public static void main(String... args) {
        new ClosureExample().r.run();
    }
}
```

Variables used in a lambda expression must be arguments, local, or **effectively final** - that is, if not declared final, then not mutated.

Mutation is not allowed on “closure” variables in lambda expressions.
What else?

- So lambda expressions in Java provide the ability to define proper closures as in Haskell.
- The java.util.functions package and its functional interfaces provide the framework of functions and application.
- Generics provide the polymorphism.
- What else are we missing in terms of using Java as a functional programming language?

First-class functions

We can’t quite just pass any old method of the correct type around as if it were a function. It needs to be converted somehow to being the apply method of Function<T,U> type or similar.

This is where method references enter the game.
Method References

- **Method References** provide a means of referencing a given method as if it were a function in Java.

- I’ll show you an example of using a static method reference.

- Remember this example? I can rewrite it more simply

```java
int absApply(Function<Integer,Integer> f , int x){
    return f.apply(x < 0 ? -x : x)
}
// example use
Function<Integer,Integer> absSqrt = x -> Math.sqrt(x) ;
absApply( absSqrt , -25 ) // returns 5
```

```java
int absApply(Function<Integer,Integer> f , int x){
    return f.apply(x < 0 ? -x : x)
}
// example use
absApply( Math::sqrt , -25 ) // returns 5
```
Method Reference Syntax

• The operator :: in Java is an instruction to build a lambda expression for the given method.

• So :: constructs a lambda expression with an appropriate number of arguments for that method
  • e.g. Math::sqrt constructs the expression i → Math.sqrt(i)

• The constructed lambda expression can then be used in a functional interface (as we saw in the example above).

• This is a well-known operation in the lambda-calculus called \( \eta \)-conversion (eta-conversion).

• It essentially says that, for any function \( f \), \( \lambda x \rightarrow f(x) \) is equal to \( f \) itself.

• This seems lovely and straightforward - but of course this is Java so there are some special cases to consider.
Different types of Method Reference

- **Static** \((\text{ClassName}::\text{methodName})\)
  - **Known Instance** \((\text{instanceRef}::\text{methodName})\)
  - **Super** \((\text{super}::\text{methodName})\)
  - **Unknown Instance** \((\text{ClassName}::\text{methodName})\)
  - **Constructor** \((\text{ClassName}::\text{new})\)
  - **Array Constructor** \((\text{TypeName}[] :: \text{new})\)

We’ve seen an example of Static method reference - it \(\eta\)-converts the method as a lambda expression using the parameters of the method as the lambda arguments.

This is also true of the Known Instance, Super, and Constructor method references.

For Unknown Instance method references, we use a lambda expression that also takes as an argument an object of the given ClassName.
Different types of Method Reference

Static

ClassName::staticMethod

Known Instance

obj::method

super

super::method

Constructor

ClassName::new

Unknown Instance

ClassName::method

Finally, Array Constructor method references allow an integer argument to determine array size.

Array Constructor

TypeName[].::new

\((\text{args}) \rightarrow \text{new TypeName}[n]\)
Example of Unknown Instance Method Reference

Suppose we have a class `Person` with a bunch of data such as name, age, height, weight etc with accessor methods for each.

Suppose also that we have an array `ps` of these and wish to list per property, the values for each `Person`.

```java
listAll(Function<Person,R> f, Person[] ps { 
    for(Person p : ps) {
        System.out.println( f.apply(p) )
    }
} 

We can call this using method references:

listAll( Person::getName , ps)
listAll( Person::getHeight , ps) …
```

This is a higher-order function that accepts a Function.
Recursion

- Oh, we haven’t talked about recursively defined functions in Java yet.
- **Anticlimax warning**: they aren’t very good.

```java
class FactNonExample {
    static Function<Integer,Integer> f =
        i → (i==0)? 1 : i * f.apply(i-1)
    int fact(int n) = return (f.apply(n));
}
```

This looks plausible as a recursive definition of factorial. But unfortunately it doesn’t compile as you cannot refer to a variable in its own initialiser.
Recursion

• But we can use a sensible workaround for this:
• Build a general class called Recursive that wraps a function.

```java
class FactExample {
    Recursive<Function<Integer,Integer>> rec = new Recursive<>();
    rec.f = i → (i==0)? 1 : i * rec.f.apply(i-1)
    int fact(int n) = return (rec.f.apply(n));
}
```
YOUR QUESTIONS

Next Lecture:
Functional Iteration and Streaming in Java