Lambda Expressions

Functional Programming

Lambda-calculus

- Alonzo Church, ~1935
- The first “programming language”
- The function $f(x, y) := x+y$:

$$\lambda x.\ \lambda y.\ (x + y)$$

Lambda-calculus Principles

- A language of **functions**

- In the mathematical sense:
  - same input → same output
  - i.e., *stateless*
  - a.k.a. *purity, absence of side-effects, referential transparency*

- No variables, no assignments, no loops!

Functional Languages

- LISP
  - LISt Processor
  - 1958, second programming language after Fortran
    - Scheme
    - Clojure (2007, runs on JVM)
- Haskell (1990)
- ML
  - Meta Language
  - 1973
    - F# (Microsoft .NET, multi-paradigm)
    - OCaml (multi-paradigm)

Functional Parallelism

- *Composition* instead of *communication*:

$$f(g(a), h(b), g(c))$$

- Stateless functions g and h can be evaluated *in any order*
- Even in parallel
- Communication only through return values
- No race conditions, no need to synchronize, etc.

Interfaces Get a Boost

Static Methods

- Interfaces have always had public static final *fields*
- Since Java 8, they can also have public static *methods*
- Utility classes like Math could now be interfaces

Default Methods

- A concrete instance method, which can be overridden
- Basically, a regular instance method

```
interface A {
    default void foo() {
        System.out.println("I have an implementation!");
    }
}
```

Multiple Inheritance

- Java now supports *multiple inheritance of implementation!*

```
interface A {
   default void foo() {
      System.out.println("I have an implementation!");
   }
}
interface B {
   default void foo() {
      System.out.println("I have another implementation!");
   }
}
class X implements A, B { }
```

Ambiguous: compilation error.
X must override foo

Default Methods

- Interfaces used to provide *signatures*
- They now also provide *behavior*

So, what’s the difference with abstract classes?

Interfaces vs Abstract Classes

- Interfaces are still *stateless* (**no instance fields**)
- Interfaces provide behavior, not state
- Abstract classes provide behavior **and state**

Language Evolution and Backward Compatibility

- You can add static and default methods to an interface *without affecting the implementing classes*
- Many standard interfaces have been enriched with new methods
- Collection, List, Comparator, Iterator, etc.

The Comparator Example

- Comparator contains both static and default methods

- Comparator in Java 7:
  - 1 abstract method (compare)

- Comparator in Java 8:
  - 1 abstract method
  - 9 static methods
  - 7 default methods

The Comparator Example

- A default method in Comparator<T>:

```
default Comparator<T> thenComparing(Comparator<? super T> other)
```

- Composes this comparator with another comparator in *lexicographic* order
  - Given two objects to compare…
  - …first it evaluates `this` comparator…
  - …if this comparator returns 0, it evaluates the `other` comparator

See also...

- Scalable.java
- Interfaces.java

on https://bitbucket.org/mfaella/functionaljava

Some Interfaces are more Functional *than others*

Functional Interfaces (FIs)

- Any interface with *a single abstract method*
- Static and default methods allowed
- Examples:
  - Comparable, Comparator
  - Iterable
  - Runnable

  - *Scalable*

Pure Functional Interfaces

- An FI intended to be implemented by *stateless* classes

- Examples:
  - Runnable: not pure
  - Comparable: not pure
  - Comparator: **pure**

Pure Functional Interfaces

- Pure FIs respect the Functional Programming paradigm
- They play an important role in conjunction with *streams*

The @FunctionalInterface Annotation

- Intended for **pure** FIs
- Compiler checks the “single abstract method” property

- Comparator is annotated with it
- Comparable is *not* annotated with it

FIs in the Java 9 API

- More than 40 pure FIs
- Package java.util.function

Examples

- A function accepting an object

```
public interface Consumer<T> {
    void accept(T t);
}
```

- A function producing an object

```
public interface Supplier<T> {
    T get();
}
```

Lambda Expressions

Introducing Lambda Expressions

- A compact syntax to implement FIs
- An alternative to anonymous classes

```
Comparator<String> byLength =
    (String a, String b) -> {
        return Integer.compare(a.length(), b.length());
    };
```

Lambda Expression Syntax

parameters -> body

parameters:
- (int a, int b)
- (a, b)
- (a)
- ()
- a

body:
- { block }
- expr

See also…

- Lambda1.java

on https://bitbucket.org/mfaella/functionaljava

Typing Lambda Expressions

Type Inference

- Infer a type at compile time
- The compiler filling in a missing type

- Been there since Java 5

Type Inference in Java 5

```
public static <T> T getFirst(T[] array) {
   return array[0];
}

String[] strarray = { "one",  "two", "three" };
String one = getFirst(strarray);
```

- Code does not specify type parameter for "getFirst"
- Still, no cast is needed in assignment
- Type parameter "String" is inferred from the actual parameter "strarray"

Type Inference and Lambdas

- Identify the FI being implemented
- Identify the parameter types (if omitted)

Receiving Contexts

Context must contain enough info to identify the receiving FI

- RHS of assignment

```
Consumer<String> c = lambda
```

- Actual parameter of a method or constructor

```
new Thread(lambda)
```

- Argument of 'return'

```
return lambda
```

- Argument of a cast

```
(Consumer<String>) lambda
```

See also...

- LambdaInference.java

on https://bitbucket.org/mfaella/functionaljava

Capturing Values

Capturing Values

- Lambda expressions can access:
  - **static fields** of any class (trivial)
  - **local variables** of enclosing method (needs *capture*)
  - **instance fields** of enclosing object (needs *capture*)

Capturing Locals

- Lambda expressions can access:
  - **local variables** of enclosing method, provided they are *effectively final*
  - (same rule as anonymous classes)

Implementation:

- They store a **copy of that variable**
- They “capture” that variable
- Every runtime evaluation may or may not generate a new object (see example)

Capturing Fields

- Lambda expressions can access:
  - the **enclosing object *(this)***
  - its **instance fields**

```
class Test {
   public Consumer<String> foo() {
      return (msg -> System.out.println(msg + this));
   }
}
```

The current Test object

Different from anonymous classes!

Capturing Fields

- Lambda expressions can access:
  - **instance fields** of enclosing object

Implementation:

- They store a **reference to the enclosing object**
- They “capture” the current instance: *Instance-capturing lambda expression*
- Similar to capturing the local variable “this”
- Every runtime evaluation generates a new object

Effectively Final Variables

- A variable that is used *as if it was final*

- Not reassigned

Note: it's good practice to declare them *final*

Lambda Expressions vs Anonymous Classes

- Lambdas are more succinct
- Lambdas do not create additional class files
- Not every occurrence of a lambda creates a new object!

On the other hand:

- Anonymous classes can have multiple methods
- Anonymous classes can have state (that is, fields)

See also...

- LambdaImplementation.java
  https://bitbucket.org/mfaella/functionaljava

- Article "Java 8 Lambdas - A Peek Under the Hood", by R.Urma and R.Warburton
  https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood