



Functional features of Java 8

# When?

- Java 7
  - Now, Java 6 put out to pasture Nov '12
- Java 8
  - Currently slated for Summer '13

# Java 7

- Most of what you'll see is **Project Coin** (JSR-334)
- Language features to help cut the cruft
  - Strings in **switch**
  - "Diamond" syntax `<>`
  - Binary integral literals/Underscores in numerics literals
  - Multi-catch Exceptions, precise rethrow, try-with-resources

# Java 8

- Second Part of "Plan B"
- Incorporates two major JSRs
  - Jigsaw (think OSGi)
  - Lambda (what we're interested in)

# Project Lambda

- Two goals
  - Anonymous code block syntax
    - Should be "familiar" syntax
  - Bytecode compatible with legacy Java

# Lambda syntax

```
rowList.map(row -> new Person(row.get("id"));
```

```
rowList.map(  
  row -> {  
    Person p = new Person(row.get("id"));  
    p.setFirstName(row.get("fname"));  
    p.setLastName(row.get("lname"));  
    return p;  
  });
```



Type inference

# Contrast with...

```
List<Person> l = Mapper.map(rowList, new RowMapper<Person>() {  
    public Person mapRow(Map<String, Object> row) {  
        Person p = new Person(row.get("id"));  
        p.setFirstName(row.get("fname"));  
        p.setLastName(row.get("lname"));  
        return p;  
    }  
});
```

# Lambda types

- Lambda types in Java 8 are SAMs
- Single Abstract Method
- Interfaces with one and only one method
  - Or multiple override methods with...
    - Same name
    - Type compatible parameters



# Lambda types

- The JDK uses the SAM pattern extensively
  - Runnable, Comparator\*, Predicate, PropertyChangeListener, etc...
  - Lambda syntax will be usable by a large subset of the existing Java APIs

\*

# Collection API

- In Java you tend to pass Collections to Functions
- In FP, you tend to pass Functions to Collections
- `forEach`, `filter`, `map`, `flatMap`...
- `map`, `flatMap`, `filter` produce Lazy "Streams" and are converted to collections using an `into` method

# Defender Methods

- A nice side effect of extending the Collections API while maintaining backward compatibility...
- Java gets Mixins/Traits
- Interfaces can now provide **default** implementation of methods
- Different than abstract classes?

# SAMs & Scala

- Function types in Scala boil down to:
  - A Trait (re: interface)
  - An `apply` method
- These are SAMs and Scala defines `Function[R], Function1[P,R], ..., Function22[...]` in the API

# Comparison to Scala

Java 8

Scala

- Lambda **definition** syntax

$x \rightarrow x + 1$



- Type Inference

$(\text{Integer } x) \rightarrow x + 1$   
 $x \rightarrow x + 1$



- Lambda **type** definition syntax

$\text{Function} \langle T, R \rangle f = \dots$   
 $(T \Rightarrow R) f = \dots$



# Comparison to Scala

Java 8

Scala

- Closures

```
Integer x = 10;  
Runnable r = () -> print(x);
```



# In Conclusion...

- Java is 1995 Cool again.



## Presentation Source Code

<https://github.com/ctataryn/wfpg-java8-lambda.git>

## Install JDK8 with Lambda support

<http://jdk8.java.net/lambda>