goto; copenhagen

GOTO Copenhagen 2019
Conference Nov. 18 - 20

The Sincerest Form of Flattery

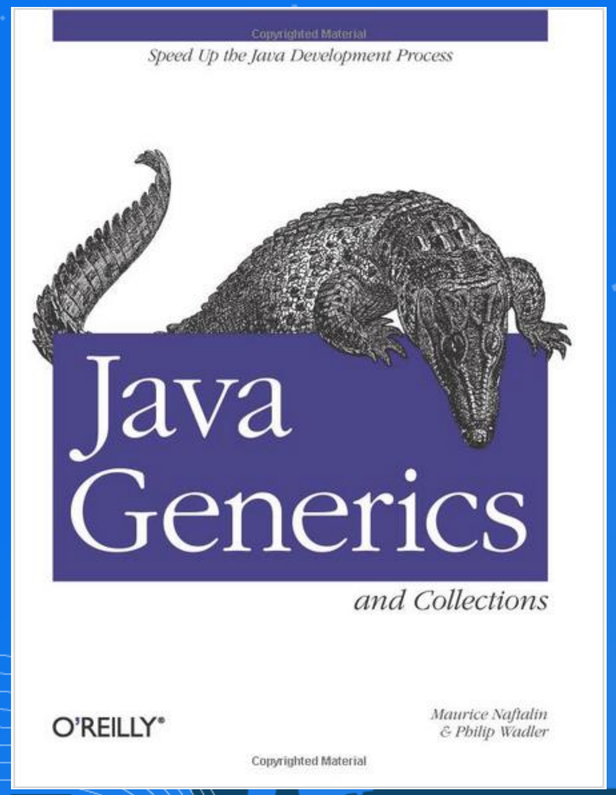
Maurice Naftalin
@MauriceNaftalin

José Paumard
@JosePaumard

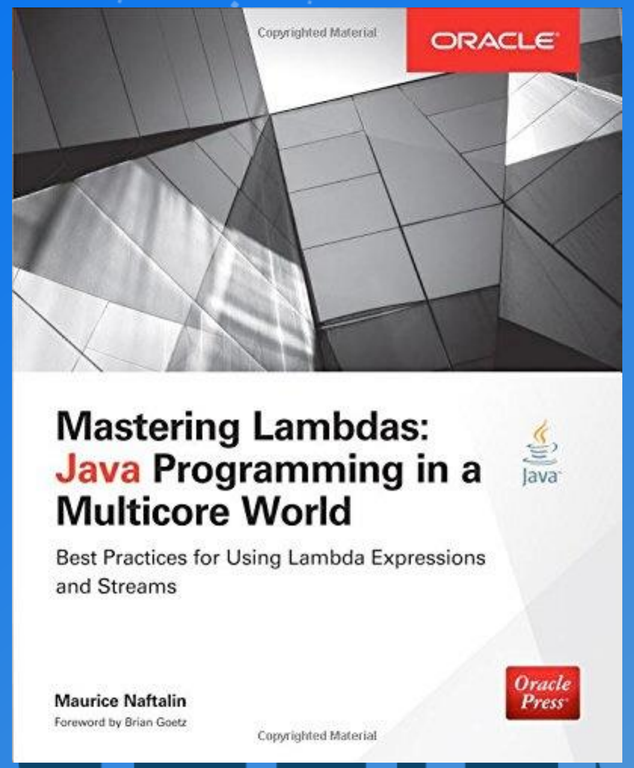
goto; copenhagen

Maurice Naftalin

Java 5



Java 8



Oracle Groundbreaker Ambassador

2013 2014 2015 2017

Follow us @gotocph

@MauriceNaftalin @JosePaumard

goto; copenhagen

Maurice Naftalin

Developer and Technology Evangelist, CDG Group
https://groupcdg.com

José PAUMARD

@JosePaumard

https://josepaumard.github.io https://github.com/JosePaumard

https://www.pluralsight.com/authors/jose-paumard

https://www.slideshare.net/jpaumard

https://www.youtube.com/user/JPaumard


JAlba
@JAlbaUnconf
https://jalba.scot/

Functional Programming: Elegant, Mathematically Sound

Higher Order Functions

(Lisp, 1958)

Parametric Polymorphism

(ML, 1975)

Pattern Matching

(Caml, 1985)

But often not practicable

Object-Oriented Languages: Successful but Envious

From the 1980s, OO languages dominated

- Subtype Polymorphism, Inheritance
- Strong Static Typing
- Automatic Memory Management

Object-Oriented Languages: Successful but Envious

From the 1980s, OO languages dominated

- Subtype Polymorphism, Inheritance
- Strong Static Typing
- Automatic Memory Management

But they always suffered feature envy

Object-Oriented Languages: Successful but Envious

From the 1980s, OO languages dominated

- Subtype Polymorphism, Inheritance
- Strong Static Typing
- Automatic Memory Management

But they always suffered feature envy

So, Pizza!

Pizza =

Java

+

Generics + Higher Order Functions + Pattern-Matching

Pizza into Java:
Translating theory into practice

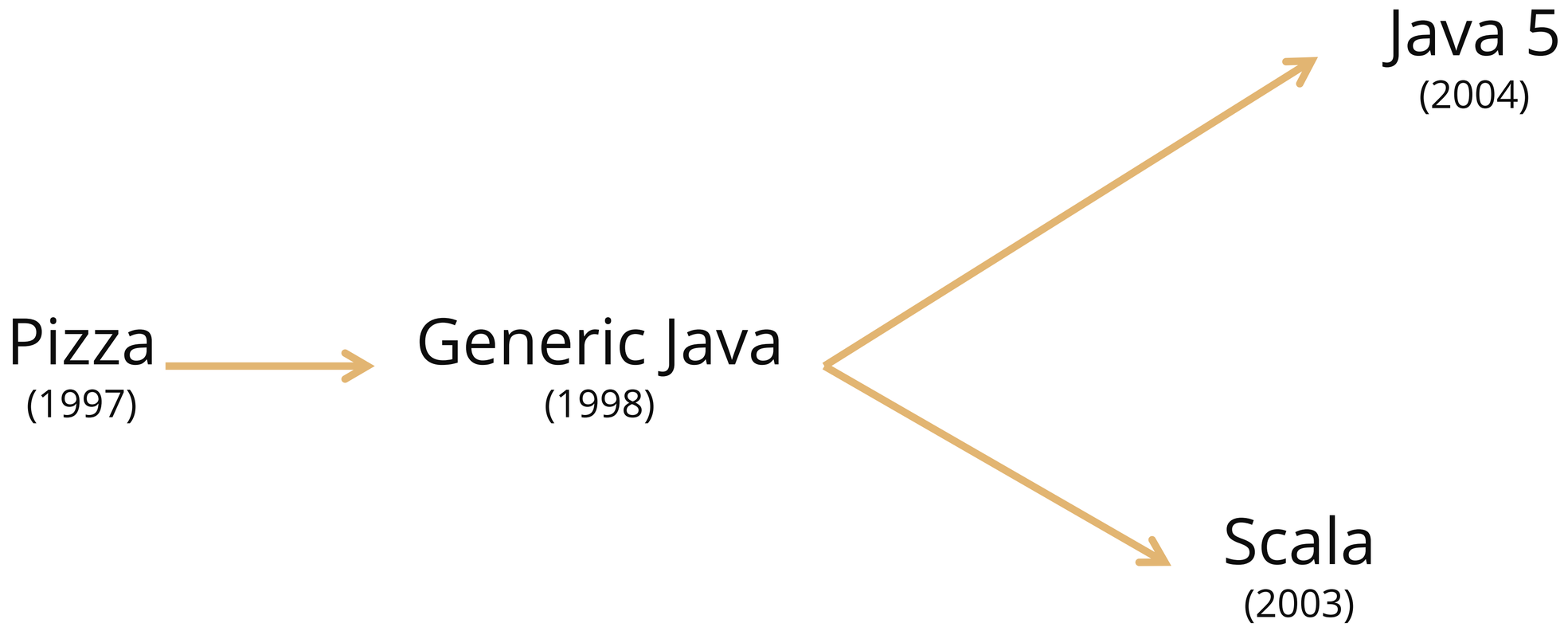
Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

Abstract

Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymorphism, higher-order functions, and algebraic data types.

- parametric polymorphism,
- higher-order functions, and
- algebraic data types.


Pizza
(1997)
Generic Java
(1998)
Java 5
(2004)
Scala
(2003)

Java Ate Pizza in Three Courses

- Parametric Polymorphism in 2004
- Higher Order Functions in 2014
- Pattern Matching 20x☺

Scala Ate Pizza In One Bite

Scala used all of those in 2004

Much earlier than Java

- No need for backward compatibility!

Part 1: Generics

Generics – Stating the problems

Problem 1: Type information availability at runtime

What you want to write:

```
List<Integer> ints = ...;
Integer[] intArray = ints.toArray();
```

vs. what you have to write:

```
List<Integer> ints = ...;
Integer[] intArray = ints.toArray(Integer[]::new);
```

Generics – Stating the problems

Problem 2: Array subtyping

This code was allowed from JDK1.0:

```
Integer[] ints = ...;
Number[] numbers = ints;
```

Covariance: `Integer[]` is a subtype of `Number[]`

Generics – Stating the problems

Problem 2: Array subtyping

This code was allowed from JDK1.0:

```
Integer[] ints = ...;
Number[] numbers = ints;
```

Covariance: `Integer[]` is a subtype of `Number[]`

Adopted to make generic methods possible:

```
Arrays.sort(Object[] objects); // can be called with Integer[]
```

Generics – Stating the problems

But then the problem is: you can write this code:

```
numbers[0] = 3.14;
```

That throws an ArrayStoreException

Covariance is only good for getting things out from a container, not putting them in

Type Erasure in Generic Java

Type erasure consists in replacing the parameter type with its bound in the class file

Adopted for backwards compatibility with legacy ungenerified libraries

Type Erasure in Generic Java

There is no T in the class file

```
public class Holder<T> {

   private T t;

   public Holder(T t)

   public void set(T t)
}
```

```
public class Holder {

   private Object t;

   public Holder(Object t)

   public void set(Object t)
}
```

Type Erasure: How does it Work?

So the compiler is doing the type checking

```
Holder<String> holder = new Holder<String>();
holder.set(10); // Compiler error
```

And it also adds casts when needed

```
String result = holder.get();
```

```
String result = (String)holder.get();
```

Type Erasure vs. Inheritance

How is inheritance handled?

```
public class StringHolder extends
Holder<String> {

   private String s;

   public StringHolder(String s) {
      ...
   }

   public void set(String s) {
      ...
   }
}
```

```
public set(String):void

public synthetic bridge set(Object):void
   CHECKCAST String
   INVOKEVIRTUAL set(String):void
```

Type Erasure vs. Inheritance

How is inheritance handled?

```
public class StringHolder extends
Holder<String> {

    private String s;

    public StringHolder(String s) {
        super(s);
    }

    public String get() {
        return super.get();
    }
}
```

```
public class Holder<T> {

    private T t;

    public Holder(T t) {
        this.t = t;
    }

    public T get() {
        return this.t;
    }
}
```

How about Arrays?

Suppose the array is an array of generics

```
Holder<String>[] stringHolders = new Holder<String>[10];
Object[] objects = stringHolders;

objects[0] = new Holder<Integer>(10); // ArrayStoreException?
```

It would be nice to have an ArrayStoreException

But the array is in fact an array of erased holders

So… arrays of generics can’t be created in Java

How about List in Java?

Covariance is problematic, so what can we use for collections?

`List<Integer>` cannot be a subtype of `List<Number>`

Because we know that this code cannot work

```
List<Integer> ints = ...;
List<Number> numbers = ints;

numbers.add(3.14d);
```

How about List?

If `List<Integer>` is not a subtype of `List<Number>`…

Without generic methods, how many `sort()` overloads will we need?

```
sort(List<Number> numbers)
sort(List<Integer> ints)
... ☹
```

Wildcards to the rescue! (really?)

We need a list that accepts Number and any subtype of Number

```
List<? extends Number> numbers
```

So now we can have one method

```
~~sort(List<Number> numbers)~~
~~sort(List<Integer> ints)~~
sort(List<? extends Number> numbers)
```

Wildcards to the rescue! (really?)

We need a list that accepts Number and any subtype of Number

```
List<? extends Number> numbers
```

So now we have one method

```
sort(List<Number> numbers)
sort(List<Integer> ints)
sort(List<? extends Number> numbers)
```

Covariance works here, because the method *retrieves* values

Generics in Scala

New language, free of constraints on Java:

- New collections library, defined according to their use:
- Immutable structures can be *declared* as covariant in their component type

```
class Pair[+T](val fst: T, val snd: T)
def toStrng(p: Pair[AnyVal]){…}
toStrng(new Pair(1, 2))
toStrng(new Pair("one", "two"))
```

Java

```
class Pair<T> {
    Pair(T fst, T snd) {...}
}
void toStrng(Pair<?> p){...}
```

- Declaration-site variance simplifies client code

Part 2: Closures and Lambdas

Closures: What Pizza Did

How to include in Java, which does not have explicit function types?

Solution: for each function, create an abstract class with an apply method.

Capturing non-final local variables? They sat on the fence!

Closures in Pizza

Functions as first-class citizens: the essence of FP

```
max(List<S>, (S, S) => boolean) => S
max(List.of("one", "three"), (s0, s1) -> s0.length() > s1.length())
// returns "three"
```

Higher-order functions originally from λ-calculus / LISP

~~Closures~~ Lambdas in Java

2006 debate over whether to introduce a function type.

Eventually resolved in favour of using a Single Abstract Method Interface (Functional Interface)

No change in the type system: it preserves the nominal typing and simplicity of language

Capture of non-final local variables prevented, to preserve thread safety

Closures in Scala

Function types (structural) fundamental element of the type system.

Closures in Scala can capture vars

How does it work with concurrency?

Short answer: it doesn’t. You have to do the work!

Partial Application in Scala

One fundamental operation in Functional Programming is Partial Application, achieved by currying

```
def partialApplication(bi: (String, String) => Int, word: String) {
   return s => bi(word, s)
}

val bi: = (word: String, s: String) => word.indexOf(s)
val f: String => Integer = partialApplication(_.indexOf(_), "Hello")
```

```
val bi: = (word: String)(s: String) => word.indexOf(s)
val f: String => Integer = bi("Hello")
```

Partial Application in Java

The same thing is done in Java with bound method references

```
Function<String, Integer> partialApplication(
   BiFunction<String, String, Integer> biFunction, String word) {
   return s -> bi.apply(word, s);
}

BiFunction<String, String, Integer> bi = (word, s) -> word.indexOf(s);
var f = partialApplication(String::indexOf, "Hello");
```

```
Function<String, Integer> f = "Hello"::indexOf;
```

Part 3: Pattern Matching

Pattern Matching in Pizza

Just a switch on types already with deconstruction

```
class Vehicle {
  case Car(String color);
  case Bus();
}

static int capacity(Vehicle vehicle) {
  switch(vehicle) {
    case Car(color):
      return "red".equals(color)? 20: 4; // red cars are BIG!
    case Bus:
      return 12;
  }
}
```

Pattern Matching in Scala

Match expression on types with deconstruction, no exhaustiveness check

```
sealed trait Vehicle;
case class Car(String color) extends Vehicle
case object Bus extends Vehicle

def capacity(vehicle: Vehicle) {
  return vehicle match {
    case Car(color) => "red".equals(color)? 20: 4
    case Bus => 12
  }
}
```

Pattern Matching in Java

Switch expression on types with deconstruction and exhaustiveness check

```
sealed interface Vehicle {
   record Car(String color) implements Vehicle {}
   static class Bus implements Vehicle {}
}

static int capacity(Vehicle vehicle) {
  return switch(vehicle) {
    case Car("red") -> 20;
    case Car(var color) -> 4;
    case Bus -> 12;
  };
}
```

Conclusion

Language design is really complicated!

Backwards compatibility makes it *even more* complicated

Java has to take it very seriously

Scala has chosen fast language evolution over backwards compatibility

Both viewpoints have their advantage!

Conclusion

« We don’t want to be the first to include a feature, because every feature we add will never be removed »

Brian Goetz,
Java Language Architect

goto; copenhagen

GOTO Copenhagen 2019
Conference Nov. 18 - 20

Thank you!

Maurice Naftalin
@mauricenaftalin

José Paumard
@JosePaumard

Follow us @gotocph

@MauriceNaftalin @JosePaumard

goto; copenhagen

GOTO Copenhagen 2019

Conference Nov. 18 - 20

Click ‘Rate Session’ to rate session and ask questions.

Follow us @gotocph