START

Movie Time!

JAVA 8

THE GOOD PARTS

Andrzej Grzesik

Konrad Malawski

JAVA 8

Andrzej Grzesik

Konrad Malawski

JAVA 8

THE GOOD PARTS

Andrzej Grzesik

Konrad Malawski

Andrzej Grzesik

@ags313

andrzej@grzesik.it

andrzejgrzesik.info

ABOUT:ME

[sckrk]

lunar logic polska

Konrad `@ktosopl` Malawski

Konrad `@ktosopl` Malawski

OUR OPINIONS ARE OUR OWN

disclaimer

QUESTIONS?

QUESTIONS?

ask them right away!

JAVA 8

is going to be amazing!

TWITTER SAYS:

JAVA 8 IS THE NEW GUAVA

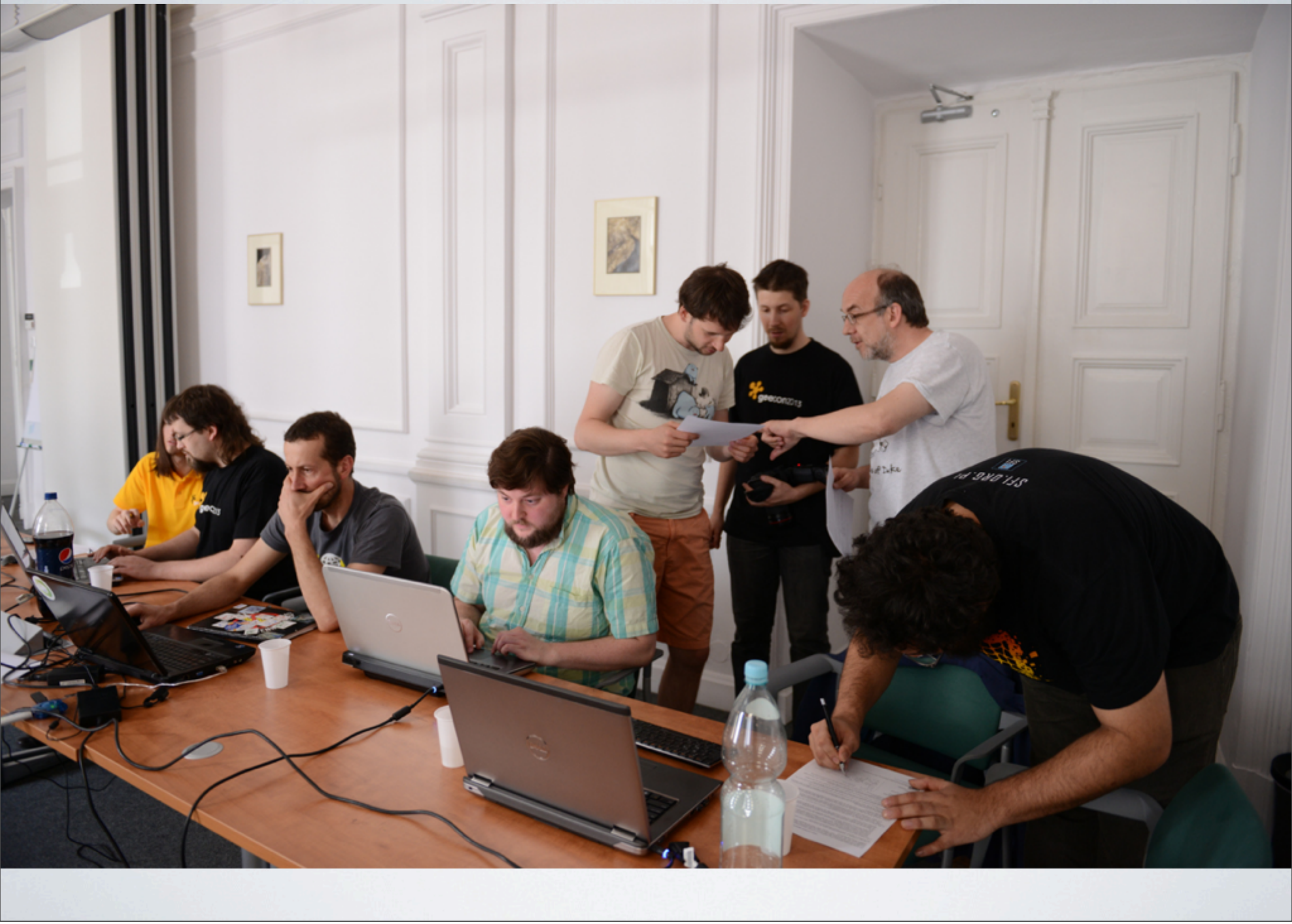
THE MOST EXCITING RELEASE IN HISTORY

DONE WITH COMMUNITY

ADOPT OPENJDK

adoptopenjdk.java.net

FALSY
VALUES
Neo4j
burton



YOU CAN HELP!

FIX

TEST

HACK

DOCUMENT

ADOPTOPENJDK.JAVA.NET

ADOPTAJSR.JAVA.NET

HOW DO I CHECK JDK8?

JDK8.JAVA.NET

IDE SUPPORT

```
map.computeIfPresent(aKey, new BiFunction<Key, Value, Value>()
{
    @Override
    public Value apply(Key key, Value va
    {
        // ...
    }
});
```

Replace with lambda

Introduce local variable

```
        map.compute(aKey, new BiFunction<Key, Value, Value>(){

                                                          lue u) {
                    throw new UnsupportedOperationException("Not supported
                }


        });
    }
```

Use Lambda

Convert Anonymous to Member

Configure "Null Pointer Dereference" Hint

JENV

http://jenv.be

JENV

```
$ jenv versions

  system
  oracle64-1.6.0.51
  oracle64-1.7.0.40
* oracle64-1.8.0-ea (set by /Users/ktoso/.jenv/version)
```

JENV

```
ktoso @ 月/tmp
$ jenv local oracle64-1.7.0.40
```

JENV

```
ktoso @ 月/tmp
$ jenv local oracle64-1.7.0.40
```

```
ktoso @ 月/tmp
$ jenv versions
  systema
  oracle64-1.6.0.51
* oracle64-1.7.0.40 (set by /tmp/.java-version)
  oracle64-1.8.0-ea
```

NEW TIME API

jsr 310

```
void immutable()
{
    LocalTime aTime    = LocalTime.now();
    print("now:    %s", aTime);

    LocalTime newTime = aTime.plusMinutes(16);
    print("now:    %s, later: %s", aTime, newTime);
}
```

now: 01:25:56.916

```
void immutable()
{
    LocalTime aTime    = LocalTime.now();
    print("now:    %s", aTime);

    LocalTime newTime = aTime.plusMinutes(16);
    print("now:    %s, later: %s", aTime, newTime);
}
```

```
now:    01:25:56.916
```

```
void immutable()
{
    LocalTime aTime    = LocalTime.now();
    print("now:    %s", aTime);

    LocalTime newTime = aTime.plusMinutes(16);
    print("now:    %s, later: %s", aTime, newTime);
}
```

```
now:    01:25:56.916
later: 01:41:56.916
```

```java
private void localTime()
{
    LocalDate today     = LocalDate.now();
    LocalDate yesterday = today.minusDays(1);

    // Geek Bike Ride!
    LocalDateTime localDateTime = yesterday.atTime(11, 30);

    LocalDateTime earlyMorning  = LocalDate.of(2013, 9, 22)
                                      .atStartOfDay();
}
```

```
void flightTime()
{
    ZoneId LHR = ZoneId.of("Europe/London");
    ZoneId SFO = ZoneId.of("America/Los_Angeles");

    LocalDate date = LocalDate.of(2013, Month.SEPTEMBER, 14);
    LocalTime takeoff = LocalTime.of(12, 50);
    LocalTime landing = LocalTime.of(16, 20);
    Duration flightTime = Duration.between(
                                ZonedDateTime.of(date, takeoff, LHR),
                                ZonedDateTime.of(date, landing, SFO));

    System.out.println("Flight time: " + flightTime);
}
```

```
void flightTime()
{
    ZoneId LHR = ZoneId.of("Europe/London");
    ZoneId SFO = ZoneId.of("America/Los_Angeles");

    LocalDate date = LocalDate.of(2013, Month.SEPTEMBER, 14);
    LocalTime takeoff = LocalTime.of(12, 50);
    LocalTime landing = LocalTime.of(16, 20);
    Duration flightTime = Duration.between(
                                ZonedDateTime.of(date, takeoff, LHR),
                                ZonedDateTime.of(date, landing, SFO));

    System.out.println("Flight time: " + flightTime);
}
```

Flight time:
PT11H30M

ISO BY DEFAULT

NO MORE

```
new SimpleDateFormat("yyyy-MM-dd'T'HH:mm'Z'");
```

```
void formatting()
{
    DateTimeFormatter.ISO_DATE.
                        format(LocalDateTime.of(2013, 9, 22, 10, 03));

    DateTimeFormatter.ISO_DATE_TIME.
                        format(LocalDateTime.of(2013, 9, 22, 10, 30));
}
```

```
void formatting()
{
    DateTimeFormatter.ISO_DATE.
                        format(LocalDateTime.of(2013, 9, 22, 10, 03));

    DateTimeFormatter.ISO_DATE_TIME.
                        format(LocalDateTime.of(2013, 9, 22, 10, 30));
}
```

```
void formatting()
{
    DateTimeFormatter.ISO_DATE.
                        format(LocalDateTime.of(2013, 9, 22, 10, 03));

    DateTimeFormatter.ISO_DATE_TIME.
                        format(LocalDateTime.of(2013, 9, 22, 10, 30));
}
```

2013-09-22

2013-09-22T10:30:00

```
 void formatterError()
  {
    ISO_DATE_TIME.format(LocalDate.of(2013, 9, 22));

/*
Exception in thread "main"
java.time.temporal.UnsupportedTemporalTypeException: Unsupported
field: HourOfDay
  at java.time.LocalDate.get0(LocalDate.java:670)
  at java.time.LocalDate.getLong(LocalDate.java:649)
  at
java.time.format.DateTimePrintContext.getValue(DateTimePrintContext.ja
va:297)
  (..)
*/
  }
```

TUTORIAL

http://docs.oracle.com/javase/tutorial/datetime/index.html

API ENHANCEMENTS

BETTER IO

```
void betterIO()
{
    BufferedReader bufferedReader;
    Path path;

    Stream<String> lines = bufferedReader.lines();
    Stream<String> lines = Files.lines(Path, Charset);

    Stream<Path>   paths = Files.list(Path);
    Stream<Path>   paths = Files.find(Path, depth, BiPredicate,
                                                   FileVisitOption...)

    Stream<Path>   paths = Files.walk(Path, depth, FileVisitOption...)
    Stream<Path>   paths = Files.walk(Path, FileVisitOption...)


    DirectoryStream.stream()
}
```

MAPS

```
compute()
{
    map.compute(aKey, new BiFunction<Key, Value, Value>() {
      @Override
      public Value apply(Key key, Value value)
      {
        // ...
      }
    });

    map.computeIfAbsent(aKey, new Function<Key, Value>() {
      @Override
      public Value apply(Key key)
      {
        // ...
      }
    });

    map.computeIfPresent(aKey, new BiFunction<Key, Value, Value>() {
      @Override
      public Value apply(Key key, Value value)
      {
        // ...
      }
    });
}
```

```
void computeWithLambdas()
{
    Map<Key, Value> map = // ...

    map.computeIfAbsent(aKey, key -> {
       // ...
    });

    map.computeIfPresent(aKey, (key, value) -> {
       // ...
    });
}
```

```
void moreMaps()
{
    Map<Key, Value> map = null;

    map.putIfAbsent(K, V);
    map.remove(Object, Object);
    map.replace(K, V);
    // Compare and swap
    map.replace(K, V1, V2);
    map.replaceAll(BiFunction);

    map.getOrDefault(K, V);

    map.merge(K, V, BiFunction)
}
```

```
void parallelSetAll()
{
    int[] array = new int[8];

    AtomicInteger i = new AtomicInteger();
    Arrays.parallelSetAll(array, operand -> i.incrementAndGet());
}
```

```
void parallelPrefix()
{
    int[] array = { 1, 2, 4, 8 };

    Arrays.parallelPrefix(array, (left, right) -> {
      return left + right;
    });
}
```

LAMBDAS?

LAMBDAS!

(finally)

LAMBDAS

Notable inspirations would be:

Scala
Groovy
Lisps
.NOT (!)

LAMBDAS

```
() -> {}
```

LAMBDAS

```
(Thing t) -> {}
```

LAMBDAS

LAMBDAS

```
(Thing t) -> {}
```

LAMBDAS

```
(Thing t) -> {}
```

```
(Thing t, More m) -> {}
```

LAMBDAS & TYPES

LAMBDAS & TYPES

```
GetNum _ = (t) -> {42}
```

LAMBDAS & TYPES

```
GetNum _ = (t) -> {42}
GetNum _ = (t) ->  42
```

LAMBDAS & TYPES

```
GetNum _ = (t) -> {42}
GetNum _ = (t) ->  42
GetNum _ =  t  -> 1337
```

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}
```

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };
```

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };
```

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}
```

```
Adder function = (int a, int b) -> { a + b };
```

(int, int) => int

gets converted into target type:

Adder

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };

 // or shorter:

 Adder function = (a, b) -> a + b;
```

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };

 // or shorter:

 Adder function = (a, b) -> a + b;
```

You can skip the ; sign!

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };

 // or shorter:

 Adder function = (a, b) -> a + b;
```

You can skip the ; sign!

You can skip { } sometimes

TARGET TYPING

```
interface Adder {
   int add(int a, int b);
}


 Adder function = (int a, int b) -> { a + b };

 // or shorter:

   Adder function = (a, b) -> a + b;
```

You can skip the **;** sign!

You can skip **{ }** sometimes

and the **types** are **inferred**!

FUNCTIONAL INTERFACES

```
interface Adder {
   int add(int a, int b);
}
```

FUNCTIONAL INTERFACES

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
}
```

FUNCTIONAL INTERFACES

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
}
```

Similar to **@Override**:
* not required,
* checks our intent.

FUNCTIONAL INTERFACES

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
   void wat();
}
```

FUNCTIONAL INTERFACES

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
   void wat();
}
```

java: Unexpected @FunctionalInterface annotation
pl.project13.lambda.test.examples.Adder is not a functional interface
multiple non-overriding **abstract methods found in interface** pl.project13.lambda.test.examples.Adder

DEFAULT METHODS

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
   default void wat() { /* nothing... */ }
}
```

OK!
Only 1 abstract method.

DEFAULT METHODS

```
@FunctionalInterface
interface Adder {
  default int add(int a, int b) { return a + b; }
}

@FunctionalInterface
interface Divider {
  default double divide(int a, int b) { return a / b; }
}

class Calculator implements Adder, Divider {


  public double calc(int a, int b, int c) {
    return divide(add(a, b), c);
  }
}
```

DEFAULT METHODS

```
@FunctionalInterface
interface Adder {
  default int add(int a, int b) { return a + b; }
}

@FunctionalInterface
interface Divider {
  default double divide(int a, int b) { return a / b; }
}

class Calculator implements Adder, Divider {

  public double calc(int a, int b, int c) {
    return divide(add(a, b), c);
  }
}
```

We mixed in methods!

here!

and here!

DEFAULT METHODS

```
interface A {
  default void doIt() { /* A */ }
}

interface B {
  default void doIt() { /* B */ }
}

class Thing implements A, B {
}
```

DEFAULT METHODS

```
interface A {
  default void doIt() { /* A */ }
}

interface B {
  default void doIt() { /* B */ }
}

class Thing implements A, B {
}
```

java: class com.javaone.**Thing inherits unrelated defaults for doIt()** from types com.javaone.**A** and com.javaone.**B**

DEFAULT METHODS

```
interface A {
  default void doIt() { /* A */ }
}

interface B {
  default void doIt() { /* B */ }
}

class Thing implements A, B {
  @Override
  public void doIt() {
    A.super.doIt();
  }
}
```

Resolve ambiguity manually!

DEFAULT METHODS

```
interface A {
  default void doIt() { /* A */ }
}

interface B {
  default void doIt() { /* B */ }
}

class Thing implements A, B {
  @Override
  public void doIt() {
    A.super.doIt();
  }
}
```

Resolve ambiguity manually!

DEFAULT IN ITERABLE

```
package java.lang;

@FunctionalInterface
public interface Iterable<T> {

    Iterator<T> iterator();

    /** @since 1.8 */
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
```

λ IN ACTION

```
void withoutLambda()
{
    button.addActionListener(new AbstractAction()
    {
      @Override
      public void actionPerformed(ActionEvent e)
      {
        System.out.println("example");
      }
    });
}
```

BEFORE LAMBDAS

in IntelliJ

```
// when
button.addActionListener((AbstractAction) (e) -> { System.out.println("example"); });
```

λ IN ACTION

```
void withLambda()
{
    button.addActionListener((e) ->
    {
        System.out.println("example");
    });
}
```

```
void composingFunctions()
{
    // given
    Function<Integer, Integer> timesTwo = n -> n * 2;
    Function<Integer, Integer> plusOne = n -> n + 1;

    // when
    Function<Integer, Integer> multiplyThenAdd =
        timesTwo.andThen(plusOne);

    // equivalent to
    Function<Integer, Integer> multiplyThenAdd =
        plusOne.compose(timesTwo);

    // then
    int result = multiplyThenAdd.apply(1);
    assertThat(result).isEqualTo(3);

}
```

REMOVING BOILERPLATE

STREAMS

```
void transform()
{
    Iterables.transform(
                newArrayList(1, 2, 3),
                new Function<Integer, String>()
    {
      @Override
      public String apply(Integer input)
      {
        return input.toString();
      }
    });
}
```

```
void transform()
{
    Iterables.transform(
                newArrayList(1, 2, 3),
                new Function<Integer, String>()
    {
      @Override
      public String apply(Integer input)
      {
        return input.toString();
      }
    });
 }
```

VS

```
void noMoreTransform()
{
    items.stream().map(i -> i.toString());
}
```

```
items.stream().map(Item::getName);
```

compared to Scala

```
items map { _.getName }
```

yay, we're cool now!

```
items.stream().map(Item::getName);
```

compared to Scala

```
items map { _.getName }
```

STREAMS

```
items.stream().

      filter(predicate);
      map(mapper);
      mapToInt(mapper);
      flatMap(mapper);
      distinct();
      sorted();
      sorted(comparator);
      peek(consumer);
      limit(maxSize);
      forEach(func);
```

INTERNAL ITERATION

```
void internalIteration()
{
    List<Thing> things = ...;

    things.forEach(System.out::println);
}
```

PARALLELIZE?

PARALLEL ITERATION

```
void parallelIteration()
{
    List<Thing> things = ...;

    things.parallelStream().forEach(System.out::println);
}
```

STREAMS ARE LAZY!

STREAMS ARE LAZY

```
   List<Integer> is = newArrayList(1, 2, 3);

is.stream()
   .map(a -> printAndReturn("A",  a))
   .map(a -> printAndReturn("B", a));
```

STREAMS ARE LAZY

```
   List<Integer> is = newArrayList(1, 2, 3);

is.stream()
   .map(a -> printAndReturn("A",  a))
   .map(a -> printAndReturn("B", a));
```

```
Prints:
```

STREAMS ARE LAZY

```
List<Integer> is = newArrayList(1, 2, 3);

is.stream()
  .map(a -> printAndReturn("A",  a))
  .map(a -> printAndReturn("B", a));
```

Prints:

STREAMS ARE LAZY

```
List<Integer> is = newArrayList(1, 2, 3);

is.stream()
  .map(a -> printAndReturn("A",  a))
  .map(a -> printAndReturn("B", a))
  .collect(toList());
```

STREAMS ARE LAZY

```
List<Integer> is = newArrayList(1, 2, 3);

is.stream()
  .map(a -> printAndReturn("A",  a))
  .map(a -> printAndReturn("B", a))
  .collect(toList());
```

Prints:

A1
B1
A2
B2
A3
B3

STREAMS ARE LAZY

```
List<Integer> is = newArrayList(1, 2, 3);

is.stream()
  .map(a -> printAndReturn("A",  a))
  .map(a -> printAndReturn("B", a))
  .collect(toList());
```

```
Prints:

A1
B1
A2
B2
A3
B3
```

It's ONE iteration!

METHOD HANDLES

think function pointers

KEEPING REFERENCES

```
class Person {
   String getName();
}
```

```
??? method = Person::getName
```

KEEPING REFERENCES

```
class Person {
   String getName();
}
```

```
Supplier<String> method = Person::getName
```

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

```
void referringToMethods()
{
    String name = Person.getName();

    String name = applyTo(heinz, Person::getName);
}
```

REFERRING TO METHODS

```
String normalName = heinz.getName();

String magicName = applyTo(heinz, Person::getName);


public <T, R> R applyTo(T obj, Function<T, R> function) {
  return function.apply(obj);
}
```

JAVA.UTIL.FUNCTION.*

```
Supplier<T>
  => T

Consumer<T>
  T => void

Predicate<T>
  T => Boolean

BiPredicate<T1, T2>
  (T1, T2) => Boolean

Function<T, R>
  T => R

BiFunction<T1, T2, R>
  (T1, T2) => R

and more...!
```

JAVA.UTIL.FUNCTION.*

Fact: in order to **refer to:**

```
String doThing(String a, String b, String c, Integer d);
```

JAVA.UTIL.FUNCTION.*

Fact: in order to **refer to:**

```
String doThing(String a, String b, String c, Integer d);
```

you have to:

```
@FunctionalInterface
 interface Function4<T1, T2, T3, T4, R> {
   R apply(T1 a, T2 b, T3 c, T4 d);
 }
```

JAVA.UTIL.FUNCTION.*

Fact: in order to **refer to:**

```
String doThing(String a, String b, String c, Integer d);
```

you have to:

```
@FunctionalInterface
 interface Function4<T1, T2, T3, T4, R> {
   R apply(T1 a, T2 b, T3 c, T4 d);
 }

 Function4<String, String, String, Integer, String> fun =
   Example::doThing;
```

BACK TO THE FUTURE!

- http://cr.openjdk.java.net/**~briangoetz/lambda/collections-overview.html**
- http://docs.oracle.com/javase/tutorial/java/javaOO/**lambdaexpressions.html**
- http://www.techempower.com/blog/2013/03/26/**everything-about-java-8/**
- For fun, Lambda Spec: github.com/**ktoso/lambda-spec**

THANK YOU!

Andrzej Grzesik

@ags313

Konrad Malawski

@ktosopl

TWEET PLEASE!