ABOUT LUKAS EDER

Lukas is a Java and SQL enthusiast developer. He created the Data Geekery GmbH. He is the creator of jOOQ, a comprehensive SQL library for Java, and he is blogging mostly about these three topics: Java, SQL and jOOQ.

Java 8 Friday: The Dark Side of Java 8

Posted by: Lukas Eder in Core Java April 8th, 2014

At Data Geekery, we love Java. And as we're really into jOOQ's fluent API and query DSL, we're absolutely thrilled about what Java 8 will bring to our ecosystem.

Java 8 Friday

Every Friday, we're showing you a couple of nice new tutorial-style Java 8 features, which take advantage of lambda expressions, extension methods, and other great stuff. You'll find the source code on GitHub.

The dark side of Java 8

So far, we've been showing the thrilling parts of this new major release. But there are also caveats. Lots of them. Things that

- … are confusing
- … are wrong
- … are omitted (for now)
- … are omitted (for long)

There are always two sides to Java major releases. On the bright side, we get lots of new functionality that most people would say was *overdue*. Other languages, platforms have had generics long before Java 5. Other languages, platforms have had lambdas long before Java 8. But now, we finally have these features. In the usual quirky Java-way.

Lambda expressions were introduced quite elegantly. The idea of being able to write every anonymous SAM instance as a lambda expression is very compelling from a backwards-compatiblity point of view. So what *are* the dark sides to Java 8?

Overloading gets even worse

Overloading, generics, and varargs aren't friends. We've explained this in a previous article, and also in this Stack Overflow question. These might not be every day problems in your odd application, but they're very important problems for API designers and maintainers.

With lambda expressions, things get "worse". So you think you can provide some convenience API, overloading your existing

```
run()
```

method that accepts a

```
Callable
```

to also accept the new

```
Supplier
```

type:

```
static <T> T run(Callable<T> c) throws Exception {
    return c.call();
}

static <T> T run(Supplier<T> s) throws Exception {
    return s.get();
}
```

What looks like perfectly useful Java 7 code is a major pain in Java 8, now. Because you cannot just simply call these methods with a lambda argument:

```
    run(() -> null);
    //  ^^^^^^^^^^ ambiguous method call
}
```

Tough luck. You'll have to resort to either of these "classic" solutions:

```
run((Callable<Object>) (() -> null));
    run(new Callable<Object>() {
        @Override
        public Object call() throws Exception {
            return null;
        }
    });
```


So, while there's always a workaround, these workarounds always "suck". That's quite a bummer, even if things don't break from a backwards-compatibility perspective.

Not all keywords are supported on default methods

Default methods are a nice addition. Some may claim that Java finally has traits. Others clearly dissociate themselves from the term, e.g. Brian Goetz:

> *The key goal of adding default methods to Java was "interface evolution", not "poor man's traits."*

As found on the lambda-dev mailing list.

Fact is, default methods are quite a bit of an orthogonal and irregular feature to anything else in Java. Here are a couple of critiques:

They cannot be made final

Given that default methods can also be used as convenience methods in API:

```
public interface NoTrait {

    // Run the Runnable exactly once
    default final void run(Runnable r) {
        //  ^^^^^ modifier final not allowed
        run(r, 1);
    }

    // Run the Runnable "times" times
    default void run(Runnable r, int times) {
        for (int i = 0; i < times; i++)
            r.run();
    }
}
```

Unfortunately, the above is not possible, and so the first overloaded convenience method could be overridden in subtypes, even if that makes no sense to the API designer.

They cannot be made synchronized

Bummer! Would that have been difficult to implement in the language?

```
public interface NoTrait {
    default synchronized void noSynchronized() {
        //  ^^^^^^^^^^^^ modifier synchronized
        //  not allowed
        System.out.println("noSynchronized");
    }
}
```

Yes,

```
synchronized
```

is used rarely, just like final. But when you have that use-case, why not just allow it? What makes interface method bodies so special?

The default keyword

This is maybe the weirdest and most irregular of all features. The

```
default
```

keyword itself. Let's compare interfaces and abstract classes:

```
// Interfaces are always abstract
public /* abstract */ interface NoTrait {

    // Abstract methods have no bodies
    // The abstract keyword is optional
    /* abstract */ void run1();

    // Concrete methods have bodies
    // The default keyword is mandatory
    default void run2() {}
}
```

```
12
13 // Classes can optionally be abstract
14 public abstract class NoInterface {
15
16     // Abstract methods have no bodies
17     // The abstract keyword is mandatory
18     abstract void run1();
19
20     // Concrete methods have bodies
21     // The default keyword mustn't be used
22     void run2() {}
23 }
```

If the language were re-designed from scratch, it would probably do without any of

```
abstract
```

or

```
default
```

keywords. Both are unnecessary. The mere fact that there is or is not a body is sufficient information for the compiler to assess whether a method is abstract. I.e, how things should be:

```
1 public interface NoTrait {
2     void run1();
3     void run2() {}
4 }
5
6 public abstract class NoInterface {
7     void run1();
8     void run2() {}
9 }
```

The above would be much leaner and more regular. It's a pity that the usefulness of

```
default
```

was never really debated by the EG. Well, it was debated but the EG never wanted to accept this as an option. I've tried my luck, with this response:

> *I don't think #3 is an option because interfaces with method bodies are unnatural to begin with. At least specifying the "default" keyword gives the reader some context why the language allows a method body. Personally, I wish interfaces would remain as pure contracts (without implementation), but I don't know of a better option to evolve interfaces.*

Again, this is a clear commitment by the EG not to commit to the vision of "traits" in Java. Default methods were a pure necessary means to implement 1-2 other features. They weren't well-designed from the beginning.

Other modifiers

Luckily, the

```
static
```

modifier made it into the specs, late in the project. It is thus possible to specifiy static methods in interfaces now. For some reason, though, these methods do not need (nor allow!) the

```
default
```

keyword, which must've been a totally random decision by the EG, just like you apparently cannot define

```
static final
```

methods in interfaces.

While visibility modifiers were discussed on the lambda-dev mailing list, but were out of scope for this release. Maybe, we can get them in a future release.

Few default methods were actually implemented

Some methods would have sensible default implementations on interface – one might guess. Intuitively, the collections interfaces, like

```
List
```

or

```
Set
```

would have them on their

```
equals()
```

and

```
hashCode()
```

methods, because the contract for these methods is well-defined on the interfaces. It is also implemented in

```
AbstractList
```

, using

```
listIterator()
```

, which is a reasonable default implementation for most tailor-made lists.

It would've been great if these API were retrofitted to make implementing custom collections easier with Java 8. I could make all my business objects implement

```
List
```

for instance, without wasting the single base-class inheritance on

```
AbstractList
```

.

Probably, though, there has been a compelling reason related to backwards-compatibility that prevented the Java 8 team at Oracle from implementing these default methods. Whoever sends us the reason why this was omitted will get a free jOOQ sticker !

The wasn't invented here – mentality

This, too, was criticised a couple of times on the lambda-dev EG mailing list. And while writing this blog series, I can only confirm that the new functional interfaces are very confusing to remember. They're confusing for these reasons:

Some primitive types are more equal than others

The

```
int
```

,

```
long
```

,

```
double
```

primitive types are preferred compared to all the others, in that they have a functional interface in the java.util.function package, and in the whole Streams API.

```
boolean
```

is a second-class citizen, as it still made it into the package in the form of a

```
BooleanSupplier
```

or a

```
Predicate
```

, or worse:

```
IntPredicate
```

.

All the other primitive types don't really exist in this area. I.e. there are no special types for

```
byte
```

,

```
short
```

,

```
float
```

, and

```
char
```

. While the argument of meeting deadlines is certainly a valid one, this quirky status-quo will make the language even harder to learn for newbies.

The types aren't just called Function

Let's be frank. All of these types are simply "functions". No one really cares about the implicit difference between a

```
Consumer
```

, a

```
Predicate
```

, a

```
UnaryOperator
```

, etc.

In fact, when you're looking for a type with a non-

```
void
```

return value and two arguments, what would you probably be calling it?

```
Function2
```

? Well, you were wrong. It is called a

```
BiFunction
```

.

Here's a decision tree to know how the type you're looking for is called:

- Does your function return

  ```
  void
  ```

  ? It's called a

  ```
  Consumer
  ```

- Does your function return

  ```
  boolean
  ```

  ? It's called a

  ```
  Predicate
  ```

- Does your function return an

  ```
  int
  ```

  ,

  ```
  long
  ```

  ,

  ```
  double
  ```

  ? It's called

  ```
  XXToIntYY
  ```

  ,

  ```
  XXToLongYY
  ```

  ,

  ```
  XXToDoubleYY
  ```

  something

- Does your function take no arguments? It's called a

  ```
  Supplier
  ```

- Does your function take a single

  ```
  int
  ```

  ,

  ```
  long
  ```

  ,

  ```
  double
  ```

  argument? It's called an

  ```
  IntXX
  ```

  ,

  ```
  LongXX
  ```

  ,

  ```
  DoubleXX
  ```

  something

- Does your function take two arguments? It's called

  ```
  BiXX
  ```

- Does your function take two arguments of the same type? It's called

  ```
  BinaryOperator
  ```

- Does your function return the same type as it takes as a single argument? It's called

  ```
  UnaryOperator
  ```

```
UnaryOperator
```

- Does your function take two arguments of which the first is a reference type and the second is a primitive type? It's called

```
ObjXXConsumer
```

(only consumers exist with that configuration)

- Else: It's called

```
Function
```

Good lord! We should certainly go over to Oracle Education to check if the price for Oracle Certified Java Programmer courses have drastically increased, recently… Thankfully, with Lambda expressions, we hardly ever have to remember all these types!

More on Java 8

Java 5 generics have brought a lot of great new features to the Java language. But there were also quite a few caveats related to type erasure. Java 8's default methods, Streams API and lambda expressions will again bring a lot of great new features to the Java language and platform. But we're sure that Stack Overflow will soon burst with questions by confused programmers that are getting lost in the Java 8 jungle.

Learning all the new features won't be easy, but the new features (and caveats) are here to stay. If you're a Java developer, you better start practicing now, when you get the chance. Because we have a long way to go.

Reference: Java 8 Friday: The Dark Side of Java 8 from our JCG partner Lukas Eder at the JAVA, SQL, AND JOOQ blog.

Tagged with: JAVA 8

9 COMMENTS

Adler Fleurant
April 8th, 2014 at 2:27 pm

It is understandable why default methods cannot be final. I believe default methods are provided in subclasses by "default" if no implementations are provided. Making a default method final would mean that it is the only possible implementation for any subclass. By definition, default and final should be incompatible.

Still no synchronized makes not sense to me.

Reply

Lukas Eder
April 9th, 2014 at 9:45 am

Yes, your line of thought certainly makes sense.

The critique here, however, is the fact that "default" is an unnecessary keyword *if* we allow interface methods to have bodies in general. And if we do that, then final wouldn't be so unnatural. The expert group expressly tried to avoid making interfaces and classes more alike, though.

Reply

Rajan
February 13th, 2015 at 8:40 am

you are right dear.

Reply

Jake Zim

April 8th, 2014 at 7:31 pm

The overloading thing is a non-issue. Don't bother with the second method. If, for some reason, you want to throw a supplier in there, you can simply put in a method reference, which Java will automatically convert into a Callable.
I mostly disagree with your complaints about the naming conventions for the functional interfaces. For the most part, their names are actually helpful. Some of them are a little worthless, though (the ones involving primitives). I don't know why you would complain about Predicate; any time someone makes a lambda that returns boolean, it's named predicate (or pred, for short). They're just following conventions.

Reply

Lukas Eder
April 9th, 2014 at 9:58 am

> *The overloading thing is a non-issue*

I'm more than willing to accept bets on how often such questions will surface Stack Overflow in the near future :-)

Reply

Jake Zim
April 9th, 2014 at 3:01 pm

Maybe so, since it's an easy mistake to make, but we're already prepared with the answer :)

Reply

John Kozlov
April 10th, 2014 at 11:49 pm

Here is a simple example demonstrating how implementing default methods can stop existing code to compile. Suppose they added implementation for the method Collection.isEmpty():

```
public interface Collection {
…

default boolean isEmpty() {
return size() == 0;
}

…
}
```

Then, suppose someone defined an interface A:

```
public interface A {
boolean isEmpty();
}
```

… and interface B:

```
public interface B extends Collection, A {
…
}
```

After migrating to JDK8, the code is no longer valid. The error is: The default method isEmpty() inherited from Collection conflicts with another method inherited from A.

Reply

Lukas Eder
April 11th, 2014 at 11:00 am

Good point, I hadn't thought of that. Do you have a pointer to the EG's decision why B.isEmpty() is not just re-declared abstract again?

Reply

Lukas Eder
April 11th, 2014 at 11:05 am

What would be most interesting is a particular decision-process where this line of thought makes sense (e.g. don't touch Collection) as opposed to where they ran the risk of introducing such regression (e.g. who cares about Iterator.remove())

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

– one = 2

Sign me up for the newsletter!

Post Comment

Notify me of followup comments via e-mail. You can also subscribe without commenting.

KNOWLEDGE BASE

- Courses
- Examples
- Resources
- Tutorials
- Whitepapers

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- Web Code Geeks

HALL OF FAME

- “Android Full Application Tutorial” series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike. JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

DISCLAIMER