DZone

Why There's Interface Pollution in Java 8

by Edwin Dalorzo · Apr. 08, 14 · Java Zone · Not set

I was reading this interesting post about The Dark Side of Java 8. In it, Lukas Edger, the author, mentions how bad it is that in the JDK 8 the types are not simply called functions. For instance, in a language like C#, there is a set of predefined function types accepting any number of arguments with an optional return type (Func and Action each one going up to 16 parameters of different types T1, T2, T3, ..., T16), but in the JDK 8 what we have is a set of different functional interfaces, with *different* names and*different method* names, and whose abstract methods represent a subset of well know function signatures (i.e. nullary, unary, binary, ternary, etc).

The Type Erasure Issue

So, in a way, both languages suffer from some form of interface pollution (or delegate pollution in C#). The only difference is that in C# they all have the same name. In Java, unfortunately, due to type erasure, there is no difference between `Function<T1,T2>` and `Function<T1,T2,T3>` or `Function<T1,T2,T3,...Tn>` , so evidently, we couldn't simply name them all the same way and we had to come up with creative names for all possible types of function combinations.

Don't think the expert group did not struggle with this problem. In the words of Brian Goetz in thelambda mailing list:

[...] As a single example, let's take function types. The lambda strawman offered at devoxx had function types. I insisted we remove them, and this made me unpopular. But my objection to function types was not that I don't like function types — I love function types — but that function types fought badly with an existing aspect of the Java type system, erasure. Erased function types are the worst of both worlds. So we removed this from the design.

But I am unwilling to say "Java never will have function types" (though I

recognize that Java may never have function types.) I believe that in order to get to function types, we have to first deal with erasure. That may, or may not be possible. But in a world of reified structural types, function types start to make a lot more sense [...]

So, how does this affect us as developers? The following is a categorization of some of the most important new functional interfaces (and some old ones) in the JDK 8 organized by function return type and the number of expected arguments in the interface method.

Functions with Void Return Type

In the realm of functions with a void return type, we have the following:

TYPE OF FUNCTION	LAMBDA EXPRESSION	KNOWN FUNCTIONAL INTERFACES
Nullary	`() -> doSomething()`	Runnable
Unary	`foo  -> System.out.println(foo)`	Consumer IntConsumer LongConsumer DoubleConsumer
Binary	`(console,text) -> console.print(text)`	BiConsumer ObjIntConsumer ObjLongConsumer ObjDoubleConsumer
n-ary	`(sender,host,text) -> sender.send(host, text)`	Define your own

Functions with Some Return Type T

In the realm of functions with a return type T, we have the following:

TYPE OF FUNCTION	LAMBDA EXPRESSION	KNOWN FUNCTIONAL INTERFACES
Nullary	`() ->  "Hello World"`	Callable Supplier BooleanSupplier IntSupplier LongSupplier DoubleSupplier
		Function IntFunction LongFunction DoubleFunction IntToLongFunction IntToDoubleFunction

Unary	`n -> n +  1`
	`n -> n >=  0`
		LongToIntFunction
		LongToDoubleFunction
		DoubleToIntFunction
		DoubleToLongFunction
		
		UnaryOperator
		IntUnaryOperator
		LongUnaryOperator
		DoubleUnaryOperator
		
		Predicate
		IntPredicate
		LongPredicate
		DoublePredicate

Binary	`(a,b) -> a > b ?  1 :  0`
	`(x,y) -> x + y`
	`(x,y) -> x % y ==  0`
		Comparator
		BiFunction
		ToIntBiFunction
		ToLongBiFunction
		ToDoubleBiFunction
		
		BinaryOperator
		IntBinaryOperator
		LongBinaryOperator
		DoubleBinaryOperator
		
		BiPredicate

n-ary	`(x,y,z) ->  2  * x + Math.sqrt(y) - z`	Define your own

An advantage of this approach is that we can define our own interface types with methods accepting as many arguments as we would like, and we could use them to create lambda expressions and method references as we see fit. In other words, we have the power to pollute the world with yet even more new functional interfaces. Also we can create lambda expressions even for interfaces in earlier versions of the JDK or for earlier versions of our own APIs that defined SAM types like these. And so now we have the power to use `Runnable` and `Callable` as functional interfaces.

However, these interfaces become more difficult to memorize since they all have different names and methods.

Still, I am one of those wondering why they didn't solve the problem as in Scala, defining interfaces like `Function0` , `Function1` , `Function2` , ..., `FunctionN` . Perhaps, the only argument I can come up with against that is that they wanted to maximize the possibilities of defining lambda expressions for interfaces in earlier versions of the APIs as mentioned before.

Lack of Value Types

So, evidently type erasure is one driving force here. But if you are one of those wondering why we also need all these additional functional interfaces with similar names and method signatures and whose only difference is the use of a primitive type, then let me remind you that in Java we *also* lack of value typeslike those in a language like C#. This means that the generic types used in our generic classes can only be reference types, and not primitive types.

In other words, we can't do this:

```
List< int > numbers = asList( 1 , 2 , 3 , 4 , 5 );
```

But we can indeed do this:

```
List<Integer> numbers = asList( 1 , 2 , 3 , 4 , 5 );
```

The second example, though, incurs in the cost of boxing and unboxing of the wrapped objects back and forth from/to primitive types. This can become really expensive in operations dealing with collections of primitive values. So, the expert group decided to create this explosion of interfaces to deal with the different scenarios. To make things "less worse" they decided to only deal with three basic types: int, long and double.

Quoting the words of Brian Goetz in the lambda mailing list:

More generally: the philosophy behind having specialized primitive streams (e.g., IntStream) is fraught with nasty tradeoffs. On the one hand, it's lots of ugly code duplication, interface pollution, etc. On the other hand, any kind of arithmetic on boxed ops sucks, and having no story for reducing over ints would be terrible. So we're in a tough corner, and we're trying to not make it worse.

Trick #1 for not making it worse is: we're not doing all eight primitive types. We're doing int, long, and double; all the others could be simulated by these. Arguably we could get rid of int too, but we don't think most Java developers are ready for that. Yes, there will be calls for Character, and the answer is "stick it in an int." (Each specialization is projected to ~100K to the JRE footprint.)

Trick #2 is: we're using primitive streams to expose things that are best done in the primitive domain (sorting, reduction) but not trying to duplicate everything you can do in the boxed domain. For example, there's no IntStream.into(), as Aleksey points out. (If there were, the next question(s) would be "Where is IntCollection? IntArrayList? IntConcurrentSkipListMap?) The intention is many streams may start as reference streams and end up as primitive streams, but not vice versa. That's OK, and that reduces the number of conversions needed (e.g., no overload of map for int -> T, no specialization of Function for int -> T, etc.)

We can see that this was a difficult decision for the expert group. I think few would agree that this is cool, and most of us would most likely agree it was necessary.

The Checked Exceptions Issue

There was a third driving force that could have made things even worse, and it is the fact that Java supports two type of exceptions: checked and unchecked. The compiler requires that we handle or explicitly declare checked exceptions, but it requires nothing for unchecked ones. So, this creates an interesting problem, because the method signatures of most of the functional interfaces do not declare to throw any exceptions. So, for instance, this is not possible:

```
Writer out =  new  StringWriter();
Consumer<String> printer = s -> out.write(s);  //oops! compiler error
```

It cannot be done because the `write` operation throws a checked exception (i.e. `IOException`) but the signature of the `Consumer` method does not declare it throws any exception at all. So, the only solution to this problem would have been to create even more interfaces, some declaring exceptions and some not (or come up with yet another mechanism at the language level for exception transparency). Again, to make things "less worse" the expert group decided to do nothing in this case.

In the words of Brian Goetz in the lambda mailing list:

Yes, you'd have to provide your own exceptional SAMs. But then lambda conversion would work fine with them.

The EG discussed additional language and library support for this problem, and in the end felt that this was a bad cost/benefit tradeoff.

Library-based solutions cause a 2x explosion in SAM types (exceptional vs not), which interact badly with existing combinatorial explosions for primitive specialization.

The available language-based solutions were losers from a complexity/value tradeoff. Though there are some alternative solutions we are going to continue to explore — though clearly not for 8 and probably not for 9 either.

In the meantime, you have the tools to do what you want. I get that you prefer we provide that last mile for you (and, secondarily, your request is really a thinly-veiled request for "why don't you just give up on checked exceptions already"), but I think the current state lets you get your job done.

So, it's up to us, the developers, to craft yet even more interface explosions to deal with these in a case-by-case basis:

```
interface IOConsumer<T> {
void accept(T t) throws IOException;
}
static<T> Consumer<T> exceptionWrappingBlock(IOConsumer<T> b) {
return e -> {
try { b.accept(e); }
catch (Exception ex) { throw new RuntimeException(ex); }
};
}
```

In order to do:

```
Writer out = new StringWriter();
Consumer<String> printer = exceptionWrappingBlock(s -> out.write(s));
```

Probably, in the future (maybe JDK 9) when we get Support for Value Types in Java and Reification, we will be able to get rid of (or at least no longer need to use anymore) these multiple interfaces.

In summary, we can see that the expert group struggled with several design issues. The need, requirement or constraint to keep backwards compatibility made things difficult, then we have other important conditions like the lack of value types, type erasure and checked exceptions. If Java had the first and lacked of the other two the design of JDK 8 would probably have been different. So, we all must understand that these were difficult problems with lots of tradeoffs and the EG had to draw a line somewhere and make a decisions.

So, when we find ourselves in the dark side of Java 8, probably we need to remind ourselves that there is a reason why things are dark in that side of the JDK :-)

Topics:

Published at DZone with permission of Edwin Dalorzo . See the original article here.
Opinions expressed by DZone contributors are their own.

DZone Research: The Problems With Java

by Tom Smith STAFF · Apr 30, 18 · Java Zone · Research

To gather insights on the current and future state of the Java ecosystem, we talked to executives from 14 companies. We began by asking, "What are the most common problems with the Java ecosystem today?" Here's what the respondents told us:

Verbosity

- **It’s a bit verbose.** Other languages are simpler. A lot of libraries – hard to choose which one. The JDK is complete but there are a lot of dependencies.
- The most common complaint about Java is that it **tends to be long-winded**, meaning you have to type more characters to get what you want done than some other, more modern languages. That’s something newer languages have been trying to tackle, with the goal of trying to get the quality, features, and power of Java without needing to do quite as much typing. The ability to write code faster is one of the biggest reasons that some people prefer languages that are actually worse languages when compared at side-by-side with Java.
- Java developers have no problems with Java. **Those on the outside will complain that it’s too verbose.** The Java 9 upgrade is not seamless, you must add dependencies to adopt. It will not be supported long term. Java 10 will be out this month and may result in a lot of people skipping the Java 9 upgrade.
- **Java itself is verbose** but there are alternatives like Kotlin, Scala, and project Lombok.

Nothing

- **I think the ecosystem is pretty healthy right now,** relatively speaking. A benevolent dictator and an engaged community is probably the best mix. Feels like good energy stemming from more frequent releases.
- If you had asked me this same question one year ago, I would have suggested that the

uncertainty of Oracle’s intentions around Java EE was a substantial problem. As previously mentioned, Java dominates the enterprise. While we did have folks like IBM, Tomitribe, Red Hat, Payara and others working to create MicroProfile, it was unclear what would happen if Oracle remained neutral. **Now, we have Jakarta EE at Eclipse and we are open for innovation yet again.**

Other

- Software engineering quality is degrading with less care and pride of workmanship. No real engineering processes. **No one is nailing down well-known methodologies.**
- **It can be hard to consume all that’s contained in a release every six months.**
- 1) The Maven repository was revolutionary in its time as a central repository for Java libraries, but it is starting to show its age, especially compared with the npm library. Compare the simplicity of searching for quality libraries in the npm libraries, which includes a great search tool, a way to grade the quality of a library, and a culture of readmes that help you figure out what the library is all about at a glance. 2) **Java libraries tend to be too large** and try to include as much stuff as possible in a library, with, in the best case, huge documentation that is very difficult to comprehend due to the number of things to understand, and to the worst case where all you get is a huge Javadoc. Compare and contrast that with the JS library ecosystem that tends to favor small libraries, a “pick and choose” methodology, which enables easy understanding of a library. 3) The slowness of the evolution of the language is a big advantage, but because it was *too* slow, it has driven many developers, especially thought leaders, away from the language.
- Governance of the JVM. Eclipse is a good steward of open source. **The rate of innovation into the JVM is stalling.** Java 9 represents the last attempt to push technology into the platform. I’m not sure where the product roadmap goes after that. The language goes in circles. Every generation needs their own language. It's nicer to be part of a vibrant ecosystem. Java’s guarantee of compatibility gives you a wide variety of languages to choose from.
- **Participation and engagement,** growing real-world developer interest. We want to hear from every user, not just architects. We encourage developers to contribute as a group. The Brazilian User Groups have adopted the JSR concept to forward feedback as a group. To continue the momentum, we have an open JDK adoption group.
- **The freedom can sometimes become a curse.** In languages such as .Net where you are more "within boundaries" it is easier to not make the wrong decisions. The different permutations of what dependencies you can use together can make your system become an unproven snowflake. There's also the notion of Java being an "old and grumpy" language. Though I don't agree with this notion the rumor can be a bit hurtful. Hopefully, this will change with the new release cadence.
- **It lags behind because it’s used by large enterprises.** With slowness comes stability. It lacks some of the niceties of other languages; however, it provides quick wins with fast coding.
- **The main problem will be release fatigue.** The increase in cadence means developers have to keep up with new versions of Java. So much going on in open source community it’s hard to get a handle on new APIs, components, projects. Every time you try to learn something new you’re placing a bet with your mindshare on whether or not it will be relevant in a couple of years.

Here’s who we spoke to:

- Gil Tayar, Senior Architect and Evangelist, Applitools
- Frans van Buul, Commercial Developer, Evangelist, AxonIQ
- Carlos Sanches, Software Engineer, CloudBees
- Jeff Williams, Co-founder and CTO, Contrast Security
- Doug Pearson, CTO, FlowPlay
- John Duimovich, Distinguished Engineer and Java CTO, IBM
- Brian Pontarelli, CEO, Inversoft
- Wayne Citrin, CTO, JNBridge
- Ray Augé, Sr. Software Architect, Liferay
- Matt Raible, Java Champion and Developer Advocate, Okta
- Heather VanCura, Chair of the Java Community Process Program, Oracle
- Burr Sutter, Director Developer Experience, Red Hat
- Ola Petersson, Software Consultant, Squeed
- Roman Shoposhnik, Co-founder, V.P. Product and Strategy, Zededa

Topics: JAVA, JAVA EE, JAKARTA EE, VERBOSITY