Applause from Michelé Clarke and 11 others

Adrian D. Finlay
@thewiprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ adriandavid.me/network
Mar 16 · 9 min read

New Java 11 Language Features: Local-Variable Type Inference (var) extended to Lambda Expression Parameters

Rasim Muratovic on Google+

In keeping with a commitment to a 6 month release cycle, work on Java 11 has already commenced. As of the publishing of this article, the latest version of the Early Access JDK 11 EA Build is +4. You can grab it here.

Not Caught up to Java 10? Learn more below:

Java 10 Sneak Peek: Local-Variable Type Inference (var)!

Per the rolling release cycle promise, JDK 10 is schedule to be released on March 20th, 2018.

medium.com

Not Caught up to Java 9? Learn more below:

New Language Features in Java 9

Java 9 is here!

medium.com

Cool Java 9 Features You Might've Missed

While the predominant features of Java SE 9 were the modularity efforts of Project Jigsaw, Java 9...

medium.com

What's New in Java FX—Java 9 Updates

JavaFX is a 3rd Generation Java GUI platform (after AWT, Swing) for the development of Desktop an...

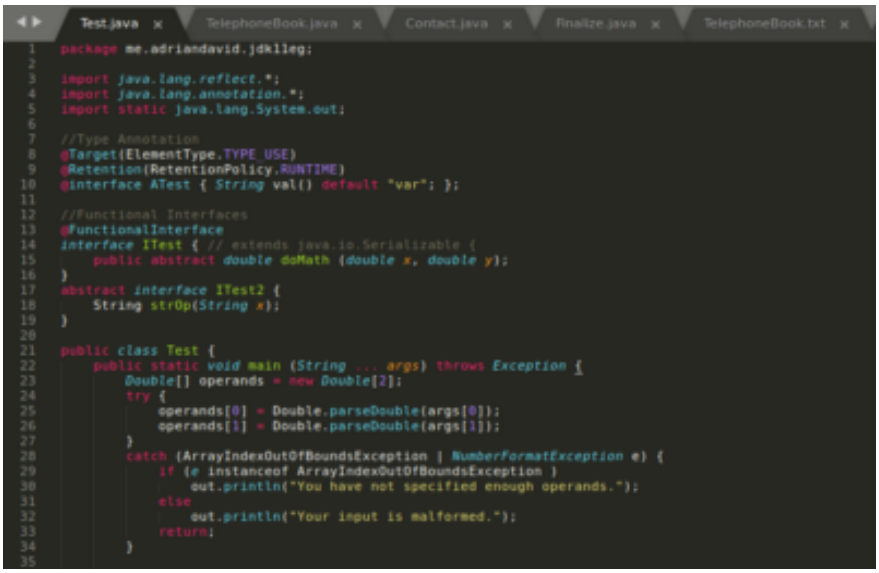
medium.com

So far, the only language feature included in this release is JEP 323: Local-Variable Syntax for Lambda Parameters.

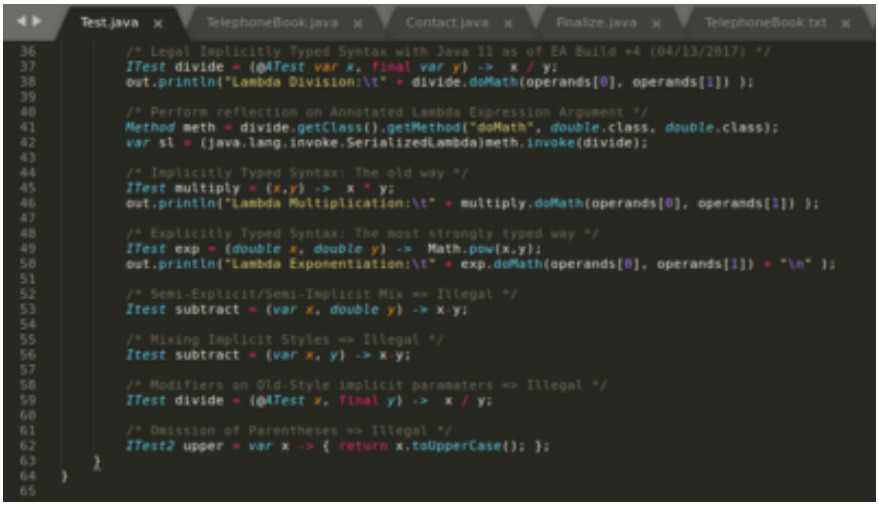
This is a minor update to Local Variable Type Inference introduced in JDK 10. I discuss this in another blurb.

In Java 10, using the var reserved word for type inference was forbidden when declaring the parameter list of implicitly typed lambda expressions. Now, we can do things like this:

```
ITest divide = (var x, var y) ->  x / y;
```

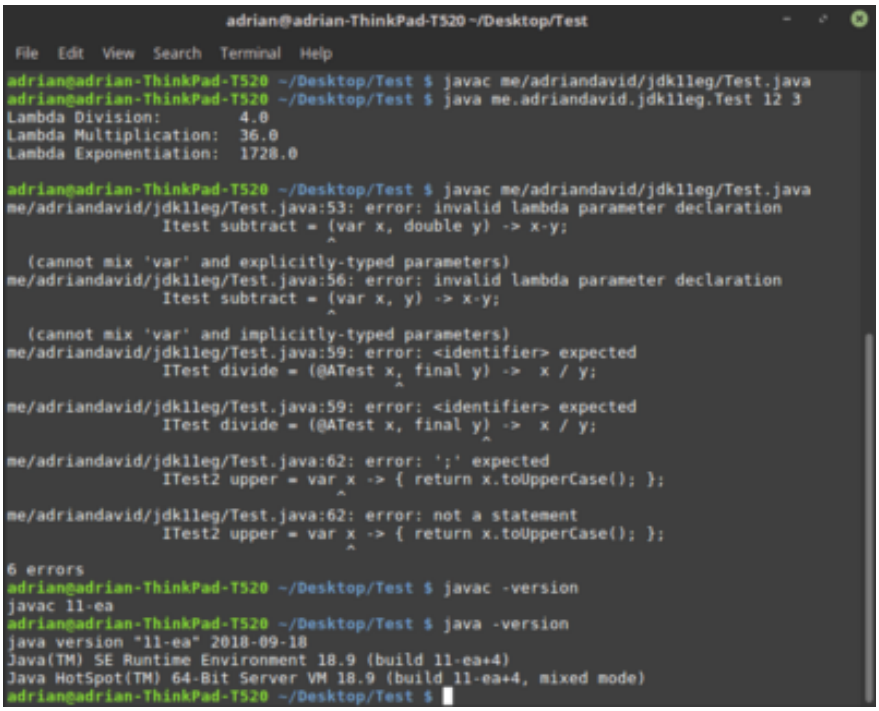



```java
package me.adriandavid.jdk11eg;

import java.lang.reflect.*;
import java.lang.annotation.*;
import static java.lang.System.out;

//Type Annotation
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)
@interface ATest { String val() default "var"; };

//Functional Interfaces
@FunctionalInterface
interface ITest { // extends java.io.Serializable {
    public abstract double doMath (double x, double y);
}
abstract interface ITest2 {
    String strOp(String x);
}

public class Test {
    public static void main (String ... args) throws Exception {
        Double[] operands = new Double[2];
        try {
            operands[0] = Double.parseDouble(args[0]);
            operands[1] = Double.parseDouble(args[1]);
        }
        catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
            if (e instanceof ArrayIndexOutOfBoundsException )
                out.println("You have not specified enough operands.");
            else
                out.println("Your input is malformed.");
            return;
        }
```




```java
        /* Legal Implicitly Typed Syntax with Java 11 as of EA Build +4 (04/13/2017) */
        ITest divide = (@ATest var x, final var y) ->  x / y;
        out.println("Lambda Division:\t" + divide.doMath(operands[0], operands[1]) );

        /* Perform reflection on Annotated Lambda Expression Argument */
        Method meth = divide.getClass().getMethod("doMath", double.class, double.class);
        var sl = (java.lang.invoke.SerializedLambda)meth.invoke(divide);

        /* Implicitly Typed Syntax: The old way */
        ITest multiply = (x,y) ->  x * y;
        out.println("Lambda Multiplication:\t" + multiply.doMath(operands[0], operands[1]) );

        /* Explicitly Typed Syntax: The most strongly typed way */
        ITest exp = (double x, double y) ->  Math.pow(x,y);
        out.println("Lambda Exponentiation:\t" + exp.doMath(operands[0], operands[1]) + "\n" );

        /* Semi-Explicit/Semi-Implicit Mix => Illegal */
        Itest subtract = (var x, double y) -> x-y;

        /* Mixing Implicit Styles => Illegal */
        Itest subtract = (var x, y) -> x-y;

        /* Modifiers on Old-Style implicit paramaters => Illegal */
        ITest divide = (@ATest x, final y) ->  x / y;

        /* Omission of Parentheses => Illegal */
        ITest2 upper = var x -> { return x.toUpperCase(); };
    }
}
```

The Result



Let's Talk about the Source

The expressed goal of this feature is very simple: **allow var to be used to declare the formal parameters of an implicitly typed lambda expression**.

For brevity's sake, I kept the three interfaces and the class in one file. You will likely not do this in production code.

For the most part, this code is pretty straight forward and will be familiar to those who create functional interfaces.

You may not be familiar with Annotations. An Annotation is a programming language element used to provide metadata in a Java application. Annotations cannot directly change the behavior of your code. They are data about data. You can, however, by means of reflection, make decisions in your code based on the metadata stored in a Runtime annotation. Underneath the hood, annotations leverage the plumbing of interfaces. They are, mostly, source code mechanisms. Typically, annotations are read by compiler plug-ins, other developers reading your code, and tools used in testing.

For example. @Override(java.lang.Override) is a common marker annotation used to let other developers working on your code base be

aware that you have overriden a method. If you are interested in how annotations are useful for compiler-plugins, check out the Checker Framework.

Check out my article on Annotations below, should you like to know more.

In this source file we have two interfaces which contain one and only one abstract method. As such, these interfaces are considered to be eligible Functional Interfaces. These are the only interfaces that are compatible with lambda expressions. Notice that this is the only requirement. You may have private, static, and default methods in said interfaces, should you please. If you are unfamiliar with lambda expressions, check out my article on how a typical use case in various programming languages.

Notice that in our example that the annotation is retained until runtime and is designated for type use. This will allow us to (potentially) perform reflection on the annotated member. Also notice the differences between the two functional interfaces.

```
// #1
@FunctionalInterface
interface ITest {
```

```
 public abstract double doMath (double x, double y);
}


//#2
abstract interface ITest2 {
 String strOp(String x);
}
```

The first thing to know about these interfaces is that aside from the method signature and the return type of their abstract methods, these functional interfaces are (for most practical purposes) the same.

In the first interface we annotated it with the @FunctionalInterface (java.lang.FunctionalInterface), which is a marker annotation used to let readers of your code as well as build tools know that the interface is a Functional Interface. Should you attempt to include more than one abstract method in an interface marked with @FunctionalInterface, the compiler will give you a specific warning should you attempt to do this; You will not receive such a warning should you add an additional abstract method with the second example. The second example will still fail only if you attempt to try to implement the interface as a lambda expression. The errors would look something like this:

```
//#1
Error: Unexpected @FunctionalInterface annotation
<interface> is not a functional interface
multiple non-overriding abstract methods found in
interface <interface>


//#2
error: incompatible types: <interface> is not a
functional interface
```

Also notice that in the first example our interface is explicitly marked as abstract. This is redundant, but legal. Interfaces are implicitly abstract. Also note that we have omitted the access modifier (public) and the optional modifier (abstract). This is fine because all non-private,non-static, and non-default interface methods are implicitly marked **public abstract**. This makes sense as an interface as an abstract type is essentially a contract between itself and it's implementing classes.

Implementing an interface means you must override all of it's abstract methods. When you use a lambda expression to implement an interface, the compiler creates an implementation of the interface as well as its abstract method. The actual method that is created by the compiler reflects the implementation that you provide.

Next, we parse command line arguments as double primitives. We wrap the parse operations in an exception handling block to safeguard against any exception that may occur.

Thereafter we make use of Java's new syntax for implicitly typed lambda expressions. Before we address this, we should briefly discuss the difference between implicitly and explicitly typed lambda expressions.

Implicit vs Explicit

```
/* Pre-Java 11*/

// Explicitly Typed
ITest exp = (double x, double y) ->  Math.pow(x,y);

// Implicitly Typed
ITest multiply = (x,y) ->  x * y;
```

The difference deals with whether or not the compiler is tasked with inferring the type of the lambda expression arguments. In the first example we explicitly declare that the variables x and y are of type double. In the second example, we tasked the compiler with inferring that x and y are of type double (it matched the lambda expressions arguments to the parameter list of the abstract method of the interface that the lambda expression is attempting to implement.

When we use var, we are inferring the type. Therefore, Java has two mechanisms for type inference on local lambda expression variables.

```
//New Implicitly Typed Java 11+ Syntax
ITest divide = (var x, var y) ->  x / y;
```

You may have noticed that the preceding example is equivalent to the first implicitly typed example that we discussed. As a matter of fact, you probably don't see the advantage to this method as it is more verbose yet it adds no more clarity as to the identity of the argument's types.

So what's the point anyway?

Good question. In my opinion there are really only two reasons to add this feature.

1. Uniform Local Variable Type-Inference Syntax
2. Ability to add **access modifiers** and **annotations** to Implicitly-Typed Lambda Local Variables

Making the type-inferred (var) syntax consistent across local variables (uniformity) provides the benefit of productivity in that developers will experience less errors due to using var in the wrong places. This is, however, a very insignificant reason to add the feature, in my personal opinion; it's nice but far from game changing. Developers typically are already tasked with learning and remembering many things. Having to remember that you cannot use var in a lambda formal declaration would probably not be a huge encumbrance to most developers.

In my opinion, the second reason is the most significant reason to add the feature. Notice the following examples.

```
// #1 - Legal
ITest divide = (@ATest var x, final var y) ->  x / y;

/* #2 Modifiers on Old-Style implicit paramaters =>
Illegal */
ITest divide = (@ATest x, final y) ->  x / y;
```

Notice that this now enables us to reap the benefits of both A) less verbosity due to type inference and B) ability to use modifiers on type inferred variables.

> *Previously, we could only do one or the other.*

Now we can have both. Previously, if we wanted to use modifiers on lambda formals we had to use explicit syntax. Imagine a choice between the following:

```
// #1 - Pre Java 11
ITest op = (@ATest SomeVeryObnioxiouslyLongClassName x,
final SomeVeryObnioxiouslyLongClassName y, final
SomeVeryObnioxiouslyLongClassName z,) -> .... ;


// #2 - Java 11+
ITest op = (@ATest var x, final y, final z) -> .... ;
```

Having this feature saves you some time and cuts down the verbosity of Java, which as long been considered to be it's bane.

Lastly, with the introduction of the ability to annotate implicitly typed lambda formals, comes the ability to perform reflection on these variables. As you may have noticed in my source, I tried to perform this operation. Unfortunately, I was unsuccessful; I tried many different things. I have read on various posts on Stack Overflow of people not being able to successfully do this in an elegant way. The java.lang.reflect API provides no tools for performing reflection on lambda expression formals. **Should you discover an elegant means by which to this with reflection, please let me know in the comments below!** In theory, should one be able to create an instance of a SerializedLambda one might be able to make use of some of the API in java.lang.reflect.*. However, I could be wrong on this as well. The issue at hand is that the actual method generated by the compiler is not easily accessible to us in a manner that allows us to perform reflection on it. Perhaps this will be addressed in / is the concern of JDK Bug: JDK-8027181.

Other Illegal Scenarios with Lambdas/var

With this feature, comes a few caveats. We have discussed some of them and you may reference them in the bottom half of the source (notice the preceding comments, it will be indicated there).

Mixing Explicit / Implicit Syntax

It is illegal to mix explicit and implicit styles. Lambda Expression formals should either be all implicitly or all explicitly typed. This example will not compile.

```
/* Semi-Explicit/Semi-Implicit Mix => Illegal */
Itest subtract = (var x, double y) -> x-y;
```

Mixing Inferred Syntax Styles

It is illegal to mix inferred styles of declaring Lambda Expression formals. Implicitly typed Lambda Expression formals should take one syntax or the other but not both. This example will not compile.

```
/* Mixing Implicit Styles => Illegal */
 Itest subtract = (var x, y) -> x-y;
```

Omission of Parentheses in single argument

Unlike it's non-var peer, var is treated as if it were explicitly typed. You must include a parentheses enclosing a single argument. This example will not compile.

```
/* Omission of Parentheses => Illegal */
  ITest2 upper = var x -> { return x.toUpperCase(); };

// Legal
  ITest2 upper = x -> { return x.toUpperCase(); };
```

Final Thoughts

All in all, I think this is a useful, non-invasive, low risk feature and one that is deserving in being a part of Java.

Want the source? Grab it here.

afinlay5/Java11VarLambda

Java11VarLambda - Gradle source code repository for Java 11 source code examples posted on...

github.com

For more information on this feature, check out JEP 323.

> *JEP 323: Local-Variable Syntax for Lambda Parameters*

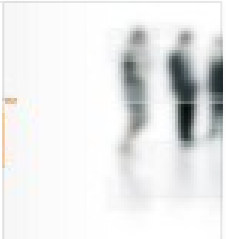
Looney Tunes Ending

Interested in Java? Join my Java group on Facebook:

Join My Java Facebook Group

Interested in Java? Check out my Facebook Group: Java Software Development Group!

medium.com



Like my Content? Subscribe to my mailing list:

Don’t forget to give it a…. ;)

IEmoji.com