

Applause from Cory Althoff, Michelé Clarke, and 14 others



Adrian D. Finlay

@thewipprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ adriandavid.me/network
Oct 12, 2017 · 6 min read

The Java Feature You Never Knew About

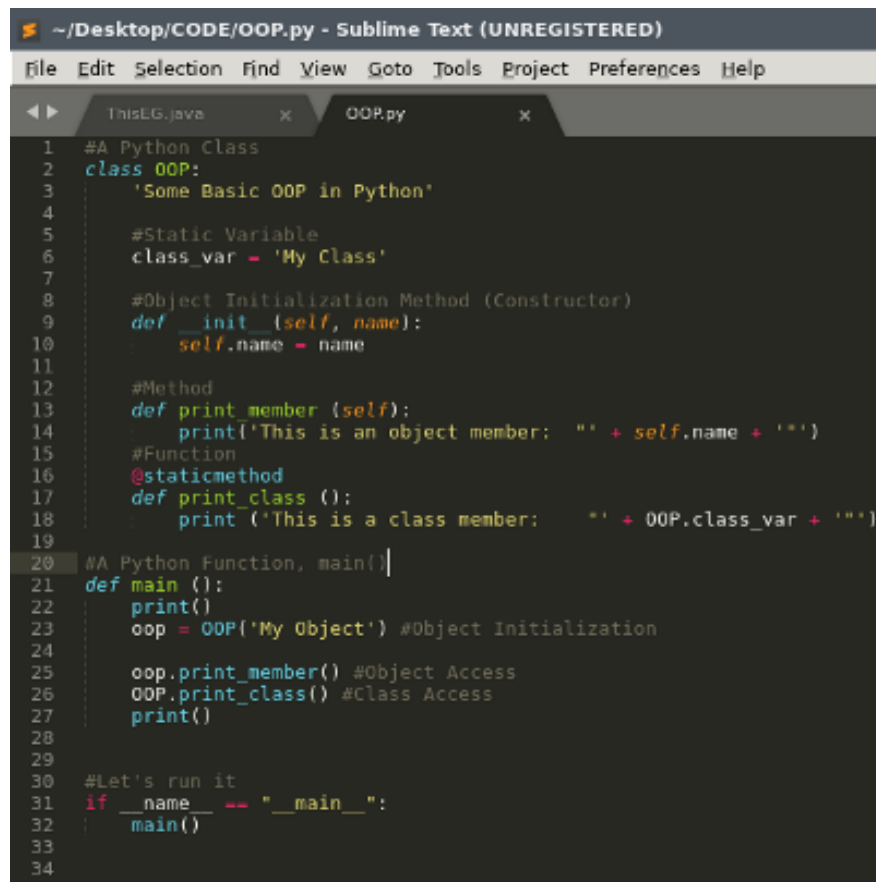


Java 8 Logo from an Oracle Blog [1]

Released on March 18, 2014, Java 8 was generally considered a major feature update, which brought in some functional-ish support to Java —Lambda Expressions, Method References, Functional Interfaces, & the Stream API.

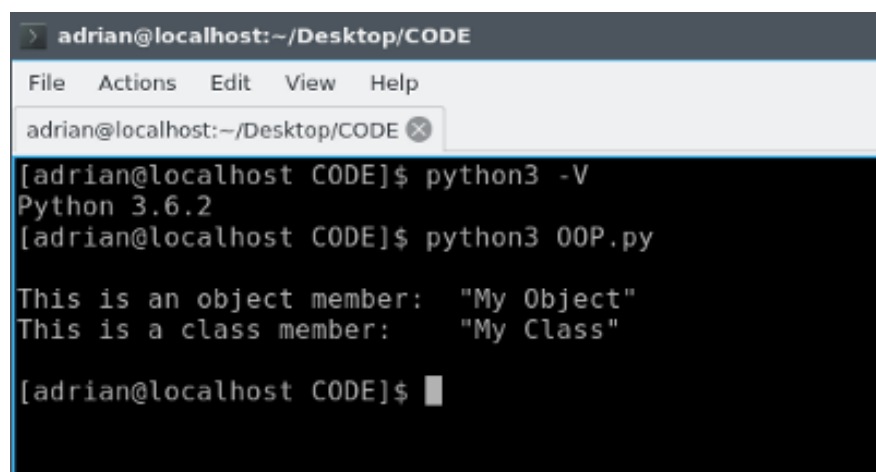
However, Java 8 brought in a language feature that is very familiar to Python developers: The explicit use of the **this** pointer in method arguments. Python devs cordially and typically call it **self**. On the surface this may seem like a triviality as you can already access the **this** pointer inside a Java method body anyway. But there is some use to exposing this, as we shall soon see.

Let's look at some of Python's OOP parlance



```
1 #A Python Class
2 class OOP:
3     'Some Basic OOP in Python'
4
5     #Static Variable
6     class_var = 'My Class'
7
8     #Object Initialization Method (Constructor)
9     def __init__(self, name):
10         self.name = name
11
12     #Method
13     def print_member (self):
14         print('This is an object member: ' + self.name + '')
15
16     #Function
17     @staticmethod
18     def print_class ():
19         print ('This is a class member: ' + OOP.class_var + '')
20
21 #A Python Function, main()
22 def main ():
23     print()
24     oop = OOP('My Object') #Object Initialization
25
26     oop.print_member() #Object Access
27     OOP.print_class() #Class Access
28     print()
29
30 #Let's run it
31 if __name__ == "__main__":
32     main()
33
34
```

And when run gives us,

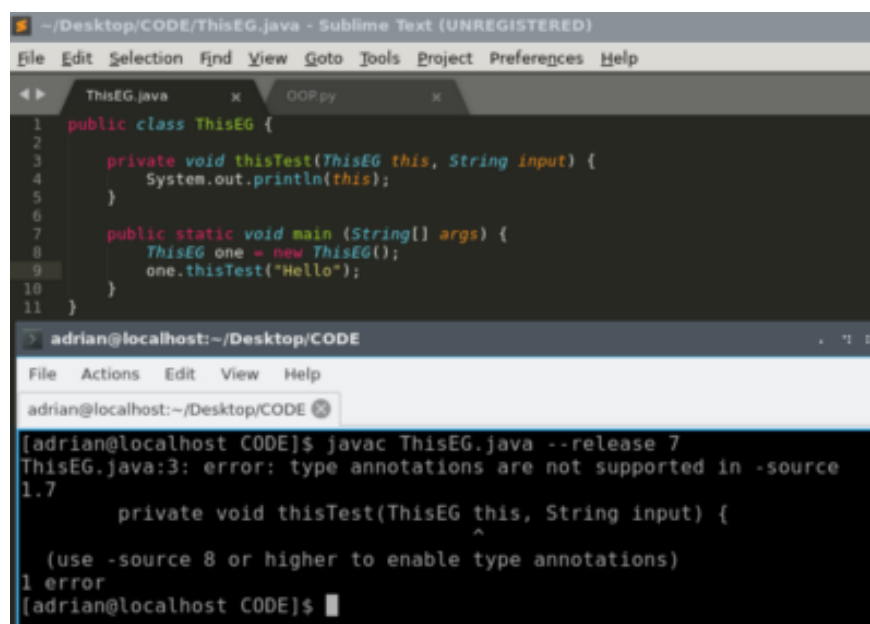


```
> adrian@localhost:~/Desktop/CODE
File Actions Edit View Help
adrian@localhost:~/Desktop/CODE x
[adrian@localhost CODE]$ python3 -V
Python 3.6.2
[adrian@localhost CODE]$ python3 OOP.py
This is an object member: "My Object"
This is a class member: "My Class"
[adrian@localhost CODE]$
```

Python3 via QTerminal on my Fedora Linux Workstation.

One can still grasp this example without previous exposure to Python specifically, provided one has been exposed to OOP in general. The key thing here is that what distinguishes an instance method from a class method in the aforementioned example is the presence of an argument called **self** which we referred to in the object initialization method. The **self** argument is a reference to the instance itself. Consequently, in python, the instance members are (in a sense) “declared” in the `__init__(self, ...)` method definition. While this is not technically correct as python is not statically typed, it is a comparable way to think about it for people who have no background in python. If you already know Python, feel free to ignore my previous remarks. If you know both languages, then you will likely understand why I explained this in the way I did for those familiar with Java. Notwithstanding, the use of **self** has very obvious need in Python development, that is not the case with the implementation of OOP in Java.

Many people who have studied OOP in some depth know that many languages include a pointer to the object in question as the first argument to a method. In most languages offering OOP support this is usually done implicitly. The pointer referring to the method’s invoking object is usually not accessible as a method argument in non-internal code (Application source code). In Java, this is called the *receiver parameter* (`<ClassName> this, ...`). It is always (implicitly) the first argument in a method parameter list. As of Java SE 8, this is now explicitly available to Java programmers.



```
~/Desktop/CODE/ThisEG.java - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

ThisEG.java x OOP.py x
1 public class ThisEG {
2
3 private void thisTest(ThisEG this, String input) {
4     System.out.println(this);
5 }
6
7 public static void main (String[] args) {
8     ThisEG one = new ThisEG();
9     one.thisTest("Hello");
10 }
11 }

adrian@localhost:~/Desktop/CODE
File Actions Edit View Help
adrian@localhost:~/Desktop/CODE
[adrian@localhost CODE]$ javac ThisEG.java --release 7
ThisEG.java:3: error: type annotations are not supported in -source
1.7
    private void thisTest(ThisEG this, String input) {
                           ^
    (use -source 8 or higher to enable type annotations)
1 error
[adrian@localhost CODE]$
```

Why should I care?

As is probably obvious to many readers—we can already make use of the **this** reference within the body of a method, so why is this worth discussing let alone useful? Well, take a look at the Java 7 compiler error. The answer is **Type Annotations**.

The **usefulness** of this is that one (or rather, build tools, IDEs, Enterprise Frameworks, etc.) **can now perform checks on annotated types for the purpose of software quality**. It is important to note that the ability to perform checks is a mainstay of the usefulness of annotations in general. The uniqueness is that this functionality has now been extended to types as opposed to merely declarations, as had previously been the case. This strengthens the ability to catch a variety of errors at compile-time as opposed to run-time, where they are obviously more problematic. Less NullPointerExceptions are a good thing.

JEP 104 was the community effort behind this feature. It might surprise you to discover that while this feature was added to Java in late 2014, the matter was first voted out 8 years earlier, in 2006. JSR 308 describes that process.

*A method's **this** reference (receiver parameter) is one of the types that can now be annotated.*

From the Java Language Specification v.8 and JSR 308,

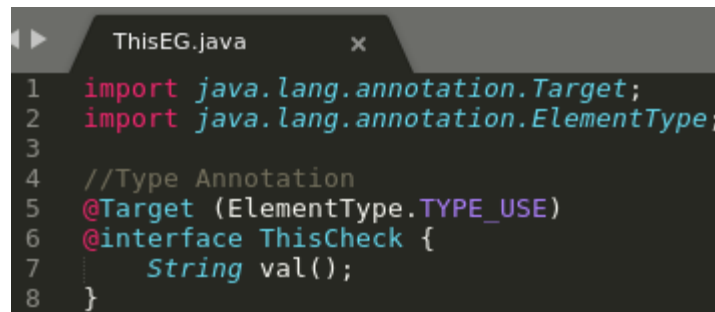
“The *receiver parameter* is an optional syntactic device for an instance method or an inner class’s constructor. For an instance method, the receiver parameter represents the object for which the method is invoked. For an inner class’s constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. Either way, **the receiver parameter exists solely to allow the type of the represented object to be denoted in source code, so that the type may be annotated. The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable (§4.12.3), it is never bound to any value passed as an argument in a method invocation expression or qualified class instance creation expression, and it has no effect whatsoever at run time.**”[2]

“The only purpose of writing the receiver explicitly is to make it possible to annotate the receiver’s type.”[3]

What are Type Annotations?

Type Annotations are simply annotations marking a Java type. They are legal in most but not all places where a type is used. All type annotations must use **ElementType.TYPE_USE** as a target.

Type Annotation Definition, Example Of

A screenshot of a Java code editor window titled 'ThisEG.java'. The code defines a custom annotation '@Target' and an interface 'ThisCheck'.

```
1 import java.lang.annotation.Target;
2 import java.lang.annotation.ElementType;
3
4 //Type Annotation
5 @Target (ElementType.TYPE_USE)
6 @interface ThisCheck {
7     String val();
8 }
```

As of present, Java does not ship with standard Type Annotations. There is a proposal known as JSR 305: Annotations for Software Defect Detection which could bring in standard Type Annotations like @NonNull.

Unfortunately, as of present the project is **dormant**. It was originally proposed to be added to Java SE 7.

To learn more see “Enhanced Metadata—Annotations and Access to Parameter Names” by Alex Buckley and Michael Ernst.

Practical Use—Annotated Receivers

The Source

```

1  import java.lang.annotation.*;
2  import java.lang.reflect.*;
3
4  //Type Annotation
5  @Target (ElementType.TYPE_USE) //So we can use it with Types
6  @Retention(RetentionPolicy.RUNTIME) //So we can perform reflection
7  @interface MutatesMember { //So we can define the annotation
8      boolean truth() default true; //We'll assume it mutates member
9  }
10
11 public class ThisEG {
12
13     private int member =0;
14
15     private void doX (@MutatesMember ThisEG this) { this.member = 9; }
16     private void doY (@MutatesMember(truth = false) ThisEG this ) { }
17
18
19     public static void main (String[] args) throws NoSuchMethodException {
20
21         System.out.println();
22
23         // Perform Reflection to tell us what the result is
24         // The method is private so we need getDeclaredMethod()
25         Method m = ThisEG.class.getDeclaredMethod("doX");
26         Method m2 = ThisEG.class.getDeclaredMethod("doY");
27
28         AnnotatedType a = m.getAnnotatedReceiverType();
29         AnnotatedType a2 = m2.getAnnotatedReceiverType();
30
31         Annotation[] anno = new Annotation[2];
32         anno[0] = a.getDeclaredAnnotation(MutatesMember.class);
33         anno[1] = a2.getDeclaredAnnotation(MutatesMember.class);
34
35         for (Annotation an: anno) { System.out.println(an); }
36         System.out.println();
37     }
38 }

```

The Output

```

adrian@localhost:~/Desktop/CODE
File Actions Edit View Help
adrian@localhost:~/Desktop/CODE
[adrian@localhost CODE]$ java ThisEG
@MutatesMember(truth=true)
@MutatesMember(truth=false)
[adrian@localhost CODE]$

```

How is this useful?

Type Checking. Pluggable type annotations enable third party java compiler plugins and programs to check various qualities of types. One such (and probably the most popular) type checking system is University of Washington's Checker Framework. The author of this framework was behind the JSR that brought Type Annotations into Java SE 8. Java does not (yet, at least) provide a stock type checking framework but you may write pluggable modules that can be used with

javac. This enhances the strength (as well as the marketability) of the Java Platform overall in that there is strong support for checking types.

Why annotate the receiver specifically? Perhaps you may need to impose a restriction at compile-time to check if callers of an instance method are calling said method **on an instance that meets or doesn't meet some condition**. This is one way to do that.

Or perhaps, more specifically, you want to guarantee to clients calling said method that the method does not modify itself in some way. This is one way to check for that. While my example was rather primitive and did not include a pluggable compiler module for checking `@MutatesMember`, it shows the beginnings about how something like this might be done. Try the Checker Framework if you want to see how this works in action.

For more information on this feature, check these out:

The Java Tutorials: Type Annotations and Pluggable Type Systems

Type Annotations Specification (JSR 308) [3]

Java Platform Group, Product Management Blog: Java 8's New Type Annotations

Want the source? Grab it here.

afinlay5/UnknownJavaFt

UnknownJavaFt - Gradle source code repository for the Java 8 examples on personal blog...

github.com





Looney Tunes Ending [4]

Works Cited

[1]—<https://blogs.oracle.com/java/completablefuture-in-java-8>

[2]—Java Language Specification, Java SE 8

[3]—Java Specification Request (JSR) 308: Annotations on Java Types

[4]—https://www.youtube.com/watch?v=0FHEeG_uq5Y

