

Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.



Adrian D. Finlay [Follow](#)

@thewiprogrammer. Lover of learning, coding. Tech Writer, Java aficionado. Proud mango, fishing addict! And much more. Find me @ <http://adriandavid.me/network>

Nov 15, 2017 · 7 min read

New Java 10 Language Feature: Local-Variable Type Inference (var)!



ATC2016 Innovation Lab

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

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



While it is not expected to be a major feature release, so far, it includes a feature that I find invaluable for clearing up java verbosity: **Local-Variable Type Inference!**

Now we can have code like this! :

```
var x = new HashMap<String,Integer>();  
//inferred to be HashMap<String,Integer>
```

Instead of this....

```
HashMap<String,Integer> x = new HashMap<String,Integer> ();
```

Experienced readers will quickly see how this will become useful for shortening the syntactic baggage that has been carried along in Java since it's inception. It doesn't really serve the language much (in my opinion) to mandate typing the return type on the LHS if we can **readily and accurately infer the type from the RHS**. It makes for redundant code, which makes for less developer productivity.

EDIT (03/21/2018):

I want to thank members of the community, namely: [Andriy Drozdyuk](#), [jeswin](#), and [William Bartlett](#) for voicing their pointed concerns and criticisms of my use of the term “strongly typed”. These criticisms and reviews are part of what makes sure that information is presented as accurately as possible. As a consequence of these conversations, I am now of the opinion that the term “strongly typed” is a very poorly defined defined term, and I will personally avoid using it; This opinion, however, has not been expressed by the aforementioned individuals and is expressly my own.

At the cusp of this issue is that the term “strongly typed” is a colloquially understood term that has no universally accepted meaning. This lends to opinion on meaning, which is inherently messy, and rarely ever appropriate for informative purposes despite it’s frequent use in the developer community and programming related publications and media. As such, this codicil was written to address matters of clarity.

The trade-off of this is that this (in my opinion) would make Java code less **explicitly** typed. When considering verbosity, however, it is worth the trade-off, in my opinion.

While this is great for situations where a lot of lasagna code in object instantiation could be avoided, it is perhaps not so good for assigning variables to the result of certain method calls.

Imagine you are working with several methods from various classes (as is the almost ubiquitous case). Is it **immediately** clear where this call is coming from or what it’s return type is?

```
public void doGet() { ... }
public void doPost() { ... }

...

var r = doMath();
//No clue just by looking at this line alone what type this
returns!

//Where did it come from? Not immediately obvious to second
hand readers of your code in large projects.
```

Nope. IDE features, however, help remove a lot of the mystery in quickly figuring out what the returned type will be. Also, how will this mesh with existing Type Inference in Java?

Type Inference with the Diamond Operator

One may think that type inference with the diamond operator cannot be used with local variable type inference but this is not the case. For example

```
Map<String,Integer> x = new HashMap <> ();  
// Note that we do not need to repeat the paramater  
arguments  
// in the RHS diamond operator as it is induced from the  
// RHS return type.
```

```
var x = new HashMap<> ();  
//Legal!
```

Personally, I don't favour this code too much because it is not clear to observers of this code which parameters are instantiated as part of this type. I must admit, however, that this makes for very concise code.

In summa, and in my opinion, var typing can be a good choice in reducing boilerplate when performing **object initialization, working with generics, or working with static methods where the enclosing class is explicitly stated** (Utilities.doX()).

For other situations, such as assigning a variable to the result of a method call (var x = someMeth()), var typing is not desirable, at least in my opinion.

Java was intent-fully designed to be a statically typed language so that the types of members would be known at compile time. Note that this differs from explicit typing.

While Java has become less explicitly typed, it has still retained the properties about it that make it statically typed and type safe, and it is, again, worth it in my opinion, **in the effort to reduce verbosity and improve developer efficiency**.

Some Restrictions

1) Use falling outside of these categories

In Java 10, one would only be able to use Local-Variable Type Inference (var) with: 1) local variables with initializers, 2) for-loop indexes, 3) and within the local scope of a conditional loop. [1]

Notice something about this. var can be used with local variable initialization **only**! Which means you cannot declare a variable as follows:

```
var x; //No!!! -- Terrible code anyway!
```

Notice also that this means **you cannot use var with Methods, Formals, Fields, or anything else other than the aforementioned**. Perhaps Java will add support for var in an upcoming release. Maybe they won't, time will tell. Either way, they will not be legal Java 10 code.

2) You cannot instantiate multiple variables either

```
var x=0,y=0; // no!
```

3) No Lambdas, Method References, or Array Initializers

4) No null values!

```
var x = null; //No!  
var y = funcReturningNull(); //No!  
  
// We cannot infer the type.  
  
var x = (Object) null; // This, however, works ;)  
//Credit to Olivier Gauthier for this observation.
```

5) You cannot reassign to a different type

```
var x=0; //Inferred to be of type 'int'  
var x="c"; //No! - incompatible types!
```

Some Other Things to Know

- Capture variables used with var are mapped to their supertypes. Any type arguments, if specified are replaced with bounded wildcards.

- Some Non-Denotable types (like Anonymous Classes) can be inferred by var.

What is var itself?

One might be tempted to believe that var is a keyword. This is in fact, not true. var in Java will a **reserved type name**. After all, when we type var we are typing it where a type identifier would otherwise be.

The advantage of this is that code that already uses var as the identifier of some member or package won't be affected because **var is not a keyword**.

Grab JDK 10

To get the early the current builds for JDK 10 documentation, visit here:

<p>Java SE Development Kit 10- - Downloads</p> <p>Download JDK 10, a development environment for building applications and components using the... www.oracle.com</p>	
<p>JDK 10.0.1 GA Release</p> <p>This page provides production-ready open-source builds of the Java Development Kit, version 10.0.... jdk.java.net</p>	

To install Java 10 on macOS and Windows is pretty straight forward—download the binary, run it, and make sure the environmental variables (PATH) is set. On linux based Operating Systems you will need to download the files to a folder, add the /bin directory to your path(`export PATH=$PATH:/... to ./profile on bash, for example`) and configure the tools with **update-alternatives—install x** as well as **update-alternatives—config x** where x is the tool in question, for example, *appletviewer*.

When all is done you should get something that looks like this:

```
JDK Installation : bash — Konsole
adrian@localhost:~/Downloads/JDK Installation> javac -version
javac 10-ea
adrian@localhost:~/Downloads/JDK Installation> java -version
java version "10-ea"
Java(TM) SE Runtime Environment (build 10-ea+30)
Java HotSpot(TM) 64-Bit Server VM (build 10-ea+30, mixed mode)
adrian@localhost:~/Downloads/JDK Installation> █
```

\$bash on SUSE Linux. I created a folder with some scripts for automating switching between JDKs and openJDKs.

Let's play with var

Some JShell. Notice that var only works with JDK 10 (of course)!

```
JDK Installation : bash — Konsole
adrian@localhost:~/Downloads/JDK Installation> jshell -version
jshell 10-ea
adrian@localhost:~/Downloads/JDK Installation> jshell
| Welcome to JShell -- Version 10-ea
| For an introduction type: /help intro

jshell> var x = new String("Hello");
x ==> "Hello"

jshell> System.out.println(x.getClass().getName())
java.lang.String

jshell> /exit
| Goodbye
adrian@localhost:~/Downloads/JDK Installation> sudo update-alternatives --config jshell
[sudo] password for root:
There are 2 choices for the alternative jshell (providing /usr/bin/jshell).

  Selection    Path                                          Priority  Status
  ----
*  0            /JDK10_EA_B/JDK/jdk-10/bin/jshell          4        auto mode
    1            /JDK10_EA_B/JDK/jdk-10/bin/jshell          4        manual mode
    2            /usr/local/java/jdk-9/bin/jshell           1        manual mode

Press <enter> to keep the current choice[*], or type selection number: 2
update-alternatives: using /usr/local/java/jdk-9/bin/jshell to provide /usr/bin/jshell (jshell) in manual mode
adrian@localhost:~/Downloads/JDK Installation> jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> var x = new String("Hello");
| Error:
| cannot find symbol
|   symbol:   class var
|   var x = new String("Hello");
|   ^_^

jshell> /exit
| Goodbye
adrian@localhost:~/Downloads/JDK Installation> jshell -version
jshell 9
adrian@localhost:~/Downloads/JDK Installation> █
```

Some compiled examples.

```

1  import static java.lang.System.out;
2
3  /* Yuck! */
4  class Gen { }
5  class Gen1 <Gen> { }
6  class Gen2 <Gen1, String> { }
7  class Gen3 <Gen1, Gen2> { }
8
9  public class VAR {
10     public static void main (String[] args) {
11
12         //Example #1
13         var one = "Type Inference!"; //Inferred to java.lang.String;
14         out.println("\nOur first use of var!:\t\t" + one + "\n");
15         out.println("The variable is of/can cast to type " +
16             ((Object)one).getClass().getName() + ".\n");
17
18         //Example #2
19         var two = 9; //Inferred to int
20         out.println("\nOur second use of var!:\t\t" + two + "\n");
21         out.println("The variable is of/can cast to type " +
22             ((Object)two).getClass().getName() + ".\n");
23

```

```

24         //Example #2
25         var three = 9.9; //Inferred to double
26         out.println("\nOur second use of var!:\t\t" + three + "\n");
27         out.println("The variable is of/can cast to type " +
28             ((Object)three).getClass().getName() + ".\n");
29
30         //Example #4 - The evolution of messy generic instantiation
31         Gen3 <Gen1<Gen>, Gen2<Gen1,String>> gen3_1 = new Gen3 <Gen1<Gen>, Gen2<Gen1,String>> (); //88 Characters
32         Gen3 <Gen1<Gen>, Gen2<Gen1,String>> gen3_2 = new Gen3 <> (); //60 Characters
33         var gen3_3 = new Gen3 <Gen1<Gen>, Gen2<Gen1,String>> (); // 56 Characters
34         var four = new Gen3 <> (); //26 Characters
35         out.println("\nOur fourth use of var!:\t\t" + four + "\n");
36         out.println("The variable is of/can cast to type " +
37             ((Object)four).getClass().getName() + ".\n");
38         out.println("In Java 5-6 this generic code was 88 characters long.");
39         out.println("In Java 7-9 this generic code is 60 characters long.");
40         out.println("In Java 10 this generic code can be 26 characters long.\n");
41
42         // #Example #5 - Common Collections
43         var five = new java.util.ArrayList<String> ();
44         five.add("Hello!");
45         five.add("Hiya!");
46         out.println("\nOur fifth use of var!:\t\t" + five + "\n");
47         out.println("The variable is of/can cast to type " +
48             ((Object)five).getClass().getName() + ".\n");
49     }
50 }

```

```

CODE : bash — Konsole
adrian@localhost:~/Desktop/CODE> javac VAR.java
adrian@localhost:~/Desktop/CODE> java VAR

Our first use of var!:  "Type Inference!"
The variable is of/can cast to type java.lang.String.

Our second use of var!:  "9"
The variable is of/can cast to type java.lang.Integer.

Our second use of var!:  "9.9"
The variable is of/can cast to type java.lang.Double.

Our fourth use of var!:  "Gen3@1ff8b8f"
The variable is of/can cast to type Gen3.

In Java 5-6 this generic code was 88 characters long.
In Java 7-9 this generic code is 60 characters long.
In Java 10 this generic code can be 26 characters long.

Our fifth use of var!:  "[Hello!, Hiya!]"
The variable is of/can cast to type java.util.ArrayList.

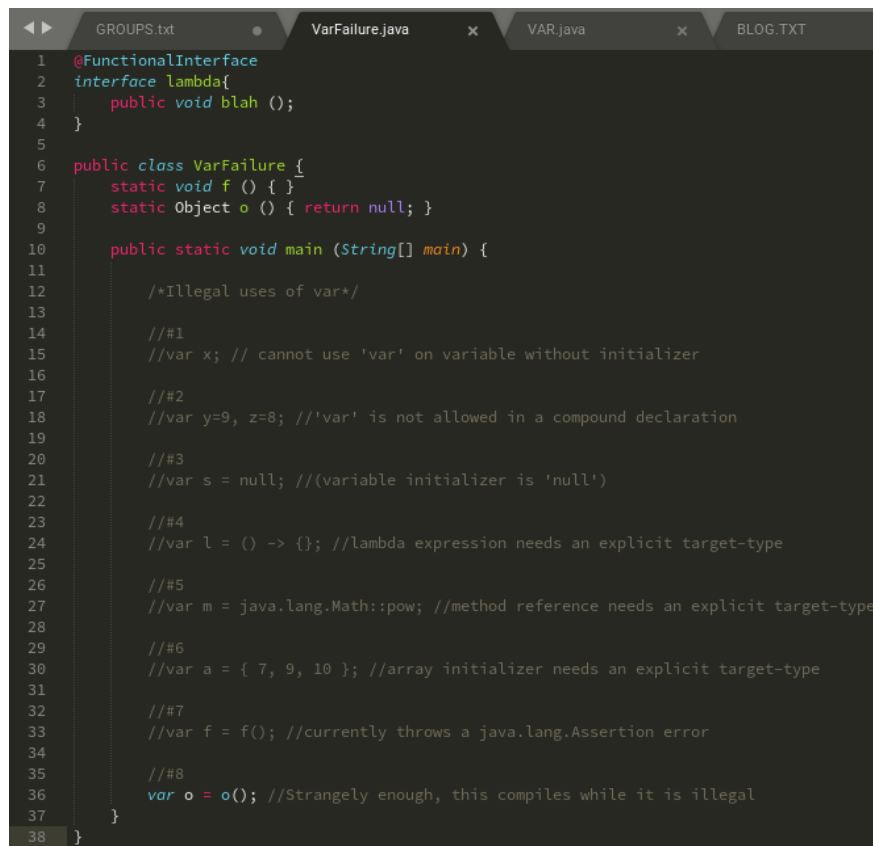
adrian@localhost:~/Desktop/CODE> java -version
java version "10-ea"
Java(TM) SE Runtime Environment (build 10-ea+30)
Java HotSpot(TM) 64-Bit Server VM (build 10-ea+30, mixed mode)
adrian@localhost:~/Desktop/CODE>
CODE : bash

```

For Loop Initializers


```
for (var x=0; x<6; x++) { }
```

Some Current Error Messages of Failures with var



```
1 @FunctionalInterface
2 interface lambda{
3     public void blah ();
4 }
5
6 public class VarFailure {
7     static void f () { }
8     static Object o () { return null; }
9
10    public static void main (String[] main) {
11
12        /*Illegal uses of var*/
13
14        // #1
15        //var x; // cannot use 'var' on variable without initializer
16
17        // #2
18        //var y=9, z=8; //'var' is not allowed in a compound declaration
19
20        // #3
21        //var s = null; //(variable initializer is 'null')
22
23        // #4
24        //var l = () -> {}; //lambda expression needs an explicit target-type
25
26        // #5
27        //var m = java.lang.Math::pow; //method reference needs an explicit target-type
28
29        // #6
30        //var a = { 7, 9, 10 }; //array initializer needs an explicit target-type
31
32        // #7
33        //var f = f(); //currently throws a java.lang.Assertion error
34
35        // #8
36        var o = o(); //Strangely enough, this compiles while it is illegal
37    }
38 }
```

Remember, it cannot be used with methods or classes!

```
var v () { return new Object(); }
//error: 'var' is not allowed here
```

```
var class vc { }
//error: 'var' is not allowed here
```

Where to learn more about Java 10

Curious about the targeted new features in JDK 10? Check them out, [here](#).

Want the source? Grab it here.

afinlay5/Java10Var

Java10Var - Java source code example
demonstrating Local variable Type Inference (var...
github.com



Java 11 Sneak Peek

See how this feature is upgraded in Java 11!

Java 11 Sneak Peek: Local-Variable Type
Inference (var) extended to Lambda Expressio...

In keeping with a commitment to a 6 month
release cycle, work on Java 11 has already...
medium.com



How do you like var with Java 10?



Looney Tunes Ending [4]

Works Cited

[1]—JEP 286: Local-Variable Type Inference

