

Applause from Cory Althoff, Michelé Clarke, and 12 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

Sep 27, 2017 · 7 min read

## New Language Features in Java 9

### Java 9 is here!



Hurrah! [1]

**The Java Standard Edition (Java SE) JDK 9** was released on September 22, 2017. The release notes, describing the new changes, are available [here](#).

The Java Enterprise Edition Specification (Java EE) SDK 8, although not the topic of this article, was released on September 22 as well. The release notes, describing the new changes, are available [here](#).

This article will discuss the **changes to the Java language introduced with Java SE Release 9**. There are **five** of them. The Java Enhancement Proposal 213, known as **JEP 213: Milling Project Coin**

was the community effort behind these changes[2]. I will be running the examples with the Oracle Official Java 9 JDK on my Linux Mint box.

| *Where to get Java 9?*

Java SE Development Kit 9—Downloads

Oracle Official Guide on Installing JDK 9

Notice that while you can download and manually install Java 9 with the tar balls available on Oracle's web page, it's less hassle (and recommended) to install Java 9 via your distribution's package manager. For many distributions, the openJDK may be the only offering in the default package manager, although there are many repositories available across various distributions to install Oracle's official JDK 9.

**For those Running Fedora\SUSE Linux, see this article.**

Normally, you should be able to double click the RPM and install or use rpm or zypper from the command line, but as of recent, that has been giving some problems.

OpenSUSE official installation guide.

**For those Running Debian, Ubuntu, & Mint based systems:** As of about 4 PM on Sept 25, the webupd8Team ppa has been updated in Apt to include the General Availability release of Java 9.

You can install Java 9 by doing the following:

**1)Add the WebUpd8 PPA repository:**

```
sudo add-apt-repository ppa:webupd8team/java
```

**2)Update/Refresh the Package Repository:**

```
sudo apt-get update
```

**3)Install the Oracle Official Java 9 JDK**

```
sudo apt-get install oracle-java9-installer
```

**4) Set the Default Java Version (if needed)**

```
sudo apt install oracle-java9-set-default
```

Understand that this will work if this is the only copy of the JRE/JDK on your system. If you are unsure run “whereis java” or “whereis javac” from the Terminal to figure it out. It is likely that you have at least one other copy of Java on your system as many other pieces of software rely on the JRE, such as LibreOffice, for example. The version of Java you will likely encounter will probably be the openJDK. You do not need to install it, both copies can coexist. If you have more than one copy of the JDK/JRE on your machine, you will want to specify one as the default. In this case, follow the instructions below.

| *What if I still need to use Java 8?*

1. **Figure out which version of Java is the default**

```
sudo update-alternatives—config java
sudo update-alternatives—config javac
sudo update-alternatives—config .....
```

Notice that Medium automatically formats “- -” as “—”. We want two dashes, one after the other in the command. Understand also that these commands will update individual executables only.

2. **Set the Default**

You will be asked to specify by number corresponding to a directory which tool to use.

Java 9 will be located in: **/usr/lib/jvm/java-9-oracle/bin/....**  
Enter the number corresponding with the location above.

3. **Test it**

```
java -version should return java version “9”
javac -version should return javac 9
```

By running these commands you can effectively toggle between Java versions.

Without further adieu, let’s begin.

## #1 Support for Private Interface Methods

It’s now legal to have code like this. This allows non-abstract interface methods (**default**, **private**, **static** methods) to share code.

```
Java9.java
1  import static java.lang.System.out;
2
3  public class Java9 implements PrivateMethDemo {
4      public static void main (String ... args) {
5
6          PrivateMethDemo demo = new Java9 ();
7          demo.demoPrivateMeth();
8      }
9  }
10
11 interface PrivateMethDemo {
12
13     private void privateMeth() {
14         out.println("Hello from a private method.");
15     }
16
17     default void demoPrivateMeth() {
18         privateMeth();
19     }
20 }
```

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE

File Edit View Search Terminal Help

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE \$ java Java9  
Hello from a private method.  
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE \$

Understand that it was illegal to have private methods in interfaces prior to JDK 9. Understand also that private methods can now have a body, unlike abstract methods. This is what would happen in Java 8.

```
Java9.java  Java8.java
1  import static java.lang.System.out;
2
3  public class Java8 implements PrivateMethDemo {
4      public static void main (String ... args) {
5
6          PrivateMethDemo demo = new Java8 ();
7          demo.denoPrivateMeth();
8      }
9  }
10
11 interface PrivateMethDemo {
12
13     private void privateMeth() {
14         out.println("Hello from a private method.");
15     }
16
17     default void demoPrivateMeth() {
18         privateMeth();
19     }
20 }
```

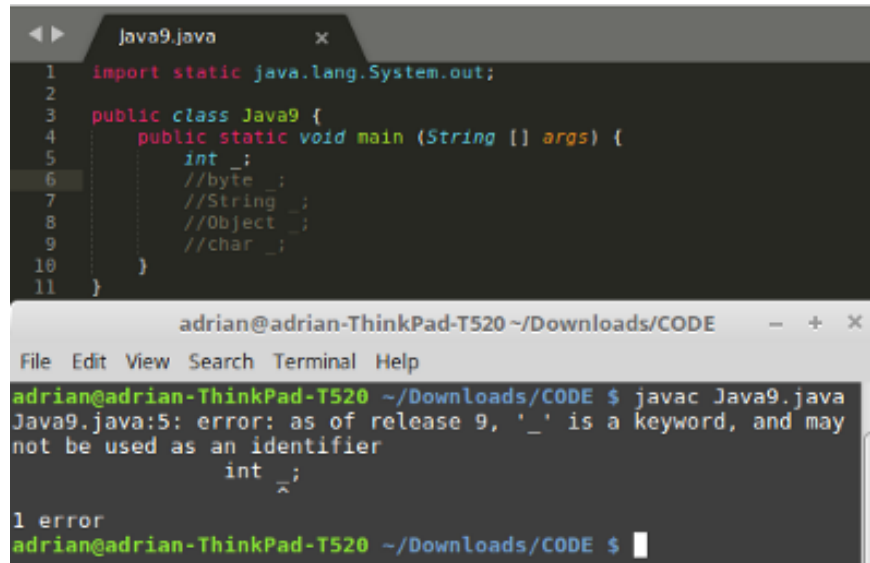
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE

File Edit View Search Terminal Help

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE \$ javac Java8.java  
Java8.java:13: error: modifier private not allowed here  
 private void privateMeth() {  
 ^  
Java8.java:3: error: Java8 is not abstract and does not override abstract method privateMeth() in PrivateMethDemo  
public class Java8 implements PrivateMethDemo {  
 ^  
Java8.java:13: error: interface abstract methods cannot have body  
 private void privateMeth() {  
 ^  
3 errors  
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE \$

## #2 Elimination of Underscore by itself as a valid identifier

Farewell, `_`, we will not miss you.



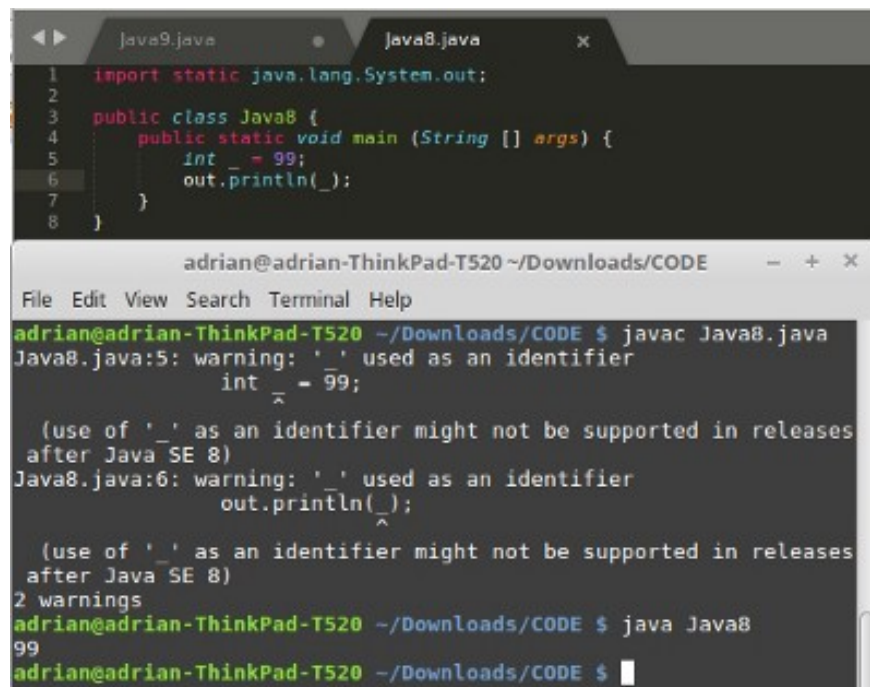
The screenshot shows an IDE window with a file named `Java9.java`. The code is as follows:

```
1 import static java.lang.System.out;
2
3 public class Java9 {
4     public static void main (String [] args) {
5         int _;
6         //byte _;
7         //String _;
8         //Object _;
9         //char _;
10    }
11 }
```

Below the code editor, a terminal window shows the command `javac Java9.java` and the resulting error:

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
Java9.java:5: error: as of release 9, '_' is a keyword, and may
not be used as an identifier
    int _;
       ^
1 error
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

Contrast that to the situation in Java 8.



The screenshot shows an IDE window with two files: `Java9.java` and `Java8.java`. The `Java8.java` file is selected and contains the following code:

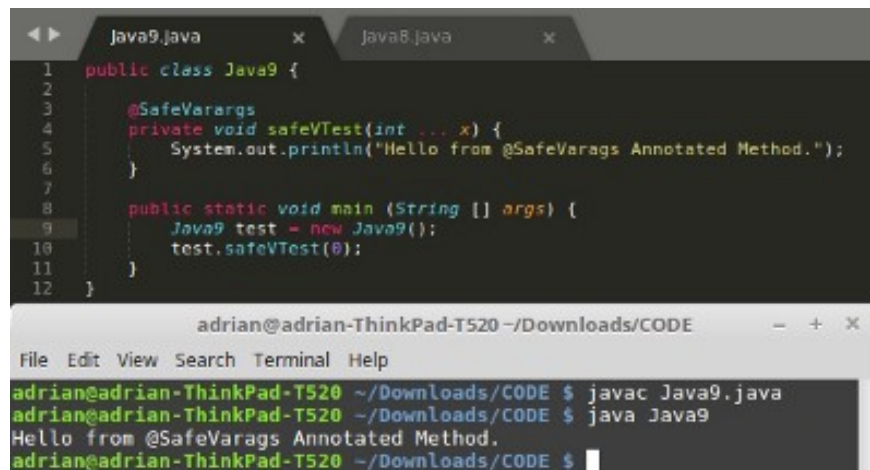
```
1 import static java.lang.System.out;
2
3 public class Java8 {
4     public static void main (String [] args) {
5         int _ = 99;
6         out.println(_);
7     }
8 }
```

The terminal window shows the command `javac Java8.java` and the resulting warnings:

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java8.java
Java8.java:5: warning: '_' used as an identifier
    int _ = 99;
       ^
  (use of '_' as an identifier might not be supported in releases
  after Java SE 8)
Java8.java:6: warning: '_' used as an identifier
    out.println(_);
               ^
  (use of '_' as an identifier might not be supported in releases
  after Java SE 8)
2 warnings
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ java Java8
99
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

## #3 `@SafeVarargs` is now legal on private instance methods

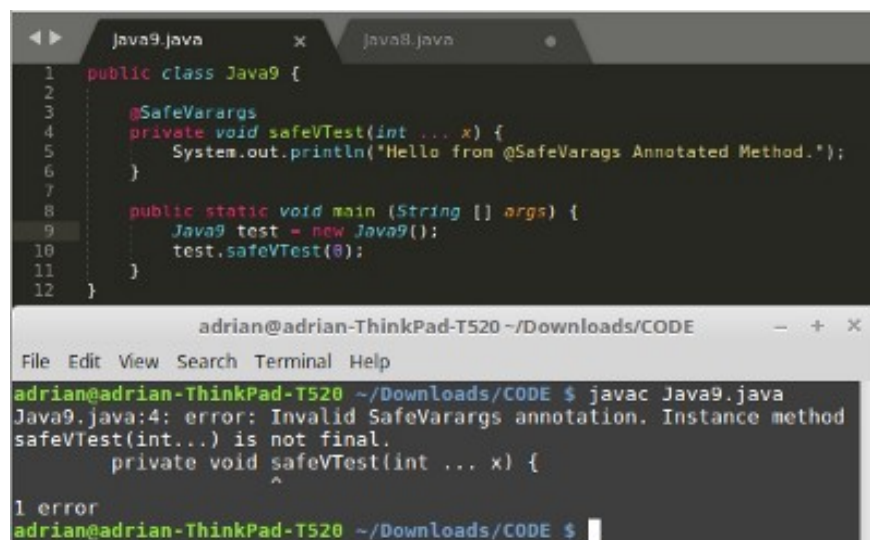
In Java 9, we can now do this.



```
1 public class Java9 {
2
3     @SafeVarargs
4     private void safeVTest(int ... x) {
5         System.out.println("Hello from @SafeVarargs Annotated Method.");
6     }
7
8     public static void main (String [] args) {
9         Java9 test = new Java9();
10        test.safeVTest(0);
11    }
12 }
```

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE
File Edit View Search Terminal Help
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ java Java9
Hello from @SafeVarargs Annotated Method.
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

Prior to JDK 9, methods annotated with `@SafeVarargs` must be methods that cannot be overridden such as static methods & final instance methods.



```
1 public class Java9 {
2
3     @SafeVarargs
4     private void safeVTest(int ... x) {
5         System.out.println("Hello from @SafeVarargs Annotated Method.");
6     }
7
8     public static void main (String [] args) {
9         Java9 test = new Java9();
10        test.safeVTest(0);
11    }
12 }
```

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE
File Edit View Search Terminal Help
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
Java9.java:4: error: Invalid SafeVarargs annotation. Instance method
safeVTest(int...) is not final.
    private void safeVTest(int ... x) {
                  ^
1 error
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

#### #4 Effectively final variables are legal with try-with-resources statements

In Java 9, we can finally use existing resources (any object that implements `java.lang.AutoCloseable` can be used as a resource) that were declared as final **outside of the try-with-resource statement** as follows.

```
1 import java.io.PrintWriter;
2 public class Java9 {
3     public static void main (String[] args) {
4
5         final PrintWriter writer = new PrintWriter(System.out);
6         try (writer) {
7             writer.println("Hello");
8         }
9         catch (Exception e) {}
10    }
11 }
```

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE
File Edit View Search Terminal Help
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ java Java9
Hello
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

Prior to JDK 9, the resource used would have had to be declared and instantiated inside of try (...). What commonly ended up happening is that resources were declared outside of the try block and References were made to said resource within the try block. This led to some long, aesthetically unpleasant, & unnecessary code.

```
1 import java.io.PrintWriter;
2 public class Java9 {
3     public static void main (String[] args) {
4
5         final PrintWriter writer = new PrintWriter(System.out);
6         try (writer) {
7             writer.println("Hello");
8         }
9         catch (Exception e) {}
10    }
11 }
```

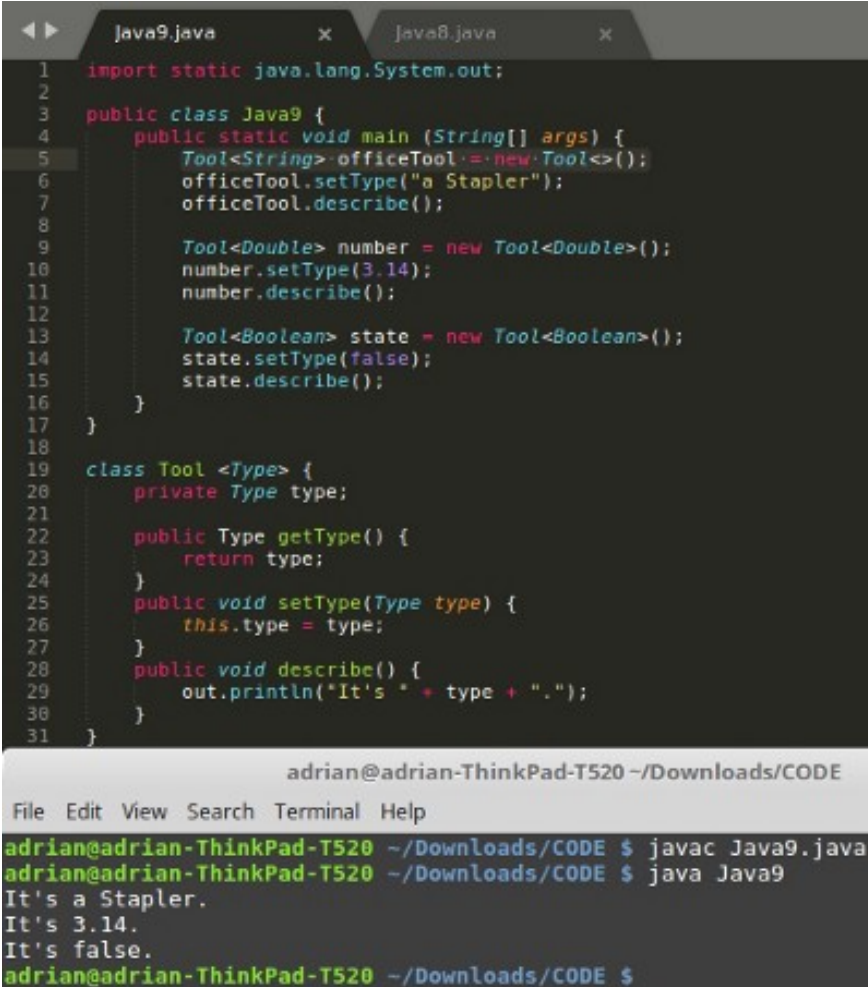
```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE
File Edit View Search Terminal Help
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
Java9.java:6: error: <identifier> expected
    try (writer) {
        ^
Java9.java:6: error: ')' expected
    try (writer) {
        ^
Java9.java:6: error: '{' expected
    try (writer) {
        ^
Java9.java:7: error: illegal start of expression
        writer.println("Hello");
        ^
Java9.java:7: error: ';' expected
        writer.println("Hello");
        ^
5 errors
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

**#5 The Diamond Operator is now legal with Anonymous Classes if the Argument Type of the Inferred Type is a Denotable Type**



In Java, **Denotable Types** are types that you write in the Java programming language (byte, String, int, and so on.). **Non-denotable types** are types only known to the compiler. It is important to note that the denotable types in discussion will be **Reference Types** as you cannot, of course, use primitive types with Generic Type parameters. The object wrappers (Integer, Double, and so on.) can be used in the stead of primitives should you need such functionality.

**Some Foreground:** Java Standard Edition Release 7 brought Generic Type Inference. Generic Type Inference enabled the compiler, when executing the constructor to **infer the type of the object based on the types declared in the reference** to the object. This was mostly done to short what could be very verbose code. Take for example this bit of code:



```
1  import static java.lang.System.out;
2
3  public class Java9 {
4      public static void main (String[] args) {
5          Tool<String> officeTool = new Tool<>();
6          officeTool.setType("a Stapler");
7          officeTool.describe();
8
9          Tool<Double> number = new Tool<Double>();
10         number.setType(3.14);
11         number.describe();
12
13         Tool<Boolean> state = new Tool<Boolean>();
14         state.setType(false);
15         state.describe();
16     }
17 }
18
19 class Tool <Type> {
20     private Type type;
21
22     public Type getType() {
23         return type;
24     }
25     public void setType(Type type) {
26         this.type = type;
27     }
28     public void describe() {
29         out.println("It's " + type + ".");
30     }
31 }
```

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE

File Edit View Search Terminal Help

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ java Java9
It's a Stapler.
It's 3.14.
It's false.
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

Most of this code is unimportant to the point. The portion to focus on is the highlighted line ( line 5) and line 9. Notice that we make use of



generic type inference in line 5 when we say “= new **Tool**<> ();” and omit the type parameter <String> within the diamond operator. Imagine if you had 5 type parameters and many constructor arguments? It’d be a long mess. The community decided that it was redundant to repeat the type parameters in the constructor call. Instead, the compiler can now infer the types, saving programmers some time. Type inference leads to less obfuscated code.

Now let’s examine the situation with anonymous classes. Here’s the subtlety. **The inferred type of an anonymous class’s constructor may not be within the types supported by its signature [3]. This is a bit of Compiler magic. Remember that Anonymous Classes are created by the compiler.** Consequently, an attempt of type inference may result in a non-denotable type.

Now the problem should be obvious. One way to remedy this would be to introduce language mechanisms to support compiler types (non-denotable types) in the class file signature. That would be a mess. Instead, provisions were made in Java SE 9 to allow the use of the diamond operator **if the inferred type was denotable.**

This was the situation in Java 8.

```
Java9.java x java8.java
1 import static java.lang.System.out;
2
3 public class Java9 {
4     public static void main (String[] args) {
5         Automobile<String> car = new Automobile <>() {
6             public void make () { out.println ("Honda"); }
7             public void model () { out.println ("Accord"); }
8             public void year () { out.println ("2006"); }
9         };
10
11         car.make();
12         car.model();
13         car.year();
14     }
15 }
16
17 interface Automobile <Car> {
18     public void make ();
19     public void model ();
20     public void year ();
21 }
22
```

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE

File Edit View Search Terminal Help

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
Java9.java:5: error: cannot infer type arguments for Automobile<Car>
    Automobile<String> car = new Automobile <>() {
                                   ^
    reason: cannot use '<>' with anonymous inner classes
    where Car is a type-variable:
      Car extends Object declared in interface Automobile
1 error
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

In Java 9, we can now do this:

```
Java9.java x java8.java
1 import static java.lang.System.out;
2
3 public class Java9 {
4     public static void main (String[] args) {
5         Automobile<String> car = new Automobile <>() {
6             public void make () { out.println ("Honda"); }
7             public void model () { out.println ("Accord"); }
8             public void year () { out.println ("2006"); }
9         };
10
11         car.make();
12         car.model();
13         car.year();
14     }
15 }
16
17 interface Automobile <Car> {
18     public void make ();
19     public void model ();
20     public void year ();
21 }
22
```

adrian@adrian-ThinkPad-T520 ~/Downloads/CODE

File Edit View Search Terminal Help

```
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ javac Java9.java
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $ java Java9
Honda
Accord
2006
adrian@adrian-ThinkPad-T520 ~/Downloads/CODE $
```

**Want the source? Grab it here.**

afinlay5/Java9Lang

Gradle source code repository for Java 9 source code examples posted on personal blog...

github.com

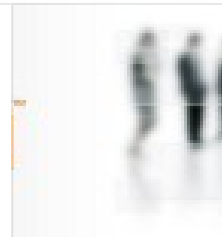


**Interested in Java? Join my Java group on Facebook:**

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:**

This embedded content is from a site that does not comply with the Do Not Track (DNT) setting now enabled on your browser.

Please note, if you click through and view it anyway, you may be tracked by the website hosting the embed.

**[Learn More about Medium's DNT policy](#)**

**Don't forget to give it a.... ;)**



lEmoji.com

#### **Works Cited**

[1]—<http://www.theyale.ca/lets-yale-hurrah/>

[2]—<http://openjdk.java.net/jeps/213>

[3]—Java Community Process: JSR-034 Final Draft Specification



