

Applause from Michelé Clarke, Tariq Ellis, and 2 others



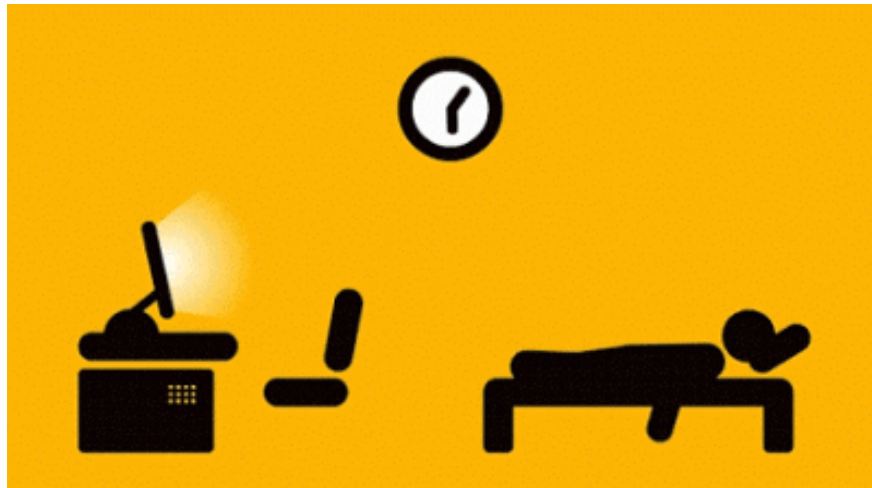
Adrian D. Finlay

@thewipprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ [adriandavid.me/network](https://adriandavid.me/network)

Oct 14, 2017 · 6 min read

## Constructor References in Java (& Method References too)

```
void code () {
```



Programming Geek's Google +

```
}
```

```
while (!adrian.isDead()){  
    adrian.code();  
}
```

JDK 8 saw the advent of a special feature: **Constructor Method References**.

## Some Foreground (Method References)

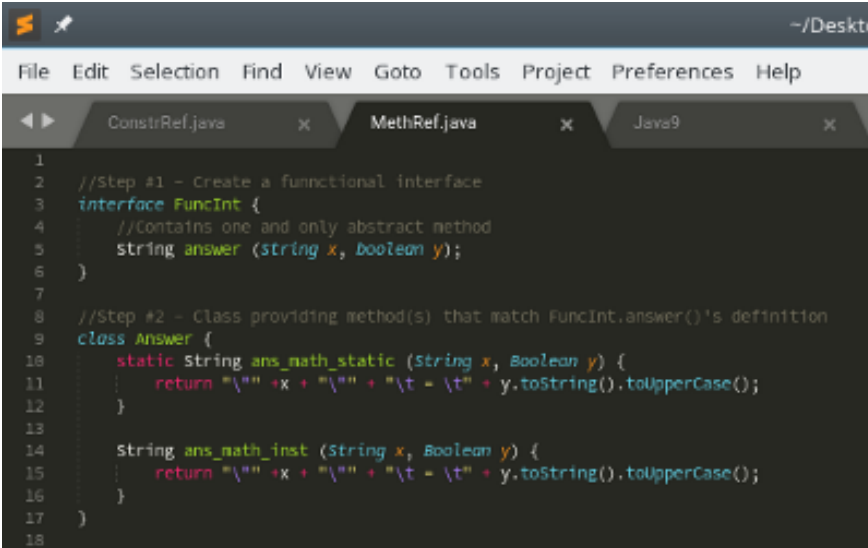
If you didn't already know, Java Constructors are, themselves, special methods. As such, it will benefit the reader to see a basic example of a

method reference, and through understanding this, understand what Constructor References are.

“Method references provide easy-to-read lambda expressions for methods that already have a name.[2]” “They provide an easy way to refer to a method without executing it. [1]” Method references can refer to both static and instance methods and can be generic. Method References are instances of a functional interface. While Lambda Expressions allow you to create method implementations on the fly, often times one ends up calling another method inside the lambda expression to fulfill what we want done. A more direct way to do this is to use a method reference; This is useful in circumstances where you already have a method that fulfills the implementation of the Functional Interface.

Let’s look at an example with static & instance methods.

Source Pt. I—Functional Interface & Class with Methods that can implement a Method Reference.

A screenshot of an IDE window showing a Java file named 'MethRef.java'. The code defines a functional interface 'FuncInt' with a single abstract method 'answer' that takes a 'String x' and a 'boolean y' and returns a 'String'. Below the interface, a class named 'Answer' is defined. It contains two methods: a static method 'ans\_math\_static' and an instance method 'ans\_math\_inst', both of which implement the 'FuncInt.answer' method by returning a formatted string. The IDE's menu bar includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. The tab bar shows 'ConstrRef.java', 'MethRef.java', and 'Java9'. The code is as follows:

```
1
2 //Step #1 - Create a functional interface
3 interface FuncInt {
4     //Contains one and only abstract method
5     String answer (String x, boolean y);
6 }
7
8 //Step #2 - Class providing method(s) that match FuncInt.answer()'s definition
9 class Answer {
10
11     static String ans_math_static (String x, Boolean y) {
12         return "\"" + x + "\"" + "\t = \t" + y.toString().toUpperCase();
13     }
14
15     String ans_math_inst (String x, Boolean y) {
16         return "\"" + x + "\"" + "\t = \t" + y.toString().toUpperCase();
17     }
18 }
```

Source Pt. II—Class Using Functional Interface Instance (Method Reference)

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

ConsolidRef.java x MethRef.java x Java9 x

20 //Step #1 - Class making use of method reference
21 public class MethRef {
22
23     // Specify the Functional Interface as parameter to a method
24     // A Method reference is a valid FuncInt reference
25     static String useMeth (String query, int qnum, FuncInt fi, String expr, Boolean ans) {
26         return query + qnum + ":\t" + fi.answer(expr,ans);
27     }
28
29     public static void main (String [] args) {
30         System.out.println();
31
32         //Remember, a Method Reference is an instance of a functional interface. Therefore...
33         FuncInt func_stat = Answer::ans_math_static;
34         FuncInt func_inst = new Answer()::ans_math_inst;
35
36         //Example #1
37         String result = useMeth("Query #1,Answer::ans_math_static, "9 > 11 ?", false);
38         System.out.println(result);
39
40         //Example #2
41         String result2 = useMeth("Query #2,Answer::ans_math_static, "987.6 < 1.1 ?", false);
42         System.out.println(result2);
43
44         //Example #3
45         String results = useMeth("Query #3,func_stat, "1 > 0.9 ?", true);
46         System.out.println(results);
47
48         //Example #4
49         System.out.println("Query #4 -> " + "\t" + func_stat.answer("T/F: Is Chengdu in Sichuan? ", true));
50
51         //Example #5
52         String results = useMeth("Query #5,func_inst, "-1 % 0.2 = 0 ?", false);
53         System.out.println(results);
54
55         //Example #6
56         System.out.println("Query #6 -> " + "\t" + func_inst.answer("T/F: Does Dwyane Wade play for the Knicks? ", false));
57
58         System.out.println();
59     }
60 }
```

Output

```
CODE: bash — Konsole
File Edit View Bookmarks Settings Help

adrian@localhost:~/Desktop/CODE> javac MethRef.java
adrian@localhost:~/Desktop/CODE> java MethRef

Query #1:      "9 > 11 ?"      =      FALSE
Query #2:      "987.6 < 1.1 ?"  =      FALSE
Query #3:      "1 > 0.9 ?"      =      TRUE

Query #4:      "T/F: Is Chengdu in Sichuan? " =      TRUE
Query #5:      "-1 % 0.2 = 0 ?"  =      FALSE

Query #6:      "T/F: Does Dwyane Wade play for the Knicks? " =      FALSE

adrian@localhost:~/Desktop/CODE> █

CODE: bash
```

The steps for making use of method references are essentially:

- #1 Define a Functional Interface
- #2 Define a method that meets the requirements of the Functional Interface's abstract method
- #3 Instantiate an instance of the Functional Interface with a method

reference to the method defined in step #2. (x :: y)

#4 Use Functional Interface instance: Instance.AbstractMethod();

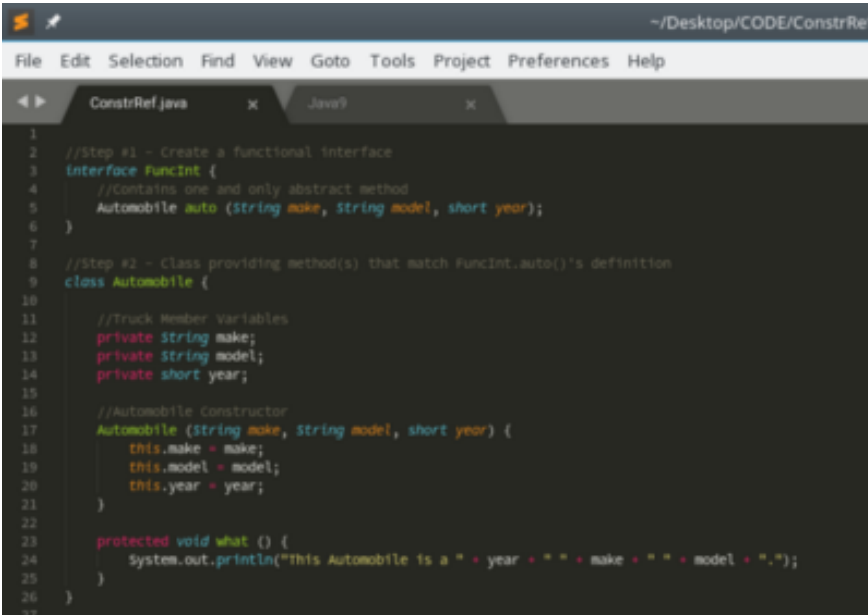
This gives a way to create pluggable instances of methods. Lambda Expression and Method References bring a Functional Aspect to Java Programming.

## Constructor Method References

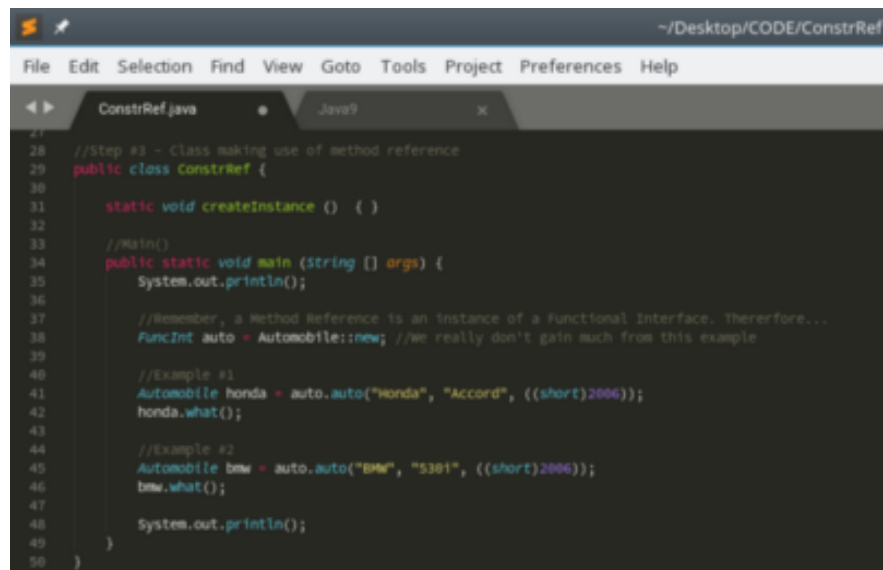
Let's get down to the meat and potatoes.

Constructors are methods just like any other. Right? Wrong. They're a bit special—they're object initialization methods. Nevertheless, they are still a method, and there is nothing stopping us from making Constructor Method References like any other method reference.

A Basic Example

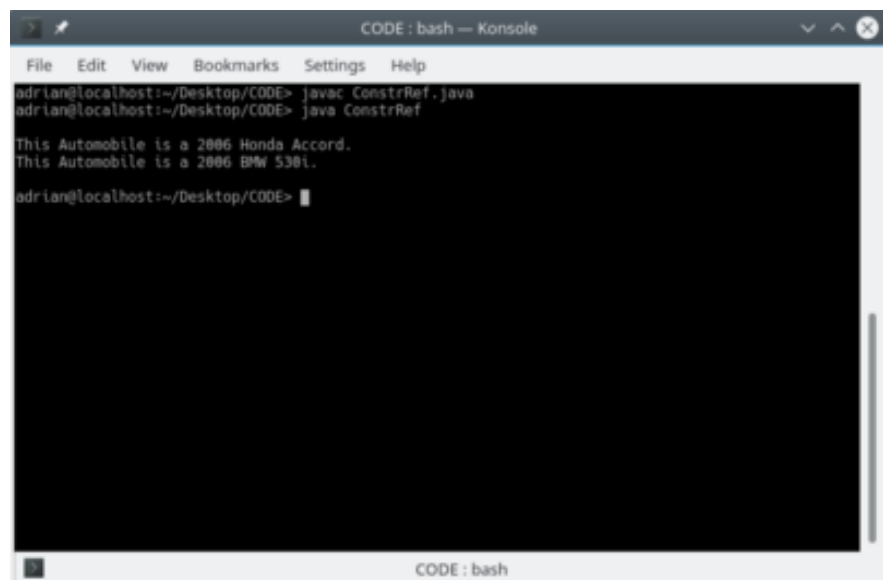


```
1 //Step #1 - Create a functional interface
2 interface FuncInt {
3     //Contains one and only abstract method
4     Automobile auto (String make, String model, short year);
5 }
6
7
8 //Step #2 - Class providing method(s) that match FuncInt.auto()'s definition
9 class Automobile {
10
11     //Track Member Variables
12     private String make;
13     private String model;
14     private short year;
15
16     //Automobile Constructor
17     Automobile (String make, String model, short year) {
18         this.make = make;
19         this.model = model;
20         this.year = year;
21     }
22
23     protected void what () {
24         System.out.println("This Automobile is a " + year + " " + make + " " + model + ".");
25     }
26 }
27
```



```
27
28 //Step #3 - Class making use of method reference
29 public class ConstrRef {
30
31     static void createInstance () { }
32
33     //Main()
34     public static void main (String [] args) {
35         System.out.println();
36
37         //Remember, a Method Reference is an instance of a Functional Interface. Therefore...
38         FuncInt auto = Automobile::new; //We really don't gain much from this example
39
40         //Example #1
41         Automobile honda = auto.auto("Honda", "Accord", ((short)2006));
42         honda.what();
43
44         //Example #2
45         Automobile bmw = auto.auto("BMW", "530i", ((short)2006));
46         bmw.what();
47
48         System.out.println();
49     }
50 }
```

Output



```
CODE: bash — Konsole
File Edit View Bookmarks Settings Help
adrian@localhost:~/Desktop/CODE> javac ConstrRef.java
adrian@localhost:~/Desktop/CODE> java ConstrRef

This Automobile is a 2006 Honda Accord.
This Automobile is a 2006 BMW 530i.
adrian@localhost:~/Desktop/CODE> █

CODE: bash
```

### Explanation:

The first thing that should be obvious to the user is that this basic example is not that useful. It is a rather roundabout way to create an instance of an object. Practically speaking, you almost certainly wouldn't go through all this trouble to create an instance of an Automobile, but for conceptual completeness, it is included here.

The steps for making use of constructor method references are essentially:

#1 Define a Functional Interface with an abstract method whose return type is the same as the Object with which you intend to make a Constructor Reference

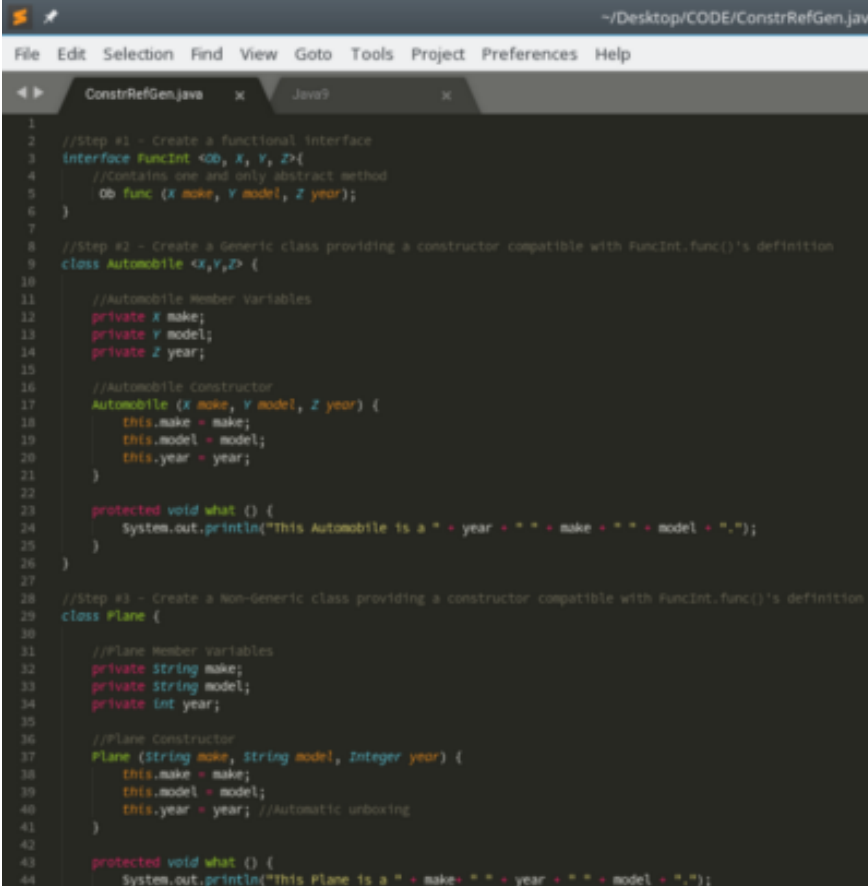
#2 Create a class with a constructor that matches the Functional Interface's abstract method

#3 Instantiate an instance of the Functional Interface with a method reference to the constructor defined in step #2. (x :: y)

#4 Instantiate an instance of the class in step #2 using the constructor reference such that MyClass x = ConstructorReference.AbstractMethod(x, y, z...)

*Where Constructor References become useful is when they are used in tandem with Generics. By using a generic factory method one can create various types of objects.*

Let's have a peek.



```
1 //Step #1 - Create a functional interface
2 interface FuncInt <Ob, X, Y, Z>{
3     //Contains one and only abstract method
4     Ob func (X make, Y model, Z year);
5 }
6
7
8 //Step #2 - Create a Generic class providing a constructor compatible with FuncInt.func()'s definition
9 class Automobile <X,Y,Z> {
10
11     //Automobile Member Variables
12     private X make;
13     private Y model;
14     private Z year;
15
16     //Automobile constructor
17     Automobile (X make, Y model, Z year) {
18         this.make = make;
19         this.model = model;
20         this.year = year;
21     }
22
23     protected void what () {
24         System.out.println("This Automobile is a " + year + " " + make + " " + model + ".");
25     }
26 }
27
28 //Step #3 - Create a Non-Generic class providing a constructor compatible with FuncInt.func()'s definition
29 class Plane {
30
31     //Plane Member variables
32     private String make;
33     private String model;
34     private int year;
35
36     //Plane Constructor
37     Plane (String make, String model, Integer year) {
38         this.make = make;
39         this.model = model;
40         this.year = year; //Automatic unboxing
41     }
42
43     protected void what () {
44         System.out.println("This Plane is a " + make + " " + year + " " + model + ".");
45     }
46 }
```

The rest of the source.

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

ConstrRefGen.java
43     protected void what () {
44         System.out.println("This Plane is a " + make + " " + year + " " + model + ".");
45     }
46 }
47
48 //Step #2 - Class making use of method reference with generics
49 public class ConstrRefGen {
50
51     //here is where the magic happens
52     static <Ob, X, Y, Z> Ob factory (FuncInt<Ob, X, Y, Z> obj, X p1, Y p2, Z p3) {
53         return obj.func(p1,p2,p3);
54     }
55
56
57
58     //Main()
59     public static void main (String [] args) {
60         System.out.println();
61
62         //Example #1
63         FuncInt<Automobile,String, Integer> auto_cons = Automobile::new;
64         Automobile<String, String, Integer> honda = factory(auto_cons, "Honda", "Accord", 2000);
65         honda.what();
66
67         //Example #2
68         FuncInt<Plane,String, String, Integer> plane_cons = Plane::new;
69         Plane cessna = factory(plane_cons, "Cessna", "Skyhawk", 172);
70         cessna.what();
71
72         System.out.println();
73     }
74 }
```

Output

```
CODE: bash — Konsole
File Edit View Bookmarks Settings Help

adrian@localhost:~/Desktop/CODE> javac ConstrRefGen.java
adrian@localhost:~/Desktop/CODE> java ConstrRefGen

This Automobile is a 2000 Honda Accord.
This Plane is a Cessna 172 Skyhawk.

adrian@localhost:~/Desktop/CODE>
```

### Explanation:

Now there is a lot to digest here. In fact, the code may appear rather obscure at first glance, if you have never dived into Generics before. Let's break it down.

The first thing we do is create a generic functional interface. Pay attention to the details. We have four generic type parameters—Ob, X,Y,Z.

Ob—The Class whose constructor we want to reference  
X,Y,Z—The arguments to the constructor of said class.

If we substitute the generic method placeholders, the Abstract method could look like this: `SomeClass func (String make, String model, int year)`. Notice that because we made the interface generic, we can specify any return type or type of class we desire. This allows to unlock the true potential of constructor references.

The next two portions are relatively straightforward—We essentially create what is the same class, one generic, and one non-generic to demonstrate their interoperability with the factory method we will later define in the public class. Notice that the Constructors of these classes are compatible with the method signature of `FuncInt.func()`.

Enter into the public class of the file. This method is where the magic happens.

```
51 //Here is where the magic happens
52 static <Ob, X, Y, Z> Ob factory (FuncInt <Ob, X, Y, Z> obj, X p1, Y p2, Z p3) {
53     return obj.func(p1,p2,p3);
54 }
```

We label the method as static, so we can do without an instance of `ConstRefGen` and, after all, it's a factory method. Notice that the factory method has the same generic type parameters as the functional interface. Notice that the return type of the method is `Ob` which will be whichever class we decide. `X,Y,Z`, are, of course, the method arguments of a method in `Ob`. Notice that the function takes an instance of the `FuncInt` as an argument (with the Class Type and the method arguments as type parameters) as well as the arguments of the method of the class of type `Ob`.

Inside the body of the method it calls the method reference and feeds it the arguments passed in `factory()`.

Our first task—create a method reference that complies with `FuncInt<>`

Here we refer to the constructors of `Automobile` and `Plane` respectively.

Our next task—Create an object with a method reference.

To do this, we call `factory()` and we feed it the Constructor Reference it needs as well as the arguments for the constructor in question as defined by `factory ()`.



factory() can agnostically create constructor references to various methods because it is generic. Because the Plane & Automobile constructors match the method signature of FuncInt.func() they will work with as a method reference with FuncInt.func(). factory() returns an instance of the class in question by calling obj.func(x,y,z) which is a constructor method reference that when evaluated will give you an instance of the class that was specified as an argument to it.

Wrestle with this one for a while—It's a VERY useful addition to Java ;)

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

afinlay5/ConstructorReferences

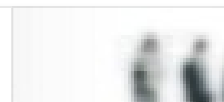
ConstructorReferences - Gradle source code repository for java source code examples...  
github.com



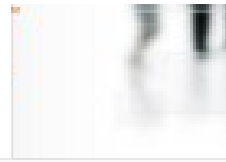
Looney Tunes Ending

**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](https://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.... ;)**



## **Works Cited**

[1]—Java, The Complete Reference 9th Ed ~ Schildt, Herbert

[2]—Oracle: “What’s New in JDK 8”

