Technical Article

Java 8: Lambdas, Part 1

by Ted Neward

Get to know lambda expressions in Java 8.

Few things excite a community of software developers more than a new release of their chosen programming language or platform. Java developers are no exception. In fact, we're probably even more excited about new releases, partly because there was a time not too long ago when we thought that Java's fortunes—like those of Java's creator, Sun—were on the wane. A brush with death tends to make one cherish renewed life all the more. But in this case, our enthusiasm also stems from the fact that unlike the prior release, Java 8 will finally get a new "modern" language feature that many of us have been requesting for years—if not decades.

Originally published in the July/Aug 2013 issue of *Java Magazine*. Subscribe today.

Of course, the major Java 8 buzz is around *lambdas* (also called *closures*), and that's where this two-part series will focus. But a language feature, on its own, will often appear anything but useful or interesting unless there's a certain amount of support behind it. Several features in Java 7 fit that description: enhanced numeric literals, for example, really couldn't get most people's eyes to light up.

In this case, however, not only do Java 8 function literals change a core part of the language, but they come alongside some additional language features designed to make them easier to use, as well as some library revamping that makes use of those features directly. These will make our lives as Java developers easier.

Java Magazine has run articles on lambdas before, but given that syntax and semantics might have changed since then and not all readers will have the time or inclination to read those articles, I will assume that readers have never seen any of this syntax before.

Note: This article is based on a prerelease version of Java SE 8 and, as such, it might not be entirely accurate when the final release ships. Syntax and semantics are always subject to change until the final release.

For those who desire a deeper, more official explanation of this material, Brian Goetz' papers, for example his "State of the Lambda: Libraries Edition" paper and others available at the Project Lambda home page, are priceless references.

Background: Functors

Java has always had a need for *functional objects* (sometimes called *functors*), though we in the community struggled mightily to downplay their usefulness and need. In Java's early days, when building GUIs, we needed blocks of code to respond to user events such as windows opening and closing, button presses, and scrollbar movement.

In Java 1.0, Abstract Window Toolkit (AWT) applications were expected, like their C++ predecessors, to extend window classes and override the event method of choice; this was deemed unwieldy and unworkable. So in Java 1.1, Sun gave us a set of "listener" interfaces, each with one or more methods corresponding to an event within the GUI.

But in order to make it easier to write the classes that must implement these interfaces and their corresponding methods, Sun gave us inner classes, including the ability to write such a class within the body of an existing class without having to specify a name—the ubiquitous *anonymous inner class*. (By the way, the listeners were hardly the only example of these that appeared during Java's history. As we'll see later, other, more "core" interfaces just like them appeared, for example, `Runnable` and `Comparator`.)

CODE = OBJECT As Java grew and matured, we found more places where treating blocks of code as objects (data, really) was not only useful but necessary.

Inner classes had some strangeness to them, both in terms of syntax and semantics. For example, an inner class was either a *static inner class* or an *instance inner class*, depending not on any particular keyword (though static inner classes could be explicitly stated as such using the `static` keyword) but on the lexical context in which the instance was created. What that meant, in practical terms, is that Java developers often got questions such as those in Listing 1 wrong on programming interviews.

Listing 1

Copy

```
class InstanceOuter {
  public InstanceOuter(int xx) { x = xx; }

  private int x;

  class InstanceInner {
    public void printSomething() {
      System.out.println("The value of x in my outer is " + x);
    }
  }
}

class StaticOuter {
  private static int x = 24;

  static class StaticInner {
    public void printSomething() {
      System.out.println("The value of x in my outer is " + x);
    }
  }
}

public class InnerClassExamples {
  public static void main(String... args) {
    InstanceOuter io = new InstanceOuter(12);

    // Is this a compile error?
    InstanceOuter.InstanceInner ii = io.new InstanceInner();

    // What does this print?
    ii.printSomething(); // prints 12

    // What about this?
    StaticOuter.StaticInner si = new StaticOuter.StaticInner();
    si.printSomething(); // prints 24
  }
}
```

“Features” such as inner classes often convinced Java developers that such functionality was best relegated to the corner cases of the language, suitable for a programming interview and not much else—except when they needed them. Even then, most of the time, they were used purely for event-handling reasons.

Above and Beyond

As clunky as the syntax and semantics were, however, the system worked. As Java grew and matured, we found more places where treating blocks of code as objects (data, really) was not only useful but necessary. The revamped security system in Java SE 1.2 found it useful to pass in a block of code to execute under a different security context. The revamped `Collection` classes that came with that same release found it useful to pass in a block of code in order to know how to impose a sort order on a sorted collection. Swing found it useful to pass in a block of code in order to decide which files to display to the user in a File Open or File Save dialog box. And so on. It worked—though often through syntax that only a mother could love.

But when concepts of functional programming began to enter mainstream programming, even Mom gave up. Though possible (see this remarkably complete example), functional programming in Java was, by any account, obtuse and awkward. Java needed to grow up and join the host of mainstream programming languages that offer first-class language support for defining, passing, and storing blocks of code for later execution.

Java 8: Lambdas, Target Typing, and Lexical Scoping

Java 8 introduces several new language features designed to make it easier to write such blocks of code—the key feature being *lambda expressions*, also colloquially referred to as *closures* (for reasons we'll discuss later) or *anonymous methods*. Let's take these one at a time.

Lambda expressions. Funda-men-tally, a lambda expression is just a shorter way of writing an implementation of a method for later execution. Thus, while we used to define a `Runnable` as shown in Listing 2, which uses the anonymous inner class syntax and clearly suffers from a "vertical problem" (meaning that the code takes too many lines to express the basic concept), the Java 8 lambda syntax allows us to write the code as shown in Listing 3.

Listing 2

Copy

```
public class Lambdas {
  public static void main(String... args) {
    Runnable r = new Runnable() {
      public void run() {
        System.out.println("Howdy, world!");
      }
    };
    r.run();
  }
}
```

Listing 3

Copy

```
public static void main(String... args) {
    Runnable r2 = () -> System.out.println("Howdy, world!");
    r2.run();
  }
```

Both approaches have the same effect: a `Runnable`-implementing object whose `run()` method is being invoked to print something to the console. Under the hood, however, the Java 8 version is doing a little more than just generating an anonymous class that implements the `Runnable` interface—some of which has to do with the `invoke dynamic` bytecode that was introduced in Java 7. We won't get to that level of detail here, but know that this is more than "just" an anonymous class instance.

Functional interfaces. The `Runnable` interface—like the `Callable<T>` interface, the `Comparator<T>` interface, and a whole host of other interfaces already defined within Java—is what Java 8 calls a `functional interface`: it is an interface that requires exactly one method to be implemented in order to satisfy the requirements of the interface. This is how the syntax achieves its brevity, because there is no ambiguity around which method of the interface the lambda is trying to define.

The designers of Java 8 have chosen to give us an annotation, `@FunctionalInterface`, to serve as a documentation hint that an interface is designed to be used with lambdas, but the compiler does not require this—it determines "functional interfaceness" from the structure of the interface, not from the annotation.

Throughout the rest of this article, we'll continue to use the `Runnable` and `Comparator<T>` interfaces as working examples, but there is nothing particularly special about them, except that they adhere to this functional interface single-method restriction. Any developer can, at any time, define a new functional interface—such as the following one—that will be the interface target type for a lambda.

Copy

```
interface Something {
  public String doit(Integer i);
}
```

The `Something` interface is every bit as legal and legitimate a functional interface as `Runnable` or `Comparator<T>`; we'll look at it again after getting some lambda syntax under our belt.

Syntax. A lambda in Java essentially consists of three parts: a parenthesized set of parameters, an arrow, and then a body, which can either be a single expression or a block of Java code. In the case of the example shown in Listing 2, `run` takes no parameters and returns `void`, so there are no parameters and no return value. A `Comparator<T>`-based example, however, highlights this syntax a little more obviously, as shown in Listing 4. Remember that `Comparator` takes two strings and returns an integer whose value is negative (for "less than"), positive (for "greater than"), and zero (for "equal").

Listing 4

Copy

```
public static void main(String... args) {
    Comparator<String> c =
      (String lhs, String rhs) -> lhs.compareTo(rhs);
    int result = c.compare("Hello", "World");
  }
```

If the body of the lambda requires more than one expression, the value returned from the expression can be handed back via the `return` keyword, just as with any block of Java code (see Listing 5).

Listing 5

Copy

```
public static void main(String... args) {
      Comparator<String> c =
        (String lhs, String rhs) ->
          {
            System.out.println("I am comparing" +
                               lhs + " to " + rhs);
            return lhs.compareTo(rhs);
          };
      int result = c.compare("Hello", "World");
    }
```

(Where we put the curly braces in code such as Listing 5 will likely dominate Java message boards and blogs for years to come.) There are a few restrictions on what can be done in the body of the lambda, most of which are pretty intuitive—a lambda body can't "break" or "continue" out of the lambda, and if the lambda returns a value, every code path must return a value or throw an exception, and so on. These are much the same rules as for a standard Java method, so they shouldn't be too surprising.

Type inference. One of the features that some other languages have been touting is the idea of `type inference`: that the compiler should be smart enough to figure out what the type parameters should be, rather than forcing the developer to retype the parameters.

Such is the case with the `Comparator` example in Listing 5. If the target type is a `Comparator<String>`, the objects passed in to the lambda *must* be strings (or some subtype); otherwise, the code wouldn't compile in the first place. (This isn't new, by the way—this is "Inheritance 101.")

In this case, then, the `String` declarations in front of the `lhs` and `rhs` parameters are entirely redundant and, thanks to Java 8's enhanced type inference features, they are entirely optional (see Listing 6).

Listing 6

Copy

```
public static void main(String... args) {
      Comparator<String> c =
        (lhs, rhs) ->
          {
            System.out.println("I am comparing" +
                               lhs + " to " + rhs);
            return lhs.compareTo(rhs);
        };
      int result = c.compare("Hello", "World");
    }
```

The language specification will have precise rules as to when explicit lambda formal type declarations are needed, but for the most part, it's proving to be the default, rather than the exception, that the parameter type declarations for a lambda expression can be left out completely.

One interesting side effect of Java's lambda syntax is that for the first time in Java's history, we find something that cannot be assigned to a reference of type `Object` (see Listing 7)—at least not without some help.

Listing 7

Copy

```
public static void main4(String... args) {
    Object o = () -> System.out.println("Howdy, world!");
      // will not compile
  }
```

The compiler will complain that `Object` is not a functional interface, though the real problem is that the compiler can't quite figure out which functional interface this lambda should implement: `Runnable` or something else? We can help the compiler with, as always, a cast, as shown in Listing 8.

Listing 8

Copy

```
public static void main4(String... args) {
    Object o = (Runnable) () -> System.out.println("Howdy, world!");
      // now we're all good
  }
```

Recall from earlier that lambda syntax works with any interface, so a lambda that is inferred to a custom interface will also be inferred just as easily, as shown in Listing 9. Primitive types are equally as viable as their wrapper types in a lambda type signature, by the way.

Listing 9

Copy

```
Something s = (Integer i) -> { return i.toString(); };
    System.out.println(s.doit(4));
```

Again, none of this is really new; Java 8 is just applying Java's long-standing principles, patterns, and syntax to a new feature. Spending a few minutes exploring type inference in code will make that clearer, if it's not clear already.

Lexical scoping. One thing that is new, however, is how the compiler treats names (identifiers) within the body of a lambda compared to how it treated them in an inner class. Consider the inner class example shown in Listing 10 for a moment.

Listing 10

Copy

```
class Hello {
  public Runnable r = new Runnable() {
      public void run() {
        System.out.println(this);
        System.out.println(toString());
      }
    };

  public String toString() {
    return "Hello's custom toString()";
  }
}

public class InnerClassExamples {
  public static void main(String... args) {
    Hello h = new Hello();
    h.r.run();
  }
}
```

When run, the code in Listing 10 counterintuitively produces "Hello$1@f7ce53" on my machine. The reason for this is simple to understand: both the keyword `this` and the call to `toString` in the implementation of the anonymous `Runnable` are bound to the anonymous inner class implementation, because that is the innermost scope that satisfies the expression.

If we wanted (as the example seems to imply) to print out `Hello`'s version of `toString`, we have to explicitly qualify it using the "outer `this`" syntax from the inner classes portion of the Java spec, as shown in Listing 11. How's that for intuitive?

Listing 11

Copy

```
class Hello {
  public Runnable r = new Runnable() {
      public void run() {
        System.out.println(Hello.this);
        System.out.println(Hello.this.toString());
      }
    };

  public String toString() {
    return "Hello's custom toString()";
  }
}
```

Frankly, this is one area where inner classes simply created more confusion than they solved. Granted, as soon as the reason for the `this` keyword showing up in this rather unintuitive syntax was explained, it sort of made sense, but it made sense in the same way that politicians' perks make sense.

Lambdas, however, are *lexically scoped*, meaning that a lambda recognizes the immediate environment around its definition as the next outermost scope. So the lambda example in Listing 12 produces the same results as the second `Hello` nested class example in Listing 11, but with a much more intuitive syntax.

Listing 12

Copy

```
class Hello {
  public Runnable r = () -> {
      System.out.println(this);
      System.out.println(toString());
    };

  public String toString() {
    return "Hello's custom toString()";
  }
}
```

This means, by the way, that `this` no longer refers to the lambda itself, which might be important in certain cases—but those cases are few and far between. What's more, if such a case does arise (for example, perhaps the lambda needs to return a lambda and it wants to return itself), there's a relatively easy workaround, which we'll get to in a second.

Variable capture. Part of the reason that lambdas are called *closures* is that a function literal (such as what we've been writing) can "close over" variable references that are outside the body of the function literal in the enclosing scope (which, in the case of Java, would typically be the method in which the lambda is being defined). Inner classes could do this, too, but of all the subjects that frustrated Java developers the most about inner classes, the fact that inner classes could reference only "final" variables from the enclosing scope was near the top.

Lambdas relax this restriction, but only by a little: as long as the variable reference is "effectively final," meaning that it's final in all but name, a lambda can reference it (see Listing 13). Because `message` is never modified within the scope of the `main` method enclosing the lambda being defined, it is effectively final and, therefore, eligible to be referenced from within the `Runnable` lambda stored in r.

Listing 13

Copy

```
public static void main(String... args) {
    String message = "Howdy, world!";
    Runnable r = () -> System.out.println(message);
    r.run();
  }
```

While on the surface this might sound like it's not much of anything, remember that the lambda semantics rules don't change the nature of Java as a whole—objects on the other side of a reference are still accessible and modifiable long after the lambda's definition, as shown in Listing 14.

Listing 14

Copy

```
public static void main(String... args) {
    StringBuilder message = new StringBuilder();
    Runnable r = () -> System.out.println(message);
    message.append("Howdy, ");
    message.append("world!");
    r.run();
  }
```

Astute developers familiar with the syntax and semantics of older inner classes will remember that this was also true of references declared "final" that were referenced within an inner class—the `final` modifier applied only to the reference, not to the object on the other side of the reference. Whether this is a bug or a feature in the eyes of the Java community remains to be seen, but it is what it is, and developers would be wise to understand how lambda variable capture works, lest a surprise bug appear. (In truth, this behavior isn't new—it's just recasting existing functionality of Java in fewer keystrokes and with more support from the compiler.)

Method references. Thus far, all the lambdas we've examined have been anonymous literals—essentially, defining the lambda right at the point of use. This is great for one-off kinds of behavior, but it doesn't really help much when that behavior is needed or wanted in several places. Consider, for example, the following `Person` class. (Ignore the lack of proper encapsulation for the moment.)

Copy

```
class Person {
  public String firstName;
  public String lastName;
  public int age;
});
```

When a Person is put into a SortedSet or needs to be sorted in a list of some form, we want to have different mechanisms by which Person instances can be sorted—for example, sometimes by first name and sometimes by last name. This is what Comparator<T> is for: to allow us to define an imposed ordering by passing in the Comparator<T> instance.

Lambdas certainly make it easier to write the sort code, as shown in Listing 15. But sorting Person instances by first name is something that might need to be done many times in the codebase, and writing that sort of algorithm multiple times is clearly a violation of the Don't Repeat Yourself (DRY) principle.

SCOPE IT OUT Lambdas are lexically scoped, meaning that a lambda recognizes the immediate environment around its definition as the next outermost scope.

Listing 15

Copy

```
public static void main(String... args) {
    Person[] people = new Person[] {
      new Person("Ted", "Neward", 41),
      new Person("Charlotte", "Neward", 41),
      new Person("Michael", "Neward", 19),
      new Person("Matthew", "Neward", 13)
    };
    // Sort by first name
    Arrays.sort(people,
      (lhs, rhs) -> lhs.firstName.compareTo(rhs.firstName));
    for (Person p : people)
      System.out.println(p);
  }
```

The Comparator can certainly be captured as a member of Person itself, as shown in Listing 16. The Comparator<T> could then just be referenced as any other static field could be referenced, as shown in Listing 17. And, truthfully, functional programming zealots will prefer this style, because it allows for the functionality to be combined in various ways.

Listing 16

Copy

But it feels strange to the traditional Java developer, as opposed to simply creating a method that fits the signature of Comparator<T> and then using that directly—which is exactly what a method reference allows (see Listing 18). Notice the double-colon method-naming style, which tells the compiler that the method `compareFirstNames`, defined on `Person`, should be used here, rather than a method literal.

Listing 18

Copy

```
class Person {
  public String firstName;
  public String lastName;
  public int age;

  public static int compareFirstNames(Person lhs, Person rhs) {
    return lhs.firstName.compareTo(rhs.firstName);
  }

  // ...
}

  public static void main(String... args) {
    Person[] people = . . .;
    // Sort by first name
    Arrays.sort(people, Person::compareFirstNames);
    for (Person p : people)
      System.out.println(p);
  }
```

Another way to do this, for those who are curious, would be to use the `compareFirstNamest` method to create a `Comparator<Person>` instance, like this:

```
Comparator cf = Person::compareFirstNames;
```

And, just to be even more succinct, we could avoid some of the syntactic overhead entirely by making use of some of the new library features to write the following, which makes use of a higher-order function (meaning, roughly, a function that passes around functions) to essentially avoid all of the previous code in favor of a one-line in-place usage:

Copy

```
Arrays.sort(people, comparing(
  Person::getFirstName));
```

This, in part, is why lambdas, and the functional programming techniques that come with them, are so powerful.

Virtual extension methods. One of the drawbacks frequently cited about interfaces, however, is that they have no default implementation, even when that implementation is ridiculously obvious. Consider, for example, a fictitious `Relational` interface, which defines a series of methods to mimic relational methods (greater than, less than, greater than or equal to, and so on). As soon as any one of those methods is defined, it's easy to see how the others could all be defined in terms of the first one. In fact, all of them could be defined in terms of a `Comparable<T>`'s `compare` method, if the definition of the `compare` method were known ahead of time. But interfaces cannot have default behavior, and an abstract class is still a class and occupies any potential subclass' one implementation-inheritance slot.

With Java 8, however, as these function literals become more widespread, it becomes more important to be able to specify default behavior without losing the "interfaceness" of the interface. Thus, Java 8 now introduces *virtual extension methods* (which used to be known in a previous draft as *defender methods*), essentially allowing an interface to specify a default behavior for a method, if none is provided in a derived implementation.

Consider, for a moment, the `Iterator` interface. Currently, it has three methods (`hasNext`, `next`, and `remove`), and each must be defined. But an ability to "skip" the next object in the iteration stream might be helpful. And because the `Iterator`'s implementation is easily defined in terms of the other three, we can provide it, as shown in Listing 19.

Listing 19

Copy

```
interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();

  void skip(int i) default {
    for (; i > 0 && hasNext(); i--) next();
  }
}
```

Some within the Java community will scream, claiming that this is just a mechanism to weaken the declarative power of interfaces and create a scheme that allows for multiple inheritance in Java. To a degree, this is the case, particularly because the rules around precedence of default implementations (in the event that a class implements more than one interface with different default implementations of the same method) will require significant study.

But as the name implies, virtual extension methods provide a powerful mechanism for extending existing interfaces, without relegating the extensions to some kind of second-class status. Using this mechanism, Oracle can provide additional, powerful behavior for existing libraries without requiring developers to track different kinds of classes. There's no `SkippingIterator` class that developers now have to downcast to for those collections that support it. In fact, no code anywhere has to change, and all `Iterator<T>`s, no matter when they were written, will automatically have this skipping behavior.

It is through virtual extension methods that the vast majority of the changes that are happening in the `Collection` classes are coming. The good news is that your `Collection` classes are getting new behavior, and the even better news is that your code won't have to change an iota in the meantime. The bad news is that we have to defer that discussion to the next article in this series.

Learn More

Brian Goetz' "State of the Lambda: Libraries Edition" paper

Project Lambda home page

"Maurice Naftalin's Lambda FAQ"

Conclusion

Lambdas will bring a lot of change to Java, both in terms of how Java code will be written and how it will be designed. Some of these changes, inspired by functional programming languages, will change the way Java programmers think about writing code—which is both an opportunity and a hassle.

We'll talk more about the impact these changes will have on the Java libraries in the next article in this series, and we'll spend a little bit of time talking about how these new APIs, interfaces, and classes open up some new design approaches that previously wouldn't have been practical due to the awkwardness of the inner classes syntax.

Java 8 is going to be a very interesting release. Strap in, it's going to be a rocket-ship ride.

Ted Neward (@tedneward) is an architectural consultant for Neudesic. He has served on several Expert Groups; authored many books, including *Effective Enterprise Java* (Addison-Wesley Professional, 2004) and *Professional F# 2.0* (Wrox, 2010); written hundreds of articles on Java, Scala, and other technologies; and spoken at hundreds of conferences.

Resources for
Careers
Developers
Investors
Partners
Researchers

Why Oracle
Analyst Reports
Best cloud-based ERP
Cloud Economics
Corporate Responsibility

Learn
What is cloud computing?
What is CRM?
What is Docker?
What is Kubernetes?

News and Events
News
Oracle CloudWorld
Oracle CloudWorld Tour

Contact Us
US Sales: +1.800.633.0738
How can we help?
Subscribe to emails
Integrity Helpline

© 2023 Oracle Privacy / Do Not Sell My Info Cookie Preferences Ad Choices Careers

Country/Region