Toby Weston

Learning Java Lambdas

An in-depth look at one of the most important features of modern Java

Packt

Learning Java Lambdas

Table of Contents

Learning Java Lambdas

Learning Java Lambdas

First published: March 2017

Production reference: 1290317

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78728-208-7

www.packtpub.com

Credits

Author Toby Weston	
Acquisition Editor Ben Renow-Clarke	**Indexer** Mariammal Chettiyar
Technical Editor Nidhisha Shetty	**Production Coordinator** Arvindkumar Gupta

About the Author

Toby Weston specializes in modern software development; particularly, functional and object-oriented programming, agile and lean best practice. He wrote the book Essential Acceptance Testing, and he has written for magazines as well as does regularly blogging. He has been part of the software industry for more than fifteen years and loves what he does. He loves talking and writing about it and sharing his experiences online.

Customer Feedback

Preface

What this book covers

This book takes an in-depth look at lambdas and their supporting features; things like functional interfaces and type inference.

After reading the book, you'll:

- Have an overview of new features in modern Java
- Understand lambdas in-depth, their background, syntax, implementation details and how and when to use them
- Understand the difference between functions to classes and why that's relevant to lambdas
- Understand the difference between lambdas and closures
- Appreciate the improvements to type inference that drive a lot of the new features
- Be able to use method references and understand scoping and "effectively final"
- Understand the differences in bytecode produced when using lambdas
- Be able to reason about exceptions and exception handling best practice when using lambdas

What you need for this book

The latest version of JDK and a text editor or IDE.

Who this book is for

Whether you're migrating legacy Java programs to a more modern Java style or building applications from scratch, this book will help you start to leverage the power of functional programming on the Java platform.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This LISP expression evaluates to a function, that when applied will take a single argument, bind it to `arg` and then add `1` to it."

A block of code is set as follows:

```
void anonymousClass() {
    final Server server = new HttpServer();
    waitFor(new Condition() {
        @Override
        public Boolean isSatisfied() {
            return !server.isRunning();
        }
    });
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
void anonymousClass() {
    final Server server = new HttpServer();
    waitFor(new Condition() {
        @Override
        public Boolean isSatisfied() {
            return !server.isRunning();
        }
    });
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/LearningJavaLambdas_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Chapter 1. Introduction

Welcome to Learning Java Lambdas.

Java underwent huge changes in it's version 8 release. A lot was driven by the perception that Java was becoming long in the tooth. To compete with modern, functional programming languages, Java 8 introduced functional programming constructs like lambdas to better support a more functional style.

This book offers a concise explanation of lambdas and various other features required to make them work in Java. As well as offering background, syntax and usage examples of lambdas, the book describes other related features, such as functional interfaces and type inference.

The road to modern Java

Java 8 was released on March 18, 2014, two years seven months after the previous release. It was plagued with delays and technical problems but when it finally came, it represented one of the biggest shifts in Java since Java 5.

The headliners were of course lambdas and a retrofit to support functional programming ideas. With languages such as **Scala** taking center stage and the modern trend towards functional programming, Java had to do something to keep up.

Although Java is not and never will be a *pure* functional programming language, the changes in Java 8 enabled developers to use functional idioms more easily than in previous versions. With discipline and experience, you can now get a lot of the benefits of functional programming without resorting to third-party libraries.

Modern Java features

To give you an idea of just how big a change Java 8 was, and why it ushered in a new, modern Java, here's a mostly complete list of the new features it introduced:

- Lambda support.
- The core APIs were updated to take advantage of lambdas, including the collection APIs and a new functional package to help build functional constructs.
- Entirely new APIs were developed that use lambdas, things like the stream API which brought functional style processing of data. For example, functions like `map` and `flatMap` from the stream API enable a declarative way to process lists and move away from external iteration to internal iteration. This in turn allows the *library vendors* to worry about the details and optimize processing however they like. For example, Java now comes with a parallel way to process streams without bothering the developer with the details.
- Minor changes to the core APIs; new helper methods were introduced for strings, collections, comparators, numbers and maths.
- Some of the additions are changing the way that people code. For example, the `Optional` class will be familiar to some, and it enables a better way to deal with nulls.
- There were various concurrency library improvements. Things like an improved concurrent hash map, completable futures, thread safe accumulators, an improved read write lock (called a StampedLock), an implementation of a work stealing thread pool and much more besides.
- Support for adding static methods to interfaces.
- Default methods (otherwise known as *virtual extension* or *defender methods*).
- Type inference was improved and new constructs like functional interfaces and method references were introduced to better support lambdas.
- An improved date and time API was introduced (similar to the popular `Joda-time` library).
- The `IO` and `NIO` packages received welcome additions to enable working with IO streams using the new streams API.
- Reflection and annotations were improved.
- An entirely new JavaScript engine shipped with Java 8. Nashorn replaced Rhino, and was faster and had better support for ECMA-Script.
- JVM improvements; the integration with JRocket was completed, creating a faster JVM.
- The JVM dropped the idea of perm gen, instead using native OS memory for class metadata. This is a huge deal and provides better memory utilization.
- The JRocket integration also brought Mission control (jmc) to the JDK as standard. It compliments JConsole and VisualVM with similar functionality but adds very inexpensive profiling.
- Other miscellaneous improvements, like improvements to JavaFX, base64 encoding support and more.

Chapter 2. Lambdas Introduction

In this chapter, we'll introduce the ideas of lambdas, we'll:

- Discuss some background to lambdas and functional programming in general
- Talk about functions versus classes in Java
- Look at the basic syntax for lambdas in Java

λs in functional programming

Before we look at things in more depth, let's look at some general background to lambdas.

If you haven't seen it before, the Greek letter λ (**lambda**) is often used as shorthand when talking about lambdas.

1930s and the lambda calculus

In computer science, lambdas go back to the lambda-calculus. A mathematical notation for functions introduced by **Alonzo Church** in the 1930s. It was a way to explore mathematics using functions and was later re-discovered as a useful tool in computer science.

It formalized the notion of *lambda terms* and the rules to transform those terms. These rules or *functions* map directly into modern computer science ideas. All functions in the lambda-calculus are anonymous which again has been taken literally in computer science.

Here's an example of a lambda-calculus expression:

A lambda-calculus expression

```
λx.x+1
```

This defines an anonymous function or *lambda* with a single argument `x`. The body follows the dot and adds one to that argument.

1950s and LISP

In the 1950s, *John McCarthy* invented LISP whilst at MIT. This was a programming language designed to model mathematical problems and was heavily influenced by the lambda-calculus.

It used the word lambda as an operator to define an anonymous function.

Here's an example:

A LISP expression

```
(lambda (arg) (+ arg 1))
```

This LISP expression evaluates to a function, that when applied will take a single argument, bind it to `arg` and then add `1` to it.

The two expressions produce the same thing, a function to increment a number. You can see the two are very similar.

The lambda-calculus and LISP have had a huge influence on functional programming. The ideas of applying functions and reasoning about problems using functions has moved directly into programming languages. Hence the use of the term in our field. A lambda in the calculus is the same thing as in modern programming languages and is used in the same way.

What is a lambda

In simple terms then, a lambda is just an anonymous function. That's it. Nothing special. It's just a compact way to define a function. Anonymous functions are useful when you want to pass around fragments of reusable functionality. For example, passing functions into other functions.

Many main stream languages already support lambdas including Scala, C#, Objective-C, Ruby, C++ (11), Python and many others.

Functions vs classes

Bear in mind that an anonymous *function* isn't the same as an anonymous *class* in Java. An anonymous class in Java still needs to be instantiated to an object. It may not have a proper name, but it can be useful only when it's an object.

A *function* on the other hand has no instance associated with it. Functions are disassociated with the data they act on whereas an object is intimately associated with the data it acts upon.

You can use lambdas in modern Java anywhere you would have previously used a single method interface so it may just look like syntactic sugar but it's not. Let's have a look at how they differ and compare anonymous classes to lambdas; classes vs. functions.

Lambdas in modern Java

A typical implementation of an anonymous class (a single method interface) in Java pre-8, might look something like this. The `anonymousClass` method is calling the `waitFor` method, passing in some implementation of `Condition`; in this case, it's saying, wait for some server to shutdown:

Typical usage of an anonymous class

```
void anonymousClass() {
    final Server server = new HttpServer();
    waitFor(new Condition() {
        @Override
        public Boolean isSatisfied() {
            return !server.isRunning();
        }
    });
}
```

The functionally equivalent lambda would look like this:

Equivalent functionality as a lambda

```
 void closure() {
     Server server = new HttpServer();
     waitFor(() -> !server.isRunning());
 }
```

Where in the interest of completeness, a naive polling `waitFor` method might look like this:

```
class WaitFor {
    static void waitFor(Condition condition) throws
    InterruptedException {
        while (!condition.isSatisfied())
            Thread.sleep(250);
    }
}
```

Some theoretical differences

Firstly, both implementations are in-fact closures, the latter is also a lambda. We'll look at this distinction in more detail later in the *Lambdas vs closures* section. It means that both have to capture their "environment" at runtime. In Java pre-8, this means copying the things the closure needs into an instance of an class (an anonymous instances of Condition). In our example, the server variable would need to be copied into the instance.

As it's a copy, it has to be declared final to ensure that it can not be changed between when it's captured and when it's used. These two points in time could be very different given that closures are often used to defer execution until some later point (see lazy evaluation for example). Modern Java uses a neat trick whereby if it can reason that a variable is never updated, it might as well be final so it treats it as *effectively final* and you don't need to declare it as final explicitly.

A lambda on the other hand, doesn't need to copy it's environment or capture any terms. This means it can be treated as a genuine function and not an instance of a class. What's the difference? Plenty.

Functions vs classes

For a start, functions; genuine functions, don't need to be instantiated many times. I'm not sure if instantiation is even the right word to use when talking about allocating memory and loading a chunk of machine code as a function. The point is, once it's available, it can be re-used, it's idempotent in nature as it retains no state. Static class methods are the closest thing Java has to functions.

For Java, this means that a lambda need not be instantiated every time it's evaluated which is a big deal. Unlike instantiating an anonymous class, the memory impact should be minimal.

In terms of some conceptual differences then:

- Classes must be instantiated, whereas functions are not.
- When classes are newed up, memory is allocated for the object.
- Memory need only be allocated once for functions. They are stored in the *permanent* area of the heap.
- Objects act on their own data, functions act on unrelated data.
- Static class methods in Java are roughly equivalent to functions.

Some concrete differences

Some concrete differences between functions and classes include their capture semantics and how they shadow variables.

Capture semantics

Another difference is around capture semantics for this. In an anonymous class, this refers to the instance of the anonymous class. For example, `Foo$InnerClass` and not `Foo`. That's why you have slightly odd looking syntax like `Foo.this.x` when you refer to the enclosing scope from the anonymous class.

In lambdas on the other hand, this refers to the enclosing scope (Foo directly in our example). In fact, lambdas are **entirely lexically scoped**, meaning they don't inherit any names from a super type or introduce a new level of scoping at all; you can directly access fields, methods and local variables from the enclosing scope.

For example, this class shows that the lambda can reference the `firstName` variable directly.

```
public class Example {
    private String firstName = "Jack";

    public void example() {
        Function<String, String> addSurname = surname -> {
            // equivalent to this.firstName
            return firstName + " " + surname;  // or even,
            this.firstName
        };
    }
}
```

Here, `firstName` is shorthand for `this.firstName` and because this refers to the enclosing scope (the class `Example`), it's value will be "Jack".

The anonymous class equivalent would need to explicitly refer to `firstName` from the enclosing scope. You can't use this as in this context, this means the anonymous instance and there is no `firstName` there. So, the following will compile:

```
public class Example {
    private String firstName = "Charlie";

    public void anotherExample() {
        Function<String, String> addSurname = new Function<String,
        String>() {
            @Override
            public String apply(String surname) {
                return Example.this.firstName + " " + surname;
                // OK
            }
        };
    }
}
```

but this will not.

```
public class Example {
    private String firstName = "Charlie";

  public void anotherExample() {
    Function<String, String> addSurname = new Function<String,
    String>() {
      @Override
      public String apply(String surname) {
        return this.firstName + " " + surname;   // compiler error
      }
    };
  }
}
```

You could still access the field directly (that is, simply calling return `firstName + " " + surname`) but you can't do so using this. The point here is to demonstrate the difference in capture schematics for this when used in lambdas vs. anonymous instances.

Shadowed variables

Referencing shadowed variables becomes much more straight forward to reason about with the simplified `this` semantics. For example,

```
public class ShadowingExample {

    private String firstName = "Charlie";

    public void shadowingExample(String firstName) {
        Function<String, String> addSurname = surname -> {
            return this.firstName + " " + surname;
        };
    }
}
```

Here, because `this` is inside the lambda, it refers to the enclosing scope. So `this.firstName` will have the value `"Charlie"` and not the method parameter of the same name. The capture semantics make it clearer. If you use `firstName` (and drop the `this`), it will refer to the parameter.

In the next example, using an anonymous instance, `firstName` simply refers to the parameter. If you want to refer to the enclosing version, you'd use `Example.this.firstName`:

```
public class ShadowingExample {

    private String firstName = "Charlie";

    public void anotherShadowingExample(String firstName) {
        Function<String, String> addSurname = new Function<String,
        String>() {
            @Override
            public String apply(String surname) {
                return firstName + " " + surname;
            }
        };
    }
}
```

Summary

Functions in the academic sense are very different things from anonymous classes (which we often treat like functions in Java pre-8). It's useful to understand the distinctions to be able to justify the use of lambdas for something other than just their concise syntax. Of course, there's lots of additional advantages in using lambdas (not least the retrofit of the JDK to heavily use them).

When we take a look at the new lambda syntax next, remember that although lambdas are used in a very similar way to anonymous classes in Java, they are technically different. Lambdas in Java need not be instantiated every time they're evaluated unlike an instance of an anonymous class.

This should serve to remind you that lambdas in Java are not just syntactic sugar.

λ basic syntax

Let's take a look at the basic lambda syntax.

A lambda is basically an anonymous block of functionality. It's a lot like using an anonymous class instance. For example, if we want to sort an array in Java, we can use the `Arrays.sort` method which takes an instance of the `Comparator` interface.

It would look something like this:

```
Arrays.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer first, Integer second) {
        return first.compareTo(second);
    }
});
```

The `Comparator` instance here is a an abstract piece of the functionality; it means nothing on its own; it's only when it's used by the `sort` method that it has purpose.

Using Java's new syntax, you can replace this with a lambda which looks like this:

```
Arrays.sort(numbers, (first, second) -> first.compareTo(second));
```

It's a more succinct way of achieving the same thing. In fact, Java treats this as if it were an instance of the `Comparator` class. If we were to extract a variable for the lambda (the second parameter), it's type would be `Comparator<Integer>` just like the anonymous instance above.

```
Comparator<Integer> ascending = (first, second) -> first.compareTo(second);
Arrays.sort(numbers, ascending);
```

Because `Comparator` has only a single abstract method on it; `compareTo`, the compiler can piece together that when we have an anonymous block like this, we really mean an instance of `Comparator`. It can do this thanks to a couple of the other new features that we'll talk about later; functional interfaces and improvements to type inference.

Syntax breakdown

You can always convert from using a single abstract method to a using lambda.

Let's say we have an an interface `Example` with a method `apply`, returning some type and taking some argument:

```
interface Example {
     R apply(A arg);
}
```

We could instantiate an instance with something like this:

```
new Example() {
    @Override
    public R apply(A args) {
        body
    }
};
```

And to convert to a lambda, we basically trim the fat. We drop the instantiation and annotation, drop the method details which leaves just the argument list and the body.

```
(args) {
    body
}
```

we then introduce the new arrow symbol to indicate both that the whole thing is a lambda and that what follows is the body and that's our basic lambda syntax:

```
(args) -> {
    body
}
```

Let's take the sorting example from earlier through these steps. We start with the anonymous instance:

```
Arrays.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer first, Integer second) {
        return first.compareTo(second);
    }
});
```

and trim the instantiation and method signature:

```
Arrays.sort(numbers, (Integer first, Integer second) {
    return first.compareTo(second);
});
```

introduce the lambda

```
Arrays.sort(numbers, (Integer first, Integer second) -> {
    return first.compareTo(second);
});
```

and we're done. There's a couple of optimizations we can do though. You can drop the types if the compiler knows enough to infer them.

```
Arrays.sort(numbers, (first, second) -> {
    return first.compareTo(second);
});
```

and for simple expressions, you can drop the braces to produce a lambda expression:

```
Arrays.sort(numbers, (first, second) -> first.compareTo(second));
```

In this case, the compiler can infer enough to know what you mean. The single statement returns a value consistent with the interface, so it says, "no need to tell me that you're going to return something, I can see that for myself".

For single argument interface methods, you can even drop the first brackets. For example the lambda taking an argument `x` and returning `x + 1`;

```
(x) -> x + 1
```

can be written without the brackets

```
x -> x + 1
```

Summary

Let's recap with a summary of the syntax options.

Syntax Summary:

```
(int x, int y) -> { return x + y; }
(x, y) -> { return x + y; }
(x, y) -> x + y; x -> x * 2
() -> System.out.println("Hey there!");
System.out::println;
```

The first example `((int x, int y) -> { return x + y; })` is the most verbose way to create a lambda. The arguments to the function along with their types are in parenthesis, followed by the new arrow syntax and then the body; the code block to be executed.

You can often drop the types from the argument list, like `(x, y) -> { return x + y; }`. The compiler will use type inference here to try and guess the types. It does this based on the context that you're trying to use the lambda in.

If your code block returns something or is a single line expression, you can drop the braces and return statement, for example `(x, y) -> x + y;`.

In the case of only a single argument, you can drop the parentheses `x -> x * 2`.

If you have no arguments at all, the "hamburger" symbol is needed,

```
() -> System.out.println("Hey there!");.
```

In the interest of completeness, there is another variation; a kind of shortcut to a lambda called a *method reference*. An example is something like `System.out::println;`, which is basically a short cut to the lambda `(value -> System.out.prinltn(value)`.

We're going to talk about method references in more detail later, so for now, just be aware that they exist and can be used anywhere you can use a lambda.

Chapter 3. Lambdas in Depth

In this section, we'll take a look at things in a little more detail and talk about some related topics, things like:

- Functional interfaces
- Method and constructor references
- Scope and effectively final variables
- Exception transparency
- The differences between lambdas and closures
- As we've talked about how lambdas aren't just syntactic sugar, we'll have a look at the bytecode lambdas produce

Functional interfaces

Java treats lambdas as an instance of an interface type. It formalizes this into something it calls *functional interfaces*. A functional interface is just an interface with a single method. Java calls the method a "functional method" but the name "single abstract method" or SAM is often used.

All the existing single method interfaces like `Runnable` and `Callable` in the JDK are now functional interfaces and lambdas can be used anywhere a single abstract method interface is used. In fact, it's functional interfaces that allow for what's called *target typing*; they provide enough information for the compiler to infer argument and return types.

@FunctionalInterface

Oracle have introduced a new annotation `@FunctionalInterface` to mark an interface as such. It's basically to communicate intent but also allows the compiler to do some additional checks.

For example, this interface compiles:

```
public interface FunctionalInterfaceExample {
    // compiles ok
}
```

but when you indicate that it should be a *functional interface* by adding the new annotation,

```
@FunctionalInterface // <- error here
    public interface FunctionalInterfaceExample {
      // doesn't compile
}
```

the compiler will raise an error. It tells us "Example is not a functional interface" as "no abstract method was found". Often the IDE will also hint, IntelliJ will say something like "no target method was found". It's hinting that we left off the functional method. A "single abstract method" needs a single, `abstract` method.

So what if we try and add a second method to the interface?

```
@FunctionalInterface
public interface FunctionalInterfaceExample {
    void apply();
    void illegal(); // <- error here
}
```

The compiler will error again, this time with a message along the lines of "multiple, non-overriding abstract methods were found". Functional interfaces can have only **one** method.

Extension

What about the case of an interfaces that extends another interface?

Let's create a new functional interface called `A` and another called `B` which extends `A`. The `B` interface is still a functional interface. It inherits the parents `apply` method as you'd expect:

```
@FunctionalInterface
interface A {
    abstract void apply();
}

interface B extends A {
}
```

If you wanted to make this clearer, you can also override the functional method from the parent:

```
@FunctionalInterface
interface A {
    abstract void apply();
}

interface B extends A {
    @Override
    abstract void apply();
}
```

We can verify it works as a functional interface if we use it as a lambda. We'll implement a method to show that a lambda can be assigned to a type of A and a type of B below. The implementation just prints out `"A"` or `"B"`.

```
@FunctionalInterface
public interface A {
    void apply();
}

public interface B extends A {
    @Override
    void apply();
}

public static void main(String... args) {
   A a = () -> System.out.println("A");
   B b = () -> System.out.println("B");
}
```

You can't add a new `abstract` method to the extending interface (B) though, as the resulting type would have two `abstract` methods and the IDE would warn us and the compiler would error.

```
@FunctionalInterface
public interface A {
    void apply();
}

public interface B extends A {
    void illegal();     // <- can't do this
}

public static void main(String... args) {
    A a = () -> System.out.println("A");
    B b = () -> System.out.println("B");    // <- error
}
```

In both cases, you can override methods from `Object` without causing problems. You can also add default methods (also new from Java 8 onward). As you'd probably expect, it doesn't make sense to try and mark an abstract class as a functional interface.

Other interface improvements

Interfaces generally have had some new features added, they include:

- Default methods (virtual extension methods)
- Static interface methods
- And a bunch of new functional interfaces in the `java.util.function` package; things like `Function` and `Predicate`

Summary

In this section we've talked about how any interface with a single method is now a "functional interface" and that the single method is often called a "functional method" or SAM (for single abstract method).

We looked at the new annotation and saw a couple of examples of how existing JDK interfaces like `Runnable` and `Callable` have been retrofitted with the annotation.

We also introduced the idea of *target typing* which is how the compiler can use the signature of a functional method to help work out what lambdas can be used where. We skimmed over this a little as we're going to talk about it later in the type inference section.

We discussed some examples of functional interfaces, how the compiler and IDE can help us out when we make mistakes and got a feel for the kinds of errors we might encounter. Things like adding more than one method to a functional interface. We also saw the exceptions to this rule, namely when we override methods from `Object` or implement default methods.

We had a quick look at interface inheritance and how that affects things and mentioned some of the other interface improvements that we'll be covering later.

An important point to take away here is the idea that any place a functional interface is used, you can now use a lambda. Lambdas can be used in-lieu of anonymous implementations of the functional interface. Using a lambda instead of the anonymous class may seem like syntactic sugar, but they're actually quiet different. See the *Functions vs. classes* section for more details.

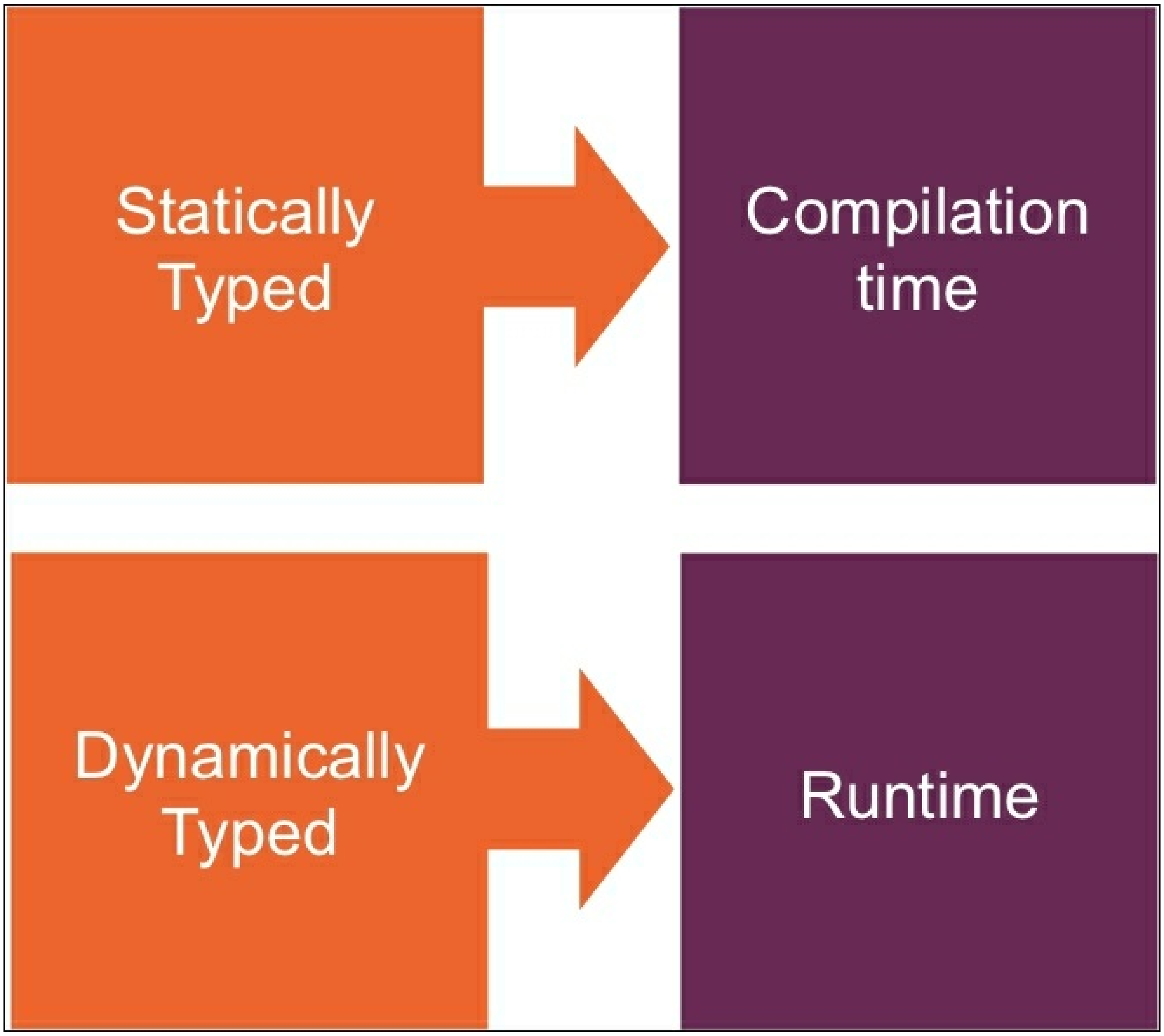
Type inference improvements

There have been several type inference improvements in modern Java. To be able to support lambdas, the way the compiler infers things has been improved to use *target typing* extensively. This and other improvements over Java 7's inference were managed under the Open **JDK Enhancement Proposal** (**JEP**) 101.

Before we get into those, lets recap on the basics.

Type inference refers to the ability for a programming language to automatically deduce the type of an expression.

Statically typed languages know the types of things at *compile* time. Dynamically typed languages know the types at *runtime*. A statically typed language can use type inference and drop type information in source code and use the compiler to figure out what's missing.



So this means that type inference can be used by statically typed languages (like Scala) to "look" like dynamic languages (like JavaScript). At least at the source code level.

Here's an example of a line of code in Scala:

```
val name = "Henry"
```

You don't need to tell the compiler explicitly that the value is a string. It figures it out. You could write it out explicitly like this,

```
val name : String = "Henry"
```

but there's no need.

As an aside, Scala also tries to figure out when you've finished a statement or expression based on it's **abstract syntax tree** (**AST**). So often, you don't even need to add a terminating semi-colon.

Java type inference

Type inference is a fairly broad topic, Java doesn't support the type of inference. I've just been talking about, at least for things like dropping the type annotations for variables. We have to keep that in:

```
String name = "Henry"; // <- we can't drop the String like Scala
```

So Java doesn't support type inference in the wider sense. It can't guess *everything* like some languages. Type inference for Java then typically refers to the way the compiler can work out types for generics. Java 7 improved this when it introduced the diamond operator (<>) but there are still lots of limitations in what Java *can* figure out.

The Java compiler was built with type erasure; it actively removes the type information during compilation. Because of type erasure, `List<String>` becomes `List<Object>` after compilation.

For historical reasons, when generics were introduced in Java 5, the developers couldn't easily reverse the decision to use erasure. Java was left with the need to understand what types to substitute for a given generic type but no information how to do it because it had all been erased. Type inference was the solution.

All generic values are really of type `Object` behind the scenes but by using type inference, the compiler can check that all the source code usages are consistent with what it thinks the generic should be. At runtime, everything is going to get passed around as instances of `Object` with appropriate casting behind the scenes. Type inference just allows the compiler to check that the casts would be valid ahead of time.

So type inference is about guessing the types, Java's support for type inferences was due to be improved in a couple of ways with Java 8:

1. Target-typing for lambdas.

and using generalized target-typing to:

1. Add support for parameter type inference in method calls.
2. Add support for parameter type inference in chained calls.

Lets have a look at the current problems and how modern Java addresses them.

Target-typing for lambdas

The general improvements to type inference in modern Java mean that lambdas can infer their type parameters; so rather than use:

```
(Integer x, Integer y) -> x + y;
```

you can drop the `Integer` type annotation and use the following instead:

```
(x, y) -> x + y;
```

This is because the functional interface describes the types, it gives the compiler all the information it needs.

For example, if we take an example functional interface.

```
@FunctionalInterface
interface Calculation {
    Integer apply(Integer x, Integer y);
}
```

When a lambda is used in-lieu of the interface, the first thing the compiler does is work out the *target* type of the lambda. So if we create a method `calculate` that takes the interface and two integers.

```
static Integer calculate(Calculation operation, Integer x, Integer y) {
    return operation.apply(x, y);
}
```

and then create two lambdas; an addition and subtraction function

```
Calculation addition = (x, y) -> x + y;
Calculation subtraction = (x, y) -> x - y;
```

and use them like this:

```
calculate(addition, 2, 2);
calculate(subtraction, 5, calculate(addition, 3, 2));
```

The compiler understands that the lambdas addition and subtraction have a target type of Calculation (it's the only "shape" that will fit the method signature of calculate). It can then use the method signature to infer the types of the lambda's parameters. There's only one method on the interface, so there's no ambiguity, the argument types are obviously `Integer`.

We're going to look at lots of examples of target typing so for now, just be aware that the mechanism Java uses to achieve lots of the lambda goodness relies on improvements to type inference and this idea of a *target* type.

Type parameters in method calls

The were some situations prior to Java 8 where the compiler couldn't infer types. One of these was when calling methods with generic type parameters as arguments.

For example, the `Collections` class has a generified method for producing an empty list. It looks like this:

```
public static final <T> List<T> emptyList() { ... }
```

In Java 7 this compiles:

```
List<String> names = Collections.emptyList(); // compiles in Java 7
```

as the Java 7 compiler can work out that the generic needed for the `emptyList` method is of type `String`. What it struggles with though is if the result of a generic method is passed as a parameter to another method call.

So if we had a method to process a list that looks like this:

```
static void processNames(List<String> names) {
    System.out.println("hello " + name);
}
```

and then call it with the empty list method.

```
processNames(Collections.emptyList()); // doesn't compile in Java 7
```

it won't compile because the generic type of the parameter has been erased to `Object`. It really looks like this.

```
processNames(Collections.<Object>emptyList names);
```

This doesn't match the `processList` method.

```
processNames(Collections.<String>emptyList names);
```

So it won't compile until we give it an extra hint using an explicit "type witness".

```
processNames(Collections.<String>emptyList());   // compiles in Java 7
```

Now the compiler knows enough about what generic type is being passed into the method.

The improvements in Java 8 include better support for this, so generally speaking where you would have needed a type witness, you no longer do.

Our example of calling the `processNames` now compiles!

```
processNames(Collections.emptyList());           // compiles in Java 8
```

Type parameters in chained method calls

Another common problem with type inference is when methods are chained together. Lets suppose we have a `List` class:

```
static class List<E> {

    static <T> List<T> emptyList() {
        return new List<T>();
    }

    List<E> add(E e) {
        // add element
        return this;
    }
}
```

and we want to chain a call to add an element to the method creating an empty list. Type erasure rears it's head again; the type is erased and so can't be known by the next method in the chain. It doesn't compile.

```
List<String> list = List.emptyList().add(":(");
```

This was due to be fixed in Java 8, but unfortunately it was dropped. So, at least for now, you'll still need to explicitly offer up a type to the compiler; you'll still need a type witness.

```
List<String> list = List.<String>emptyList().add(":(");
```

Method references

I mentioned earlier that method references are kind of like shortcuts to lambdas. They're a compact and convenient way to point to a method and allow that method to be used anywhere a lambda would be used.

When you create a lambda, you create an anonymous function and supply the method body. When you use a method reference as a lambda, it's actually pointing to a *named* method that already exists; it already has a body.

You can think of them as *transforming* a regular method into a functional interface.

The basic syntax looks like this:

```
Class::method
```

or, a more concrete example:

```
String::valueOf
```

The part preceding the double colon is the target reference and after, the method name. So, in this case, we're targeting the `String` class and looking for a method called `valueOf`; we're referring to the `static` method on `String`.

```
public static String valueOf(Object obj) { ... }
```

The double colon is called the *delimiter*. When we use it, we're not invoking the method, just *referencing* it. So remember not to add brackets on the end.

```
String::valueOf(); // <-- error
```

You can't invoke method references directly, they can only be used in-lieu of a lambda. So anywhere a lambda is used, you can use a method reference.

Example

This statement on it's own won't compile.

```
public static void main(String... args) {
    String::valueOf;
}
```

That's because the method reference can't be transformed into a lambda as there's no context for the compiler to infer what type of lambda to create.

We happen to know that this reference is equivalent to

```
(x) -> String.valueOf(x)
```

but the compiler doesn't know that yet. It can tell some things though. It knows, that as a lambda, the return value should be of type `String` because all methods called `valueOf` in `String` return a string. But it has no idea what to supply as a argument. We need to give it a little help and give it some more context.

We'll create a functional interface called `Conversion` that takes an integer and returns a string. This is going to be the target type of our lambda.

```
@FunctionalInterface
interface Conversion {
    String convert(Integer number);
}
```

Next, we need to create a scenario where we use this as a lambda. So we create a little method to take in a functional interface and apply an integer to it.

```
public static String convert(Integer number, Conversion function) {
    return function.convert(number);
}
```

Now, here's the thing. We've just given the compiler enough information to transform a method reference into the equivalent lambda.

When we call `convert` method, we can do so passing in a lambda.

```
convert(100, (number) -> String.valueOf(number));
```

And we can replace the lambda with a reference to the `valueOf` method. The compiler now knows we need a lambda that returns a string and takes an integer. It now knows that the `valueOf` method "fits" and can substitute the integer argument.

```
convert(100, String::valueOf);
```

Another way to give the compiler the information it needs is just to assign the reference to a type.

```
Conversion b = (number) -> String.valueOf(number);
```

and as a method reference

```
Conversion a = String::valueOf;
```

the "shapes" fit, so it can be assigned.

Interestingly, we can assign the same lambda to any interface that requires the same "shape". For example, if we have another functional interface with the same "shape",

Here, `Example` returns a `String` and takes an `Object` so it has the same signature shape as `valueOf`.

```
interface Example {
    String theNameIsUnimportant(Object object);
}
```

we can still assign the method reference (or lambda) to it.

```
Example a = String::valueOf;
```

Method reference types

There are four types of method reference:

- Constructor references
- Static method references
- And two types of instance method references

The last two are a little confusing. The first is a method reference of a particular object and the second is a method reference of an *arbitrary* object *but* of a particular type. The difference is in how you want to use the method and if you have the instance ahead of time or not.

Firstly then, lets have a look at constructor references.

Constructor reference

The basic syntax looks like this:

```
String::new
```

A *target* type followed by the double colon then the `new` keyword. It's going to create a lambda that will call the zero argument constructor of the `String` class.

It's equivalent to this lambda

```
() -> new String()
```

Remember that method references never have the parentheses; they're not invoking methods, just referencing one. This example is referring to the constructor of the `String` class but *not* instantiating a string.

Lets have a look at how we might actually use a constructor reference.

If we create a list of objects we might want to populate that list say ten items. So we could create a loop and add a new object ten times.

```
public void usage() {
    List<Object> list = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        list.add(new Object());
  }
}
```

but if we want to be able to reuse that initializing function, we could extract the code to a new method called `initialise` and then use a factory to create the object.

```
public void usage() {
    List<Object> list = new ArrayList<>();
    initialise(list, ...);
}

private void initialise(List<Object> list, Factory<Object> factory){
    for (int i = 0; i < 10; i++) {
        list.add(factory.create());
    }
 }
```

The `Factory` class is just a functional interface with a method called `create` that returns some object. We can then add the object it created to the list. Because it's a functional interface, we can use a lambda to implement the factory to initialize the list:

```
public void usage() {
    List<Object> list = new ArrayList<>();
    initialise(list, () -> new Object());
}
```

Or we could swap in a constructor reference.

```
public void usage() {
    List<Object> list = new ArrayList<>();
    initialise(list, Object::new);
}
```

There's a couple of other things we could do here. If we add some generics to the `initialise` method we can reuse it when initializing lists of any type. For example, we can go back and change the type of the list to be `String` and use a constructor reference to initialize it.

```
public void usage() {
    List<String> list = new ArrayList<>();
    initialise(list, String::new);
}

private <T> void initialise(List<T> list, Factory<T> factory) {
    for (int i = 0; i < 10; i++) {
        list.add(factory.create());
    }
}
```

We've seen how it works for zero argument constructors, but what about the case when classes have multiple argument constructors?

When there are multiple constructors, you use the same syntax but the compiler figures out which constructor would be the best match. It does this based on the *target* type and inferring functional interfaces that it can use to create that type.

Let's take the example of a `Person` class, it looks like this and you can see the constructor takes a bunch of arguments.

```
class Person {
    public Person(String forename, String surname, LocalDate
    birthday, Sex gender, String emailAddress, int age) {
      // ...
    }
}
```

Going back to our example from earlier and looking at the general purpose `initialise` method, we could use a lambda like this:

```
initialise(people, () -> new Person(forename, surname, birthday,
                                    gender, email, age));
```

but to be able to use a constructor reference, we'd need a lambda *with variable arguments* and that would look like this:

```
(a, b, c, d, e, f) -> new Person(a, b, c, d, e, f);
```

but this doesn't translate to a constructor reference directly. If we were to try and use

```
Person::new
```

it won't compile as it doesn't know anything about the parameters. If you try and compile it, the error says you've created an invalid constructor reference that cannot be applied to the given types; it found no arguments.

Instead, we have to introduce some indirection to give the compiler enough information to find an appropriate constructor. We can create something that can be used as a functional interface *and* has the right types to slot into the appropriate constructor.

Let's create a new functional interface called `PersonFactory`.

```
@FunctionalInterface
interface PersonFactory {
    Person create(String forename, String surname, LocalDate
    birthday, Sex gender, String emailAddress, int age);
}
```

Here, the arguments from `PersonFactory` match the available constructor on `Person`. Magically, this means we can go back and use it with a constructor reference of `Person`.

```
public void example() {
    List<Person> list = new ArrayList<>();
    PersonFactory factory = Person::new;
    // ...
}
```

Notice I'm using the constructor reference from `Person`. The thing to note here is that a constructor reference can be assigned to a target functional interface even though we don't yet know the arguments.

It may seem a bit strange that the type of the method reference is `PersonFactory` and not `Person`. This extra target type information helps the compiler to know it has to go via `PersonFactory` to create a `Person`. With this extra hint, the compiler is able to create a lambda based on the factory interface that will *later* create a `Person`.

Writing it out long hand, the compiler would generate this.

```
public void example() {
    PersonFactory factory = (a, b, c, d, e, f) -> new Person(a, b,
    c, d, e, f);
}
```

which could be used later like this:

```
public void example() {
    PersonFactory factory = (a, b, c, d, e, f) -> new Person(a, b,
    c, d, e, f);
    Person person = factory.create(forename, surname, birthday,
                                   gender, email, age);
}
```

Fortunately, the compiler can do this for us once we've introduced the indirection.

It understands the target type to use is `PersonFactory` and it understands that it's single `abstract` method can be used in-lieu of a constructor. It's kind of like a two step process, firstly, to work out that the `abstract` method has the same argument list as a constructor and that it returns the right type, then apply it with colon colon new syntax.

To finish off the example, we need to tweak our `initialise` method to add the type information (replace the generics), add parameters to represent the person's details and actually invoke the factory.

```
private void initialise(List<Person> list, PersonFactory factory,
                        String forename, String surname,
                        LocalDate birthday, Sex gender,
                        String emailAddress, int age) {
                          for (int i = 0; i < 10; i++) {
                            list.add(factory.create(forename,
                            surname, birthday, gender,
                            emailAddress, age));
                          }
                        }
```

and then we can use it like this:

```
public void example() {
    List<Person> list = new ArrayList<>();
    PersonFactory factory = Person::new;
    initialise(people, factory, a, b, c, d, e, f);
}
```

or inline, like this:

```
public void example() {
    List<Person> list = new ArrayList<>();
    initialise(people, Person::new, a, b, c, d, e, f);
}
```

Static method reference

A method reference can point directly to a static method. For example,

```
String::valueOf
```

This time, the left hand side refers to the type where a static method, in this case `valueOf`, can be found. It's equivalent to this lambda

```
x -> String.valueOf(x))
```

A more extended example would be where we sort a collection using a reference to a static method on the class `Comparators`.

```
Collections.sort(Arrays.asList(5, 12, 4), Comparators::ascending);

// equivalent to
Collections.sort(Arrays.asList(5, 12, 4), (a, b) -> Comparators.ascending(a,
b));
```

where, the static method `ascending` might be defined like this:

```
public static class Comparators {
    public static Integer ascending(Integer first, Integer second)
    {
        return first.compareTo(second);
     }
}
```

Instance method reference of particular object (in this case, a closure)

Here's an example of an instance method reference of a specific instance:

```
x::toString
```

The `x` is a specific instance that we want to get at. It's lambda equivalent looks like this:

```
() -> x.toString()
```

The ability to reference the method of a specific instance also gives us a convenient way to convert between different functional interface types. For example:

```
Callable<String> c = () -> "Hello";
```

Callable's functional method is call. When it's invoked the lambda will return `"Hello"`.

If we have another functional interface, `Factory`, we can convert the `Callable` using a method reference.

```
Factory<String> f = c::call;
```

We could have just re-created the lambda but this trick is a useful way to get reuse out of predefined lambdas. Assign them to variables and reuse them to avoid duplication.

Here's an example of it in use:

```
public void example() {
    String x = "hello";
    function(x::toString);
}
```

This is an example where the method reference is using a closure. It creates a lambda that will call the `toString` method on the instance `x`.

The signature and implementation of function above looks like this:

```
public static String function(Supplier<String> supplier) {
    return supplier.get();
}
```

The `Supplier` interface is a functional interface that looks like this:

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

When used in our function, it provides a string value (via the call to `get`) and the only way it can do that is if the value has been supplied to it on construction. It's equivalent to:

```
public void example() {
  String x = "";
  function(() -> x.toString());
}
```

Notice here that the lambda has no arguments (it uses the "hamburger" symbol). This shows that the value of `x` isn't available in the lambda's local scope and so can only be available from outside it's scope. It's a closure because must close over `x` (it will *capture* `x`).

If you're interested in seeing the long hand, anonymous class equivalent, it'll look like this. Notice again how `x` must be passed in:

```
public void example() {
    String x = "";
    function(new Supplier<String>() {
        @Override
        public String get() {
            return x.toString(); // <- closes over 'x'
        }
    });
}
```

All three of these are equivalent. Compare this to the lambda variation of an instance method reference where it doesn't have it's argument explicitly passed in from an outside scope.

Instance method reference of a arbitrary object whose instance is supplied later (lambda)

The last case is for a method reference that points to an arbitrary object referred to by its type:

```
Object::toString
```

So in this case, although it looks like the left hand side is pointing to a class (like the `static` method reference), it's actually pointing to an instance. The `toString` method is an instance method on `Object`, not a `static` method. The reason why you might not use the regular instance method syntax is because you may not yet have an instance to refer to.

So before, when we call `x` colon colon `toString`, we know the value of `x`. There are some situations where you don't have a value of `x` and in these cases, you can still pass around a reference to the method but supply a value later using this syntax.

For example, the lambda equivalent doesn't have a bound value for `x`.

```
(x) -> x.toString()
```

There difference between the two types of instance method reference is basically academic. Sometimes, you'll need to pass something in, other times, the usage of the lambda will supply it for you.

The example is similar to the regular method reference; it calls the `toString` method of a string only this time, the string is supplied to the function that's making use of the lambda and not passed in from an outside scope.

```
public void lambdaExample() {
    function("value", String::toString);
}
```

The `String` part looks like it's referring to a class but it's actually referencing an instance. It's confusing, I know but to see things more clearly, we need to see the function that's making use of the lambda. It looks like this.

```
public static String function(String value, Function<String, String> function) {
    return function.apply(value);
}
```

So, the string value is passed directly to the function, it would look like this as a fully qualified lambda.

```
public void lambdaExample() {
    function("value", x -> x.toString());
}
```

which Java can shortcut to look like `String::toString`; it's saying "supply the object instance" at runtime.

If you expand it fully to an anonymous interface, it looks like this. The `x` parameter is made available and not closed over. Hence it being a lambda rather than a closure.

```
public void lambdaExample() {
    function("value", new Function<String, String>() {
      @Override
      // takes the argument as a parameter, doesn't need to close
      over it
      public String apply(String x) {
        return x.toString();
      }
    });
}
```

Summary

Oracle describe the four kinds of method reference(http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html) as follows:

Kind	**Example**
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`ContainingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

But the instance method descriptions are just plain confusing. What on earth is an instance method of an arbitrary object of a particular type? Aren't all objects *of a* particular type? Why is it important that the object is *arbitrary*?

I prefer to think of the first as an instance method of a *specific* object known ahead of time and the second as an instance method of an arbitrary object that will be *supplied* later. Interestingly, this means the first is a *closure* and the second is a *lambda*. One is *bound* and the other *unbound*. The distinction between a method reference that closes over something (a closure) and one that doesn't (a lambda) may be a bit academic but at least it's a more formal definition than Oracle's unhelpful description.

Kind	**Syntax**	**Example**
Reference to a static method	`Class::staticMethodName`	`String::valueOf`
Reference to an instance method of a specific object	`object::instanceMethodName`	`x::toString`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`String::toString`
Reference to a constructor	`ClassName::new`	`String::new`

or as equivalent lambdas:

Kind	**Syntax**	**As Lambda**
Reference to a static method	`Class::staticMethodName`	`(s) -> String.valueOf(s)`

Reference to an instance method of a specific object	`object::instanceMethodName`	`() -> "hello".toString()`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`(s) -> s.toString()`
Reference to a constructor	`ClassName::new`	`() -> new String()`

Note that the syntax for a `static` method reference looks very similar to a reference to an instance method of a class. The compiler determines which to use by going through each applicable static method and each applicable instance method. If it were to find a match for both, the result would be a compiler error.

You can think of the whole thing as a transformation from a method reference to a lambda. The compiler provides the *transformation* function that takes a method reference and target typing and can derive a lambda.

Scoping

The good news with lambdas is that they don't introduce any new scoping. Using variables within a lambda will refer to variables residing in the enclosing environment.

This is what's called **lexical scoping**. It means that lambdas don't introduce a new level of scoping at all; you can directly access fields, methods and variables from the enclosing scope. It's also the case for the **this** and **super** keywords. So we don't have to worry about the crazy nested class syntax for resolving scope.

Let's take a look at an example. We have an example class here, with a member variable `i` set to the value of `5`.

```
public static class Example {
    int i = 5;

    public Integer example() {
        Supplier<Integer> function = () -> i * 2;
        return function.get();
    }
}
```

In the `example` method, a lambda uses a variable called `i` and multiplies it by two.

Because lambdas are lexically scoped, `i` simply refers to the enclosing classes' variable. It's value at run-time will be 5. Using this drives home the point; this within a lambda is the same as without.

```
public static class Example {
    int i = 5;
    public Integer example() {
        Supplier<Integer> function = () -> this.i * 2;
        return function.get();
    }
}
```

In the `anotherExample` method below, a method parameter is used which is also called `i`. The usual shadowing rules kick in here and `i` will refer to the method parameter and not the class member variable. The method variable *shadows* the class variable. It's value will be whatever is passed into the method.

```
public static class Example {
    int i = 5;

    public Integer anotherExample(int i) {
        Supplier<Integer> function = () -> i * 2;
        return function.get();
    }
}
```

If you wanted to refer to the class variable `i` and not the parameter `i` from within the body, you could make the variable explicit with this. For example, `Supplier<Integer> function = () -> i * 2;`.

The following example has a locally scoped variable defined within the `yetAnotherExample` method. Remember that lambdas use their enclosing scope as their own, so in this case, `i` within the lambda refers to the method's variable; `i` will be `15` and not `5`.

```
public static class Example {
    int i = 5;

    public Integer yetAnotherExample() {
        int i = 15;
        Supplier<Integer> function = () -> i * 2;
        return function.get();
    }
}
```

If you want to see this for yourself, you could use a method like the following to print out the values:

```
public static void main(String... args) {
    Example scoping = new Example();
    System.out.println("class scope        = " +
    scoping.example());
    System.out.println("method param scope = " +
    scoping.anotherExample(10));
    System.out.println("method scope       = " +
    scoping.yetAnotherExample());
}
```

The output would look like this:

```
class scope        = 10
method param scope = 20
method scope       = 30
```

So, the first method prints `10`; 5 from the class variable multiplied by two. The second method prints `20` as the parameter value was 10 and was multiplied by two and the final method prints `30` as the local method variable was set to 15 and again multiplied by two.

Lexical scoping means deferring to the enclosing environment. Each example had a different enclosing environment or scope. You saw a variable defined as a class member, a method parameter and locally from within a method. In all cases, the lambda behaved consistently and referenced the variable from these enclosing scopes.

Lambda scoping should be intuitive if you're already familiar with basic Java scoping, there's really nothing new here.

Effectively final

In Java 7, any variable passed into an anonymous class instance would need to be made final.

This is because the compiler actually copies all the context or *environment* it needs into the instance of the anonymous class. If those values were to change under it, unexpected side effects could happen. So Java insists that the variable be final to ensure it doesn't change and the inner class can operate on them safely. By safely, I mean without race conditions or visibility problems between threads.

Let's have a look at an example. To start with, we'll use Java 7 and create a method called `filter` that takes a list of people and a predicate. We'll create a temporary list to contain any matches we find then enumerate each element testing to see if the predicate holds true for each person. If the test is positive, we'll add them to the temporary list before returning all matches.

```
// java 7
private List<Person> filter(List<Person> people, Predicate<Person> predicate) {
    ArrayList<Person> matches = new ArrayList<>();
    for (Person person : people)
        if (predicate.test(person))
            matches.add(person);
    return matches;
}
```

Then we'll create a method that uses this to find all the people in a list that are eligible for retirement. We set a retirement age variable and then call the `filter` method with an arbitrary list of people and a new anonymous instance of a `Predicate` interface.

We'll implement this to return true if a person's age is greater than or equal to the retirement age variable.

```
 public void findRetirees() {
     int retirementAge = 55;
     List<Person> retirees = filter(allPeople, new
     Predicate<Person>
     () {
         @Override
         public boolean test(Person person) {
             return person.getAge() >= retirementAge; // <--
             compilation error
         }
     });
 }
```

If you try and compile this, you'll get a compiler failure when accessing the variable. This is because the variable isn't final. We'd need to add `final` to make it compile.

```
final int retirementAge = 55;
```

Note

Passing the environment into an anonymous inner class like this is an example of a closure. The environment is what a closure "closes" over; it has to capture the variables it needs to do its job. The Java compiler achieves this using the copy trick rather than try and manage multiple changes to the same variable. In the context of closures, this is called *variable capture*.

Java 8 introduced the idea of "effectively final" which means that if the compiler can work out that a particular variable is *never* changed, it can be used where ever a final variable would have be used. It interprets it as "effectively" final.

In our example, if we switch to Java 8 and drop the `final` keyword. Things still compile. No need to make the variable final. Java knows that the variable doesn't change so it makes it effectively final.

```
int retirementAge = 55;
```

Of course, it still compiles if you were to make it `final`.

But how about if we try and modify the variable after we've initialized it?

```
int retirementAge = 55;
// ...
retirementAge = 65;
```

The compiler spots the change and can no longer treat the variable as effectively final. We get the original compilation error asking us to make it final. Conversely, if adding the `final` keyword to a variable declaration doesn't cause a compiler error, then the variable is effectively final.

I've been demonstrating the point here with an anonymous class examples because the idea of effectively final isn't something specific to lambdas. It is of course applicable to lambdas though. You can convert this anonymous class above into a lambda and nothing changes. There's still no need to make the variable final.

Circumventing final

You can still get round the safety net by passing in final objects or arrays and then change their internals in your lambda.

For example, taking our list of people, lets say we want to sum all their ages. We could create a method to loop and sum like this:

```
private static int sumAllAges(List<Person> people) {
    int sum = 0;
    for (Person person : people) {
        sum += person.getAge();
    }
    return sum;
}
```

where the sum count is maintained as the list is enumerated. As an alternative, we could try and abstract the looping behavior and pass in a function to be applied to each element. Like this:

```
public final static Integer forEach(List<Person> people, Function<Integer,
Integer> function) {
  Integer result = null;
  for (Person t : people) {
    result = function.apply(t.getAge());
  }
  return result;
}
```

and to achieve the summing behavior, all we'd need to do is create a function that can sum. You could do this using an anonymous class like this:

```
private static void badExample() {
    Function<Integer, Integer> sum = new Function<Integer, Integer>
   () {
        private Integer sum = 0;

        @Override
        public Integer apply(Integer amount) {
            sum += amount;
            return sum;
        }
   };
}
```

Where the functions method takes an integer and returns an integer. In the implementation, the sum variable is a class member variable and is mutated each time the function is applied. This kind of mutation is generally bad form when it comes to functional programming.

Nether the less, we can pass this into our `forEach` method like this:

```
forEach(allPeople, sum);
```

and we'd get the sum of all peoples ages. This works because we're using the same instance of the function so the `sum` variable is reused and mutated during each iteration.

The bad news is that we can't convert this into a lambda directly; there's no equivalent to a member variable with lambdas, so there's nowhere to put the `sum` variable other than outside of the lambda.

```
double sum = 0;
forEach(allPeople, x -> {
    return sum += x;
});
```

but this highlights that the variable isn't effectively final (it's changed in the lambda's body) and so it must be made final.

But if we make it final

```
final double sum = 0;
forEach(allPeople, x -> {
    return sum += x;
});
```

we can no longer modify it in the body! It's a chicken and egg situation.

The trick around this is to use a object or an array; it's reference can remain final but it's internals can be modified

```
int[] sum = {0};
forEach(allPeople, x -> sum[0] += x);
```

The array reference is indeed final here, but we can modify the array contents without reassigning the reference. However, this is generally bad form as it opens up to all the safety issues we talked about earlier. I mention it for illustration purposes but don't recommend you do this kind of thing often. It's generally better not to create functions with side effects and you can avoid the issues completely, if you use a more functional approach. A more idiomatic way to do this kind of summing is to use what's called a *fold* or in the Java vernacular *reduce*.

Exception handling

There's no new syntax for exception handling in lambdas. Exceptions thrown in a lambda are propagated to the caller, just as you'd expect with a regular method call. There's nothing special about calling lambdas or handling their exceptions.

However, there are some subtleties that you need to be aware of. Firstly, as a caller of a lambda, you are potentially unaware of what exceptions might be thrown, if any and secondly, as the author of a lambda, you're potentially unaware what context your lambda will be run in.

When you create a lambda, you typically give up responsibility of how that lambda will be executed to the method that you pass it to. For all you know, your lambda may be run in parallel or at some point in the future and so any exception you throw may not get handled as you might expect. You can't rely on exception handling as a way to control your program's flow.

To demonstrate this, lets write some code to call two things, one after the other. We'll use `Runnable` as a convenient lambda type.

```
public static void runInSequence(Runnable first, Runnable second) {
    first.run();
    second.run();
}
```

If the first call to run were to throw an exception, the method would terminate and the second method would never be called. The caller is left to deal the exception. If we use this method to transfer money between two bank accounts, we might write two lambdas. One for the debit action and one for the credit.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> a.debit(amount);
    Runnable credit = () -> b.credit(amount);
 }
```

we could then call our `runInSequence` method like this:

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> a.debit(amount);
    Runnable credit = () -> b.credit(amount);
    runInSequence(debit, credit);
 }
```

any exceptions could be caught and dealt with by using a `try/catch` like this:

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> a.debit(amount);
    Runnable credit = () -> b.credit(amount);
    try {
        runInSequence(debit, credit);
    } catch (Exception e) {
        // check account balances and rollback
    }
  }
```

Here's the thing. As an author of the lambdas, I potentially have no idea how `runInSequence` is implemented. It may well be implemented to run asynchronously like this:

```
public static void runInSequence(Runnable first, Runnable second) {
    new Thread(() -> {
        first.run();
        second.run();
    }).start();
}
```

In which case any exception in the first call would terminate the thread, the exception would disappear to the default exception handler and our original client code wouldn't get the chance to deal with the exception.

Using a callback

Incidentally, one way round the specific problem with raising an exception on a different thread than the caller can be addressed with a callback function. Firstly, you'd defend against exceptions in the `runInSequence` method:

```
public static void runInSequence(Runnable first, Runnable second) {
    new Thread(() -> {
        try {
            first.run();
            second.run();
        } catch (Exception e) {
            // ...
        }
    }).start();
}
```

Then introduce an exception handler which can be called in the event of an exception:

```
public static void runInSequence(Runnable first, Runnable second,
        Consumer<Throwable> exceptionHandler) {
    new Thread(() -> {
        try {
            first.run();
            second.run();
        } catch (Exception e) {
            exceptionHandler.accept(e);
        }
    }).start();
}
```

Consumer is a functional interface (new in Java 8) that in this case takes the exception as an argument to it's `accept` method.

When we wire this up to the client, we can pass in a callback lambda to handle any exception.

```
public void nonBlockingTransfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> a.debit(amount);
    Runnable credit = () -> b.credit(amount);
    runInSequence(debit, credit, (exception) -> {
      /* check account balances and rollback */
    });
}
```

This is a good example of deferred execution and so has it's own foibles. The exception handler method may (or may not) get executed at some later point in time. The `nonBlockingTransfer` method will have finished and the bank accounts themselves may be in some other state by the time it fires. You can't rely on the exception handler being called when it's convenient for you; we've opened up a whole can of concurrency worms.

Dealing with exceptions when writing lambdas

Let's look at dealing with exceptions from the perspective of a lambda author, someone writing lambdas. After this, we'll look at dealing with exceptions when calling lambdas.

Lets look at it as if we wanted to implement the `transfer` method using lambdas but this time wanted to reuse an existing library that supplies the `runInSequence` method.

Before we start, let's take a look at the `BankAccount` class. You'll notice that this time, the `debit` and `credit` methods both throw a checked exception; `InsufficientFundsException`.

```
class BankAccount {
    public void debit(int amount) throws InsufficientFundsException
    {
        // ...
     }

     public void credit(int amount) throws
     InsufficientFundsException
     {
         // ...
      }
}

class InsufficientFundsException extends Exception { }
```

Let's recreate the `transfer` method. We'll try to create the debit and credit lambdas and pass these into the `runInSequence` method. Remember that the `runInSequence` method was written by some library author and we can't see or modify it's implementation.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> a.debit(amount);   <- compiler error
    Runnable credit = () -> b.credit(amount); <- compiler error
    runInSequence(debit, credit);
 }
```

The debit and credit both throw a checked exception, so this time, you can see a compiler error. It makes no difference if we add this to the method signature; the exception would happen inside the lambda. Remember I said exceptions in lambdas are propagated to the caller? In our case, this will be the `runInSequence` method and not the point we define the lambda. The two aren't communicating between themselves that there could be an exception raised.

```
// still doesn't compile
public void transfer(BankAccount a, BankAccount b, Integer amount)
       throws InsufficientFundsException {
    Runnable debit = () -> a.debit(amount);
    Runnable credit = () -> b.credit(amount);
    runInSequence(debit, credit);
}
```

So if we can't force a checked exception to be *transparent* between the lambda and the caller, one option is to wrap the checked exception as a runtime exception like this:

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> {
        try {
            a.debit(amount);
        } catch (InsufficientFundsException e) {
            throw new RuntimeException(e);
        }
   };
   Runnable credit = () -> {
       try {
           b.credit(amount);
       } catch (InsufficientFundsException e) {
           throw new RuntimeException(e);
       }
   };
   runInSequence(debit, credit);
 }
```

That gets us out of the compilation error but it's not the full story yet. It's very verbose and we still have to catch and deal with, what's now a runtime exception, around the call to `runInSequence`.

```
public void transfer(BankAccount a, BankAccount b, Integer amount){
    Runnable debit = () -> { ... };
    };
    Runnable credit = () -> { ... };
    try {
        runInSequence(debit, credit);
    } catch (RuntimeException e) {
        // check balances and rollback
    }
}
```

There's still one or two niggles though; we're throwing and catching a `RuntimeException` which is perhaps a little loose. We don't really know what other exceptions, if any, might be thrown in the `runInSequence` method. Perhaps it's better to be more explicit. Let's create a new sub-type of `RuntimeException` and use that instead.

```
class InsufficientFundsRuntimeException extends RuntimeException {
    public
    InsufficientFundsRuntimeException(InsufficientFundsException
    cause) {
        super(cause);
    }
}
```

After we've modified the original lambda to throw the new exception, we can restrict the catch to deal with only exceptions we know about; namely the `InsufficientFundsRuntimeException`.

We can now implement some kind of balance check and rollback functionality, confident that we understand all the scenarios that can cause it.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = () -> {
        try {
            a.debit(amount);
        } catch (InsufficientFundsException e) {
            throw new InsufficientFundsRuntimeException(e);
        }
   };
   Runnable credit = () -> {
       try {
           b.credit(amount);
       } catch (InsufficientFundsException e) {
           throw new InsufficientFundsRuntimeException(e);
       }
   };
   try {
       runInSequence(debit, credit);
   } catch (InsufficientFundsRuntimeException e) {
       // check balances and rollback
   }
 }
```

The trouble with all this, is that the code has more exception handling boilerplate than actual business logic. Lambdas are supposed to make things less verbose but this is just full of noise. We can make things better if we generalize the wrapping of checked exceptions to runtime equivalents. We could create a functional interface that captures an exception type on the signature using generics.

Let's name it `Callable` and its single method; `call`. Don't confuse this with the class of the same name in the JDK; we're creating a new class to illustrate dealing with exceptions.

```
@FunctionalInterface
interface Callable<E extends Exception> {
    void call() throws E;
}
```

We'll change the old implementation of transfer and create lambdas to match the "shape" of the new functional interface. I've left off the type for a moment.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    ??? debit = () -> a.debit(amount);
    ??? credit = () -> b.credit(amount);
 }
```

Remember from the type inference section that Java would be able to see this as a type of `Callable` as it has no parameters as does `Callable`, it has the same return type (none) and throws an exception of the same type as the interface. We just need to give the compiler a hint, so we can assign this to an instance of a `Callable`.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Callable<InsufficientFundsException> debit = () ->
    a.debit(amount);
    Callable<InsufficientFundsException> credit = () ->
    b.credit(amount);
 }
```

Creating the lambdas like this doesn't cause a compilation error as the functional interface declares that it could be thrown. It doesn't need to warn us at the point we create the lambda, as the signature of the functional method will cause the compiler to error if required when we actually try and call it. Just like a regular method.

If we try and pass them into the `runInSequence` method, we'll get a compiler error though.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Callable<InsufficientFundsException> debit = () ->
    a.debit(amount);
    Callable<InsufficientFundsException> credit = () ->
    b.credit(amount);
    runInSequence(debit, credit); <- doesn't compile
 }
```

The lambdas are of the wrong type. We still need a lambda of type `Runnable`. We'll have to write a method that can convert from a `Callable` to a `Runnable`. At the same time, we'll wrap the checked exception to a run time one. Something like this:

```
public static Runnable unchecked(Callable<InsufficientFundsException> function)
{
    return () -> {
        try {
            function.call();
        } catch (InsufficientFundsException e) {
            throw new InsufficientFundsRuntimeException(e);
        }
    };
}
```

All that's left to do is wire it in for our lambdas:

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = unchecked(() -> a.debit(amount));
    Runnable credit = unchecked(() -> b.credit(amount));
    runInSequence(debit, credit);
 }
```

Once we put the exception handling back in we're back to a more concise method body and have dealt with the potential exceptions in the same way as before.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    Runnable debit = unchecked(() -> a.debit(amount));
    Runnable credit = unchecked(() -> b.credit(amount));
    try {
        runInSequence(debit, credit);
    } catch (InsufficientFundsRuntimeException e) {
        // check balances and rollback
    }
}
```

The downside is this isn't a totally generalized solution; we'd still have to create variations of the unchecked method for different functions. We've also just hidden the verbose syntax away. The verbosity is still there its just been moved. Yes, we've got some reuse out of it but if exception handling were transparent or we didn't have checked exceptions, we wouldn't need to brush the issue under the carpet quite so much.

It's worth pointing out that we'd probably end up doing something similar if we were in Java 7 and using anonymous classes instead of lambdas. A lot of this stuff can still be done pre-Java 8 and you'll end up creating helper methods to push the verbosity to one side.

It's certainly the case that lambdas offer more concise representations for small anonymous pieces of functionality but because of Java's checked exception model, dealing with exceptions in lambdas will often cause all the same verbosity problems we had before.

As a caller (dealing with exceptions when calling lambdas)

We've seen things from the perspective of writing lambdas, now lets have a look at things when calling lambdas.

Let's imagine that now we're writing the library that offers the `runInSequence` method. We have more control this time and aren't limited to using `Runnable` as a lambda type. Because we don't want to force our clients to jump through hoops dealing with exceptions in their lambdas (or wrap them as runtime exceptions), we'll provide a functional interface that declares that a checked exception might be thrown.

We'll call it `FinancialTransfer` with a `transfer` method:

```
@FunctionalInterface
interface FinancialTransfer {
    void transfer() throws InsufficientFundsException;
}
```

We're saying that whenever a banking transaction occurs, there's the possibility that insufficient funds are available. Then when we implement our `runInSequence` method, we accept lambdas of this type.

```
public static void runInSequence(FinancialTransfer first,
        FinancialTransfer second) throws InsufficientFundsException
{
    first.transfer();
    second.transfer();
 }
```

This means that when clients use the method, they're not forced to deal with exceptions within their lambdas. For example, writing a method like this.

```
// example client usage
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    FinancialTransfer debit = () -> a.debit(amount);
    FinancialTransfer credit = () -> b.credit(amount);
 }
```

This time there is no compiler error when creating the lambdas. There's no need to wrap the exceptions from `BankAccount` methods as runtime exceptions; the functional interface has already declared the exception. However, `runInSequence` would now throw a checked exception, so it's explicit that the client has to deal with the possibility and you'll see a compiler error.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    FinancialTransfer debit = () -> a.debit(amount);
    FinancialTransfer credit = () -> b.credit(amount);
    runInSequence(debit, credit);  <- compiler error
 }
```

So we need to wrap the call in a `try/catch` to make the compiler happy:

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    FinancialTransfer debit = () -> a.debit(amount);
    FinancialTransfer credit = () -> b.credit(amount);
    try {
        runInSequence(debit, credit);  <- compiler error
    } catch (InsufficientFundsException e) {
        // whatever
    }
 }
```

The end result is something like we saw previously but without the need for the unchecked method. As a library developer, we've made it easier for clients to integrate with our code.

But what about if we try something more exotic? Let's make the `runInSequence` method asynchronous again. There's no need to throw the exception from the method signature as it wouldn't propagate to the caller if it were thrown from a different thread. So this version of the `runInSequence` method doesn't include the throws clause and the `transfer` method is no longer forced to deal with it. However, the calls to `.transfer` will still throw an exception.

```
public static void runInSequence(Runnable first, Runnable second) {
    new Thread(() -> {
        first.transfer();   <- compiler error
        second.transfer();  <- compiler error
    }).start();
}

public void transfer(BankAccount a, BankAccount b, Integer amount) {
    FinancialTransfer debit = () -> a.debit(amount);
    FinancialTransfer credit = () -> b.credit(amount);
    runInSequence(debit, credit);  <- compiler error
 }
```

With the compiler errors still in the `runInSequence` method, we need another way to handle the exception. One technique is to pass in a function that will be called in the event of an exception. We can use this lambda to bridge the code running asynchronously back to the caller.

To start with, we'll add the `catch` block back in and pass in a functional interface to use as the exception handler. I'll use the `Consumer` interface here, it's new in Java 8 and part of the `java.util.function` package. We then call the interface method in the catch block, passing in the cause.

```
public void runInSequence(FinancialTransfer first,
       FinancialTransfer second,
       Consumer<InsufficientFundsException> exceptionHandler) {
    new Thread(() -> {
        try {
            first.transfer();
            second.transfer();
        } catch (InsufficientFundsException e) {
            exceptionHandler.accept(e);
        }
    }).start();
}
```

To call it, we need to update the `transfer` method to pass in a lambda for the callback. The parameter, exception below, will be whatever is passed into the `accept` method in `runInSequence`. It will be an instance of `InsufficientFundsException` and the client can deal with it however they chose.

```
public void transfer(BankAccount a, BankAccount b, Integer amount) {
    FinancialTransfer debit = () -> a.debit(amount);
    FinancialTransfer credit = () -> b.credit(amount);
    Consumer<InsufficientFundsException> handler = (exception) -> {
        /* check account balances and rollback */
   };
   runInSequenceNonBlocking(debit, credit, handler);
}
```

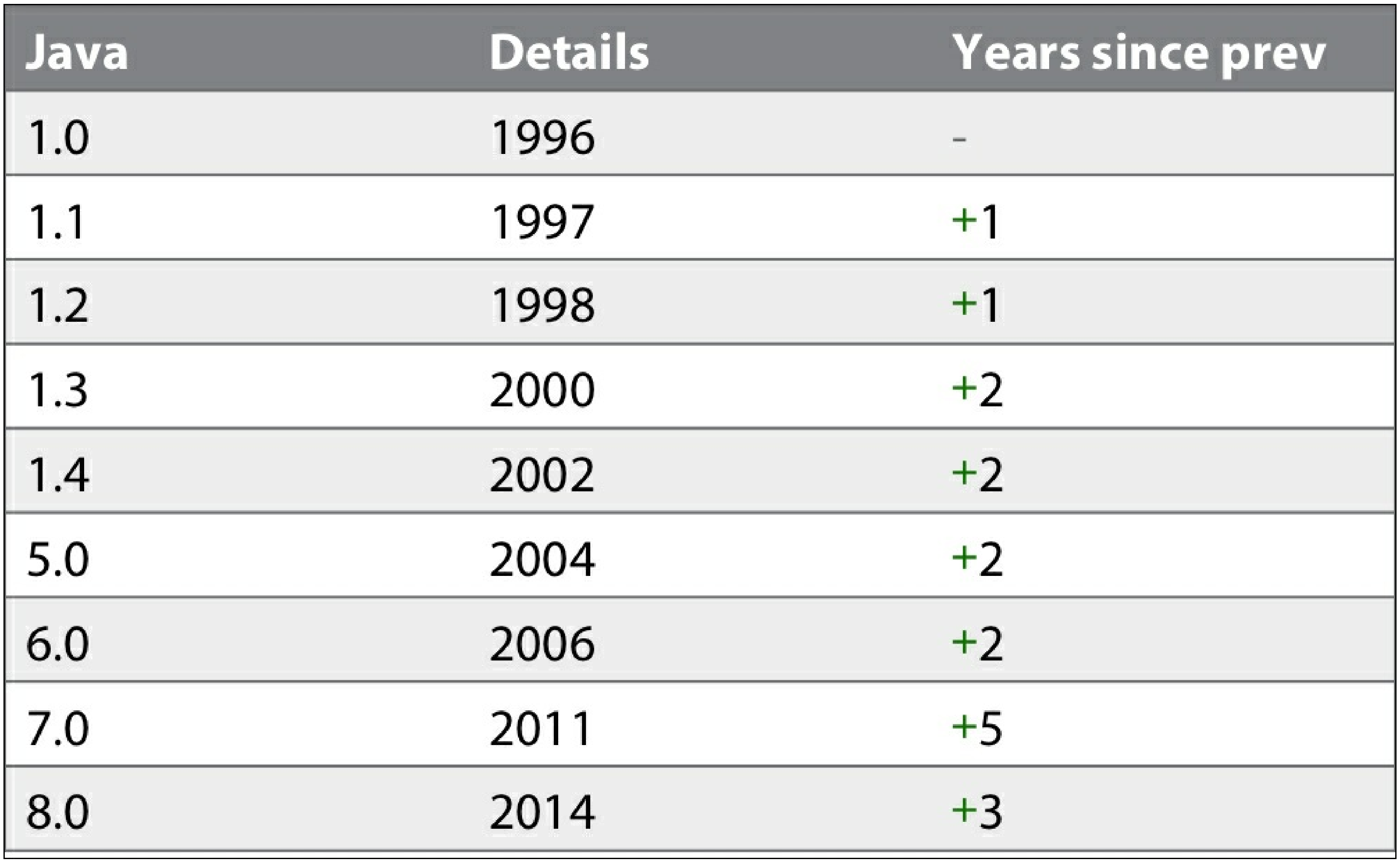
There we are. We've provided the client to our library with an alternative exception handling mechanism rather than forcing them to catch exceptions.

We've internalized the exception handling into the library code. It's a good example of deferred execution; should there be an exception, the client doesn't necessarily know when his exception handler would get invoked. For example, as we're running in another thread, the bank accounts themselves may have be altered by the time it executes. Again it highlights that using exceptions to control your program's flow is a flawed approach. You can't rely on the exception handler being called when it's convenient for you.

Lambdas vs closures

The terms *closure* and *lambda* are often used interchangeably but they are actually distinct. In this section we'll take a look at the differences so you can be clear about which is which.

Below is a table showing the release dates for each major version of Java. Java 5.0 came along in 2004 and included the first major language changes including things like generics support:



Java	Details	Years since prev
1.0	1996	-
1.1	1997	+1
1.2	1998	+1
1.3	2000	+2
1.4	2002	+2
5.0	2004	+2
6.0	2006	+2
7.0	2011	+5
8.0	2014	+3

Around 2008 to 2010 there was a lot of work going on to introduce closures to Java. It was due to go in to Java 7 but didn't quite make it in time. Instead it evolved into lambda support in Java 8. Unfortunately, around that time, people used the term "closures" and "lambdas" interchangeably and so it's been a little confusing for the Java community since. In fact, there's still a project page on the OpenJDK site for closures and one for lambdas.

From the OpenJDK project's perspective, they really should have been using "lambda" consistently from the start. In fact, the OpenJDK got it so wrong, they ignored the fact that Java *has had* closure support since version 1.1!

I'm being slightly pedantic here as although there are technical differences between closures and lambdas, the goals of the two projects were to achieve the same thing, even if they used the terminology inconsistently.

So what is the difference between lambdas and closures? Basically, a closure *is a* type of lambda but a lambda isn't necessarily a closure.

Basic differences

Just like a lambda, a closure is effectively an anonymous block of functionality, but there are some important distinctions. A closure depends on external values (not just it's arguments) whereas a lambda depends only on it's arguments. A closure is said to "close over" the environment it requires.

For example, the following:

```
(server) -> server.isRunning();
```

is a lambda, but this

```
() -> server.isRunning();
```

is a closure.

They both return a boolean indicating if some server is up but one uses it's argument and the other must get the variable from somewhere else. Both are lambdas; in the general sense, they are both anonymous blocks of functionality and in the Java language sense, they both use the new lambda syntax.

The first example refers to a server variable passed into the lambda as an argument whereas the second example (the closure) gets the server variable from somewhere else; that is the environment. To get the instance of the variable, the lambda has to "close over" the environment or capture the value of server. We've seen this in action when we talked about `effectively final` before.

Let's expand the example to see things more clearly. Firstly, we'll create a method in a static class to perform a naive poll and wait. It'll check a functional interface on each poll to see if some condition has been met.

```
class WaitFor {
    static <T> void waitFor(T input, Predicate<T> predicate)
            throws InterruptedException {
        while (!predicate.test(input))
            Thread.sleep(250);
    }
}
```

We use `Predicate` (another new `java.util` interface) as our functional interface and test it, pausing for a short while if the condition is not satisfied. We can call this method with a simple lambda that checks if some HTTP server is running.

```
void lambda() throws InterruptedException {
    waitFor(new HttpServer(), (server) -> !server.isRunning());
}
```

The server parameter is supplied by our `waitFor` method and will be the instance of `HttpServer` we've just defined. It's a lambda as the compiler doesn't need to capture the server variable as we supply it manually at runtime.

Note

Incidentally, we might have been able to use a method reference... `waitFor(new HttpServer(), HttpServer::isRunning);`

Now, if we re-implement this as a closure, it would look like this. Firstly, we have to add another `waitFor` method.

```
static void waitFor(Condition condition) throws InterruptedException {
    while (!condition.isSatisfied())
        Thread.sleep(250);
}
```

This time, with a simpler signature. We pass in a functional interface that requires no parameters. The `Condition` interface has a simple `isSatisfied` method with no argument which implies that we have to supply any values an implementation might need. It's already hinting that usages of it may result in closures.

Using it, we'd write something like this:

```
void closure() throws InterruptedException {
    Server server = new HttpServer();
    waitFor(() -> !server.isRunning());
}
```

The server instance is not passed as a parameter to the lambda here but accessed from the enclosing scope. We've defined the variable and the lambda uses it directly. This variable has to be captured, or copied by the compiler. The lambda "closes over" the server variable.

This expression to "close over" comes from the idea that a lambda expression with open bindings (or free variables) have been closed by (or bound in) the lexical environment or scope. The result is a *closed expression*. There are no unbound variables. To be more precise, closures close over *values* not *variables*.

We've seen a closure being used to provide an anonymous block of functionality and the difference between an equivalent lambda but, there are still more useful distinctions we can make.

Other differences

An anonymous function, is a function literal without a name, whilst a closure is an instance of a function. By definition, a lambda has no instance variables; it's not an instance. It's variables are supplied as arguments. A closure however, has instance's variables which are provided when the instance is created.

With this in mind, a lambda will generally be more efficient than a closure as it only needs to evaluated once. Once you have the function, you can re-use it. As a closure closes over something not in it's local environment, it has to be evaluated every time it's called. An instance has to be newed up each time it's used.

All the issues we looked at in the functions vs classes section are relevant here too. There may be memory considerations to using closures over lambdas.

Summary

We've talked about a lot here so let's summarize the differences briefly.

Lambdas are just anonymous functions, similar to static methods in Java. Just like static methods, they can't reference variables outside their scope except for their arguments. A special type of lambda, called a closure, can capture variables outside their scope (or close over them) so they can use external variables or their arguments. So the simple rule is if a lambda uses a variable from outside it's scope, it's also a closure.

Closures can be seen as instances of functions. Which is kind of an odd concept for Java developers.

A great example is the conventional anonymous class that we would pass around if we didn't have the new lambda syntax. These can "close over" variables and so are themselves closures. So we've had closure support in Java since 1.1.

Take a look at this example. The server variable has to be closed over by the compiler to be used in the anonymous instance of the `Condition` interface. This is both an anonymous class instance and a closure.

```
@since Java 1.1!
void anonymousClassClosure() {
    Server server = new HttpServer();
    waitFor(new Condition() {
        @Override
        public Boolean isSatisfied() {
            return !server.isRunning();
        }
    });
}
```

Lambda's aren't always closures, but closures are always lambdas.

In this section we'll explore how the compiler output differs when you compile anonymous classes to when you compile lambdas. First we'll remind ourselves about java bytecode and how to read it. Then we'll look at both anonymous classes and lambdas when they capture variables and when they don't. We'll compare pre-Java 8 closures with lambdas and explore how lambdas are *not* just syntactic sugar but produce very different bytecode from the traditional approaches.

Bytecode recap

To start with, let's recap on what we know about bytecode.

To get from source code to machine runnable code. The Java compiler produces bytecode. This is either interpreted by the JVM or re-compiled by the Just-in-time compiler.

When it's interpreted, the bytecode is turned into machine code on the fly and executed. This happens each time the bytecode is encountered but he JVM.

When it's Just-in-time compiled, the JVM compiles it directly into machine code the first time it's encountered and then goes on to execute it.

Both happen at run-time but Just-in-time compilation offer lots of optimizations.

So, Java bytecode is the intermediate representation between source code and machine code.

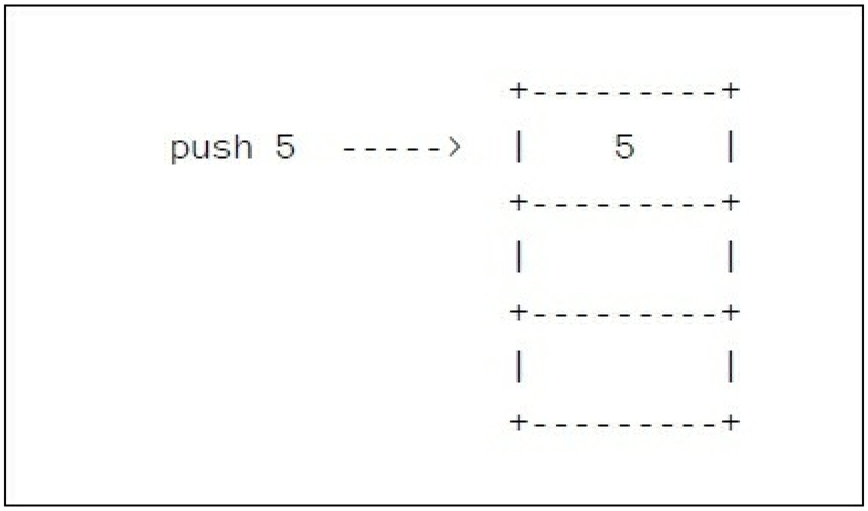
Note

As a quick side bar: Java's JIT compiler has enjoyed a great reputation over the years. But going back full circle to our introduction, it was John McCarthy that first wrote about JIT compilation way back in 1960. So it's interesting to think that it's not just lambda support that was influenced by LISP. (Aycock 2003, 2. JIT Compilation Techniques, 2.1 Genesis, p. 98).

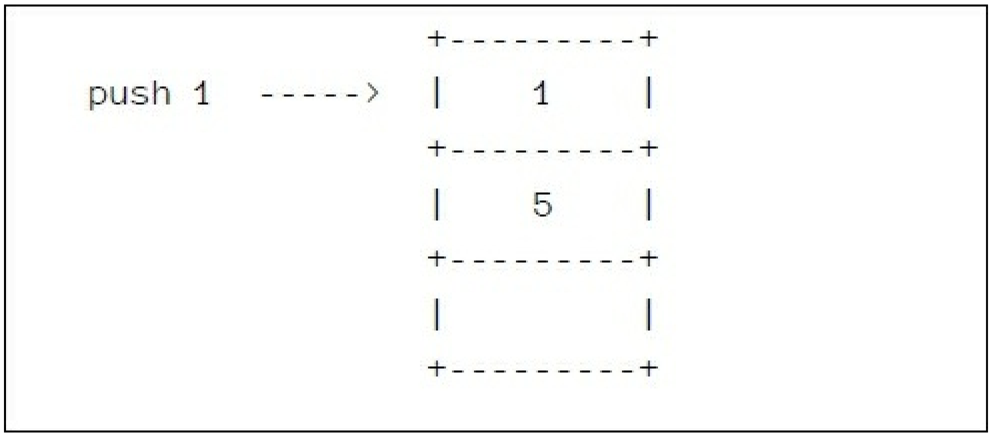
The bytecode is the instruction set of the JVM. As it's name suggests, bytecode consists of single-byte instructions (called *opcodes*) along with associated bytes for parameters. There are therefore a possible 256 opcodes available although only about 200 are actually used.

The JVM uses a stack based computation model, if we want to increment a number, we have to do it using the stack. All instructions or opcodes work against the stack.

So for example, 5 + 1 becomes 5 1 + where 5 is pushed to the stack,

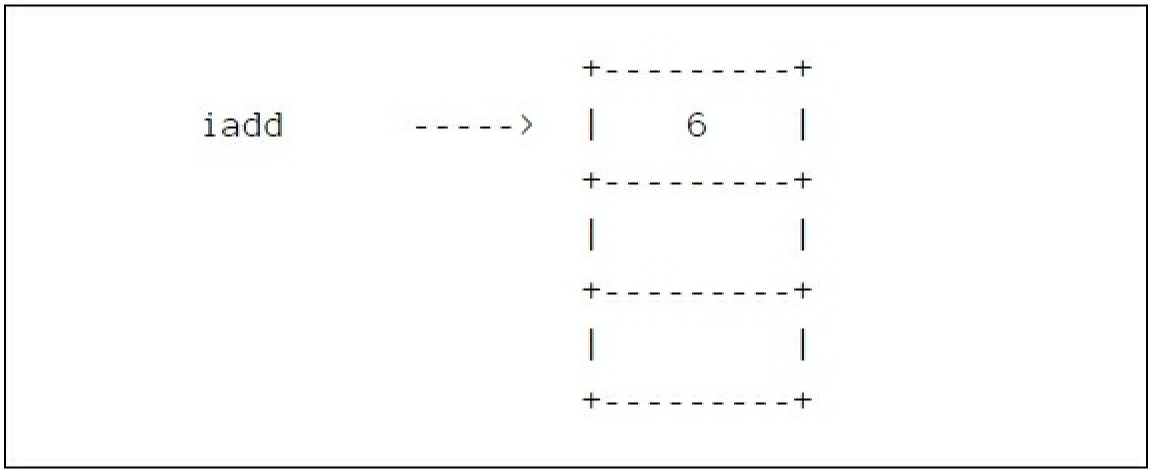


1 is pushed then...



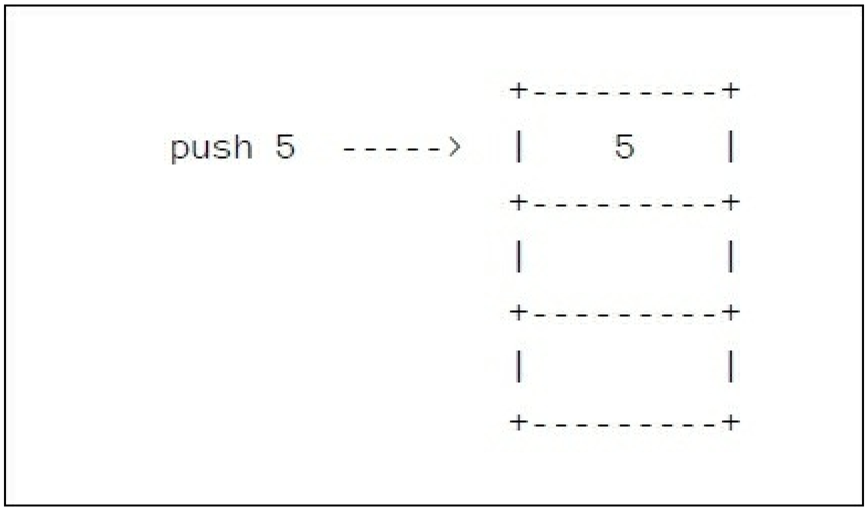
```
                 +---------+
    push 1  ---->|    1    |
                 +---------+
                 |    5    |
                 +---------+
                 |         |
                 +---------+
```

the + operator is applied. Plus would pop the top two frames, add the numbers together and push the result back onto the stack. The result would look like this.

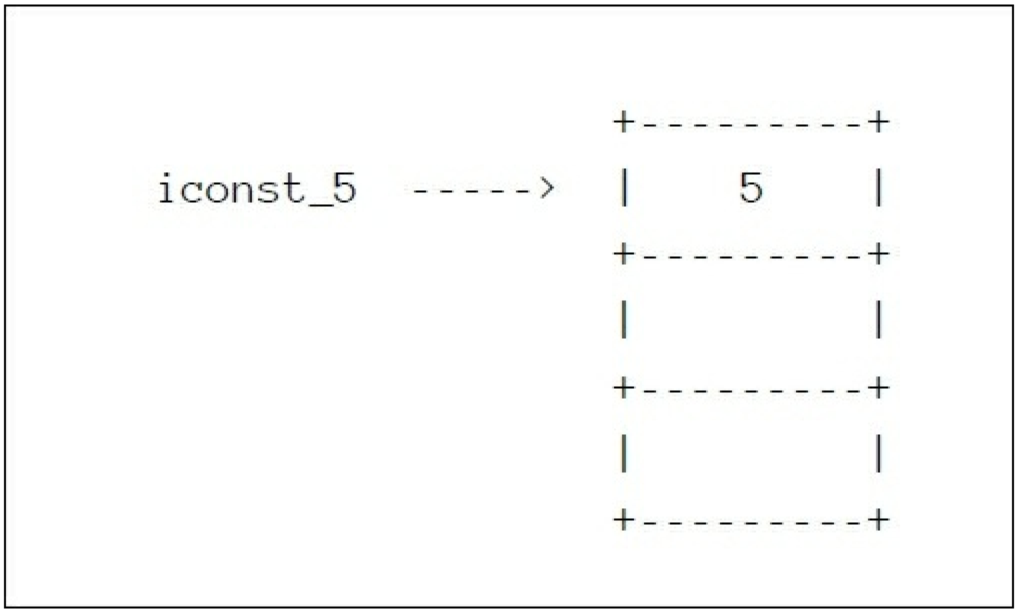


```
                 +---------+
    iadd    ---->|    6    |
                 +---------+
                 |         |
                 +---------+
                 |         |
                 +---------+
```

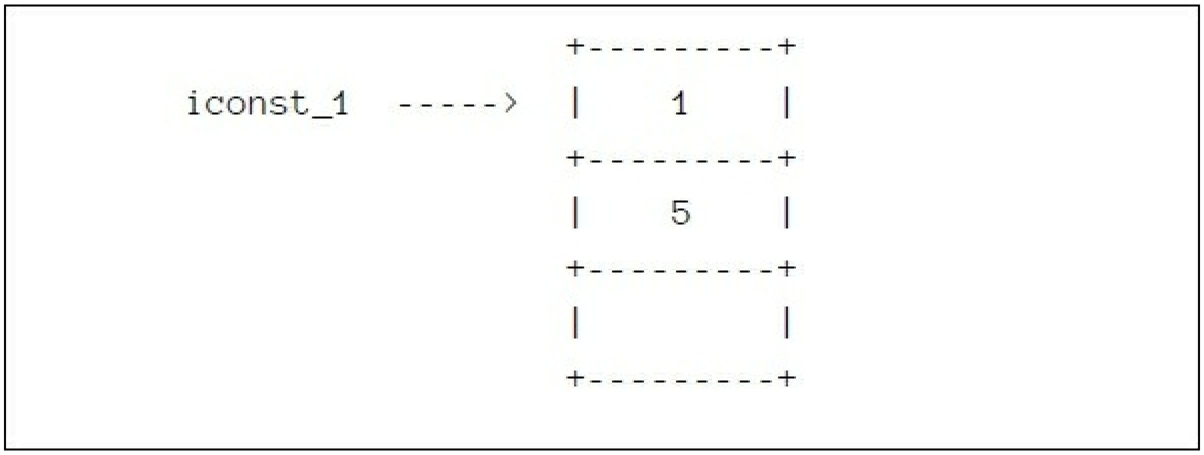
Each opcode works against the stack like this so we can translate our example into a sequence of Java bytecodes:



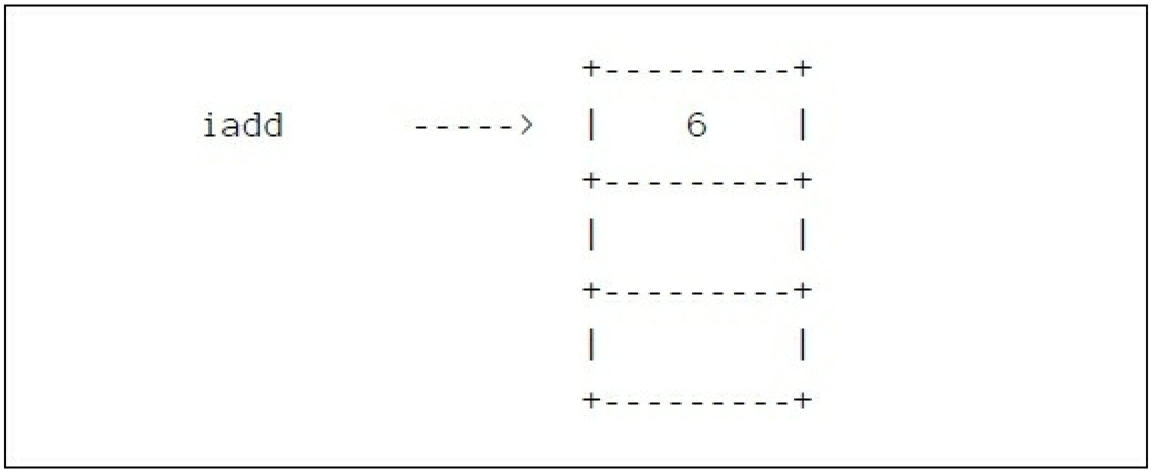
The `push 5` opcode becomes `iconst_5`.



The `push 1` opcode becomes `iconst_1`:



and `add` becomes `iadd`.



The `iconst_x` opcode and `iadd` opcode are examples of opcodes. Opcodes often have prefixes and/or suffices to indicate the types they work on, `i` in these examples refers to integer, `x` is an opcode specific suffix.

We can group the opcodes into the following categories:

Group	**Examples**
Stack manipulation	`aload_n`, `istore`, `swap`, `dup2`
Control flow instructions	`goto`, `ifeq`, `iflt`
Object interactions	`new`, `invokespecial`, `areturn`
Arithmetic, logic and type conversion	`iadd`, `fcmpl`, `i2b`

Instructions concerned with stack manipulation, like `aload` and `istore`.

To control program flow with things like if and while, we use opcodes like `goto` and if equal.

Creating objects and accessing methods use codes like `new` and `invokespecial`. We'll be particularly interested in this group when we look at the different opcodes used to invoke lambdas.

The last group is about arithmetic, logic and type conversion and includes codes like `iadd`, float compare long (`fcmpl`) and integer to byte (`i2b`).

Descriptors

Opcodes will often use parameters, these look a little cryptic in the bytecode as they're usually referenced via lookup tables. Internally, Java uses what's called *descriptors* to describe these parameters.

They describe types and signatures using a specific grammar you'll see throughout the bytecode. You'll often see the same grammar used in compiler or debug output, so it's useful to recap it here.

Here's an example of a method signature descriptor.

```
Example$1."<init>":(Lcom/foo/Example;Lcom/foo/Server;)V
```

It's describing the constructor of a class called `$1`, which we happen to know is the JVM's name for the first anonymous class instance within another class. In this case `Example`. So we've got a constructor of an anonymous class that takes two parameters, an instance of the outer class `com.foo.Example` and an instance of `com.foo.Server`.

Being a constructor, the method doesn't return anything. The `V` symbol represents void.

Have a look at breakdown of the descriptor syntax below. If you see an uppercase `Z` in a descriptor, it's referring to a boolean, an uppercase `B` a byte and so on.

Descriptor Syntax	Java Type
`Z`	`boolean`
`B`	`byte`
`C`	`char`
`S`	`short`
`I`	`int`
`J`	`long`
`F`	`float`
`D`	`double`
`L class;`	`Fully-qualified class`
`[ type`	`[type]`
`(args) type`	`Method type`
`V`	`void (no return type)`

A couple of ones to mention:

- Classes are described with an uppercase `L` followed by the fully qualified class name, followed by a semi-colon. The class name is separated with slashes rather than the dots.
- And arrays are described using an opening square bracket followed by a type from the list. No closing bracket.

Converting a method signature

Let's take the following method signature and turn it into a method descriptor:

```
long f (int n, String s, int[] array);
```

The method returns a `long`, so we describe the fact that it is a method with brackets and that it returns a `long` with a uppercase `J`.

```
()J
```

The first argument is of type `int`, so we use an uppercase `I`.

```
(I)J
```

The next argument is an object, so we use `L` to describe it's an object, fully qualify the name and close it with a semi-colon.

```
(ILString;)J
```

The last argument is an integer array so we drop in the array syntax followed by `int` type:

```
(ILString;[I)J
```

and we're done. A JVM method descriptor.

Code examples

Lets have a look at the bytecode produced for some examples.

We're going to look at the bytecode for four distinct blocks of functionality based on the example we looked at in `lambdas vs closures` section.

We'll explore:

1. A simple anonymous class.
2. An anonymous class closing over some variable (an old style closure).
3. A lambda with no arguments.
4. A lambda with arguments.
5. A lambda closing over some variable (a new style closure).

The example bytecode was generated using the `javap` command line tool. Only partial bytecode listings are shown in this section, for full source and bytecode listings, see `Appendix A`. Also, fully qualified class names have been shortened to better fit on the page.

Example 1

The first example is a simple anonymous class instance passed into our `waitFor` method.

```
public class Example1 {
    // anonymous class
    void example() throws InterruptedException {
        waitFor(new Condition() {
            @Override
            public Boolean isSatisfied() {
                return true;
            }
        });
    }
}
```

If we look at the bytecode below, the thing to notice is that an instance of the anonymous class is newed up at line 6. The `#2` refers to a lookup, the result of which is shown in the comment. So it uses the `new` opcode with whatever is at `#2` in the constant pool, this happens to be the anonymous class `Example$1`.

```
void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=3, locals=1, args_size=1
         0: new #2 // class Example1$1
         3: dup
         4: aload_0
         5: invokespecial #3 // Method Example1$1."":(LExample1;)V
         8: invokestatic #4 // Method WaitFor.waitFor:
            (LCondition;)V
        11: return
      LineNumberTable:
        line 10: 0
        line 16: 11
      LocalVariableTable:
        Start Length Slot Name Signature
            0     12    0 this LExample1;
    Exceptions:
      throws java.lang.InterruptedException
```

Once created, the constructor is called using `invokespecial` on line 9. This opcode is used to call constructor methods, private methods and accessible methods of a super class. You might notice the method descriptor includes a reference to `Example1`. All anonymous class instances have this implicit reference to the parent class.

The next step uses `invokestatic` to call our `waitFor` method passing in the anonymous class on line 10. The `invokestatic` opcode is used to call static methods and is very fast as it can direct dial a method rather than figure out which to call as would be the case in an object hierarchy.

Example 2

The `Example 2` class is another anonymous class but this time it closes over the server variable. It's an old style closure:

```
public class Example2 {
    // anonymous class (closure)
    void example() throws InterruptedException {
        Server server = new HttpServer();
        waitFor(new Condition() {
            @Override
            public Boolean isSatisfied() {
                return !server.isRunning();
            }
        });
    }
}
```

The bytecode is similar to the previous except that an instance of the `Server` class is newed up (at line 3.) and it's constructor called at line 5. The instance of the anonymous class `$1` is still constructed with `invokespecial` (at line 11.) but this time it takes the instance of `Server` as an argument as well as the instance of the calling class.

To close over the server variable, it's passed directly into the anonymous class:

```
void example() throws java.lang.InterruptedException;
    Code:
       0: new #2 // class Server$HttpServer
       3: dup
       4: invokespecial #3 // Method Server$HttpServer."":()V
       7: astore_1
       8: new #4 // class Example2$1
      11: dup
      12: aload_0
      13: aload_1
      14: invokespecial #5 // Method Example2$1."":
          (LExample2;LServer;)V
      17: invokestatic #6 // Method WaitFor.waitFor:(LCondition;)V
      20: return
```

Example 3

The `Example 3` class uses a Java lambda with our `waitFor` method. The lambda doesn't do anything other than return true. It's equivalent to example 1.

```
public class Example3 {
    // simple lambda
    void example() throws InterruptedException {
        waitFor(() -> true);
    }
}
```

The bytecode is super simple this time. It uses the `invokedynamic` opcode to create the lambda at line 3 which is then passed to the `invokestatic` opcode on the next line.

```
 void example() throws java.lang.InterruptedException;
     Code:
        0: invokedynamic #2, 0 // InvokeDynamic #0:isSatisfied:
           ()LCondition;
        5: invokestatic #3 // Method WaitFor.waitFor:(LCondition;)V
        8: return
```

The descriptor for the `invokedynamic` call is targeting the `isSatisfied` method on the `Condition` interface (line 3.).

What we're not seeing here is the mechanics of `invokedynamic`. The `invokedynamic` opcode is a new opcode to Java 7, it was intended to provide better support for dynamic languages on the JVM. It does this by not linking the types to methods until run-time. The other "invoke" opcodes all resolve types at compile time.

For lambdas, this means that placeholder method invocations can be put into the bytecode like we've just seen and working out the implementation can be done on the JVM at runtime.

If we look at a more verbose bytecode that includes the constant pool we can dereference the lookups. For example, if we look up number 2, we can see it references `#0` and `#26`.

```
Constant pool:
   #1 = Methodref #6.#21 // Object."":()V
   #2 = InvokeDynamic #0:#26 // #0:isSatisfied:()LCondition;
   ...
BootstrapMethods:
    0: #23 invokestatic LambdaMetafactory.metafactory:
           (LMethodHandles$Lookup;LString;
           LMethodType;LMethodType;
           LMethodHandle;LMethodType;)LCallSite;
      Method arguments:
        #24 ()LBoolean;
        #25 invokestatic Example3.lambda$example$25:()LBoolean;
        #24 ()LBoolean;
```

The constant `0` is in a special lookup table for bootstrapping methods (line 6). It refers to a static method call to the JDK `LambdaMetafactory` to create the lambda. This is where the heavy lifting goes on. All the target type inference to adapt types and any partial argument evaluation goes on here.

The actual lambda is shown as a method handle called `lambda$example$25` (line 12) with no arguments, returning a boolean. It's invoked using `invokestatic` which shows that it's accessed as a genuine function; there's no object associated with it. There's also no implicit reference to a containing class unlike the anonymous examples before.

It's passed into the `LambdaMetafactory` and we know it's a method handle by looking it up in the constant pool. The number of the lambda is compiler assigned and just increments from zero for each lambda required.

```
Constant pool:
    // invokestatic Example3.lambda$example$25:()LBoolean;
    #25 = MethodHandle #6:#35
```

Example 4

The `Example 4` class is another lambda but this time it takes an instance of `Server` as an argument. It's equivalent in functionality to example 2 but it doesn't close over the variable; it's not a closure.

```
public class Example4 {
    // lambda with arguments
    void example() throws InterruptedException {
        waitFor(new HttpServer(), (server) -> server.isRunning());
    }
}
```

Just like example 2, the bytecode has to create the instance of server but this time, the `invokedynamic` opcode references the `test` method of type `Predicate`. If we were to follow the reference (`#4`) to the boostrap methods table, we would see the actual lambda requires an argument of type `HttpServer` and returns a `Z` which is a primitive boolean.

```
void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: new #2 // class Server$HttpServer
         3: dup
         4: invokespecial #3 // Method Server$HttpServer."":()V
         7: invokedynamic #4, 0 // InvokeDynamic #0:test:
            ()LPredicate;
        12: invokestatic #5 // Method WaitFor.waitFor:
            (LObject;LPredicate;)V
        15: return
      LineNumberTable:
        line 13: 0
        line 15: 15
      LocalVariableTable:
        Start Length Slot Name Signature
            0     16    0 this LExample4;
    Exceptions:
      throws java.lang.InterruptedException
```

So the call to the lambda is still a static method call like before but this time takes the variable as a parameter when it's invoked.

Example 4 (with method reference)

Interestingly, if we use a method reference instead, the functionality is exactly the same but we get different bytecode.

```
public class Example4_method_reference {
    // lambda with method reference
    void example() throws InterruptedException {
        waitFor(new HttpServer(), HttpServer::isRunning);
    }
}
```

Via the call to `LambdaMetafactory` when the final execution occurs, `method_reference` results in a call to `invokevirtual` rather than `invokestatic`. The `invokevirtual` opcode is used to call public, protected an package protected methods so it implies an instance is required. The instance is supplied to the `metafactory` method and no lambda (or static function) is needed at all; there are no `lambda$` in this bytecode.

```
void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: new #2 // class Server$HttpServer
         3: dup
         4: invokespecial #3 // Method Server$HttpServer."":()V
         7: invokedynamic #4, 0 // InvokeDynamic #0:test:
            ()LPredicate;
        12: invokestatic #5 // Method WaitFor.waitFor:
            (LObject;LPredicate;)V
        15: return
      LineNumberTable:
        line 11: 0
        line 12: 15
      LocalVariableTable:
        Start Length Slot Name Signature
            0     16    0 this LExample4_method_reference;
    Exceptions:
      throws java.lang.InterruptedException
```

Example 5

Lastly, example 5 uses a lambda but closes over the server instance. It's equivalent to example 2 and is a new style closure.

```
public class Example5 {
    // closure
    void example() throws InterruptedException {
        Server server = new HttpServer();
        waitFor(() -> !server.isRunning());
    }
}
```

It goes through the basics in the same way as the other lambdas but if we lookup the `metafactory` method in the bootstrap methods table, you'll notice that this time, the lambda's method handle has an argument of type `Server`. It's invoked using `invokestatic` (line 9) and the variable is passed directly into the lambda at invocation time.

```
BootstrapMethods:
    0: #34 invokestatic LambdaMetafactory.metafactory:
            (LMethodHandles$Lookup;
             LString;LMethodType;
             LMethodType;
             LMethodHandle;LMethodType;)LCallSite;
      Method arguments:
        #35 ()LBoolean; // <-- SAM method to be implemented by the
            lambda
        #36 invokestatic Example5.lambda$example$35:
          (LServer;)LBoolean;
        #35 ()LBoolean; // <-- type to be enforced at invocation
        time
```

So like the anonymous class in example 2, an argument is added by the compiler to capture the term although this time, it's a method argument rather than a constructor argument.

Summary

We saw how using an anonymous class will create a new instance and call it's constructor with `invokespecial`.

We saw anonymous classes that close over variables have an extra argument on their constructor to capture that variable.

And we saw how Java lambdas use the `invokedynamic` instruction to defer binding of the types and that a special `lambda$` method handle is used to actually represent the lambda. This method handle has no arguments in this case and is invoked using `invokestatic` making it a genuine function.

The lambda was created by the `LambdaMetafactory` class which itself was the target of the `invokedynamic` call.

When a lambda has arguments, we saw how the `LambdaMetafactory` describes the argument to be passed into the lambda. The `invokestatic` opcode is used to execute the lambda like before. But we also had a look at a method reference used in-lieu of a lambda. In this case, no `lambda$` method handle was created and `invokevirtual` was used to call the method directly.

Lastly, we looked at a lambda that closes over a variable. This one creates an argument on the `lambda$` method handle and again is called with `invokestatic`.

Appendix A. Bytecode

WaitFor

```
package jdk8.byte_code;

class WaitFor {
    static void waitFor(Condition condition) throws
    InterruptedException {
        while (!condition.isSatisfied())
            Thread.sleep(250);
    }

    static <T> void waitFor(T input, Predicate<T> predicate)
            throws InterruptedException {
        while (!predicate.test(input))
            Thread.sleep(250);
    }
}
```

Example 1

```
package jdk8.byte_code;

import static jdk8.byte_code.WaitFor.waitFor;

@SuppressWarnings("all")
public class Example1 {

    // anonymous class
    void example() throws InterruptedException {
        waitFor(new Condition() {
            @Override
            public Boolean isSatisfied() {
                return true;
            }
        });
    }
}
```

```
Classfile Example1.class
 Last modified 08-May-2014; size 603 bytes
 MD5 checksum 7365ca98fe204fc9198043cef5d241be
 Compiled from "Example1.java"
public class jdk8.byte_code.Example1
  SourceFile: "Example1.java"
  InnerClasses:
         #2; //class jdk8/byte_code/Example1$1
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref #6.#20 // java/lang/Object." <init>":()V
  #2 = Class #21 // jdk8/byte_code/Example1$1
  #3 = Methodref #2.#22 // jdk8/byte_code/Example1$1." <init>":
(Ljdk8/byte_code/Example1;)V
  #4 = Methodref #23.#24 //
jdk8/byte_code/WaitFor.waitFor:(Ljdk8/byte_code/Condition;)V
 #5 = Class #25 // jdk8/byte_code/Example1
 #6 = Class #26 // java/lang/Object
 #7 = Utf8 InnerClasses
 #8 = Utf8 <init>
 #9 = Utf8 ()V
 #10 = Utf8 Code
 #11 = Utf8 LineNumberTable
 #12 = Utf8 LocalVariableTable
 #13 = Utf8 this
 #14 = Utf8 Ljdk8/byte_code/Example1;
 #15 = Utf8 example
 #16 = Utf8 Exceptions
 #17 = Class #27 // java/lang/InterruptedException
 #18 = Utf8 SourceFile
 #19 = Utf8 Example1.java
 #20 = NameAndType #8:#9 // "":()V
 #21 = Utf8 jdk8/byte_code/Example1$1
 #22 = NameAndType #8:#28 // "":(Ljdk8/byte_code/Example1;)V
 #23 = Class #29 // jdk8/byte_code/WaitFor
 #24 = NameAndType #30:#31 // waitFor:(Ljdk8/byte_code/Condition;)V
 #25 = Utf8 jdk8/byte_code/Example1
 #26 = Utf8 java/lang/Object
 #27 = Utf8 java/lang/InterruptedException
 #28 = Utf8 (Ljdk8/byte_code/Example1;)V
 #29 = Utf8 jdk8/byte_code/WaitFor
 #30 = Utf8 waitFor
 #31 = Utf8 (Ljdk8/byte_code/Condition;)V
{
 public jdk8.byte_code.Example1();
   descriptor: ()V
   flags: ACC_PUBLIC
   Code:
     stack=1, locals=1, args_size=1
        0: aload_0
        1: invokespecial #1 // Method java/lang/Object."<init>":()V
        4: return
     LineNumberTable:
       line 6: 0
     LocalVariableTable:
       Start Length Slot Name Signature
           0     5     0   this Ljdk8/byte_code/Example1;
 void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=3, locals=1, args_size=1
         0: new           #2         // class
         jdk8/byte_code/Example1$1
         3: dup
         4: aload_0
         5: invokespecial #3        // Method
 jdk8/byte_code/Example1$1."<init>":(Ljdk8/byte_code/Example1;)V
         8: invokestatic  #4                  // Method
jdk8/byte_code/WaitFor.waitFor:(Ljdk8/byte_code/Condition;)V
        11: return
      LineNumberTable:
        line 10: 0
        line 16: 11
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      12     0  this   Ljdk8/byte_code/Example1;
    Exceptions:
      throws java.lang.InterruptedException
}
```

Example 2

```
package jdk8.byte_code;

public interface Server {

  Boolean isRunning();

  public class HttpServer implements Server {
    @Override
    public Boolean isRunning() {
      return false;
    }
  }
}
```

```
package jdk8.byte_code;

import static jdk8.byte_code.Server.*;
import static jdk8.byte_code.WaitFor.waitFor;

public class Example2 {

    // anonymous class (closure)
    void example() throws InterruptedException {
        Server server = new HttpServer();
        waitFor(new Condition() {
            @Override
            public Boolean isSatisfied() {
                return !server.isRunning();
            }
        });
    }
}
```

```
Classfile Example2.class
 Last modified 08-May-2014; size 775 bytes
 MD5 checksum 2becf3c32e2b08abc50465aca7398c4b
 Compiled from "Example2.java"
 public class jdk8.byte_code.Example2
 SourceFile: "Example2.java"
 InnerClasses:
     #4; //class jdk8/byte_code/Example2$1
     public static #27= #2 of #25; //HttpServer=class
jdk8/byte_code/Server$HttpServer of class jdk8/byte_code/Server
 minor version: 0
 major version: 52
 flags: ACC_PUBLIC, ACC_SUPER
 Constant pool:
 #1 = Methodref     #8.#24     // java/lang/Object."<init>":()V
 #2 = Class         #26        // jdk8/byte_code/Server$HttpServer
 #3 = Methodref     #2.#24     // jdk8/byte_code/Server$HttpServer."<init>":()V
 #4 = Class              #28          // jdk8/byte_code/Example2$1
 #5 = Methodref          #4.#29       // jdk8/byte_code/Example2$1."<init>":
(Ljdk8/byte_code/Example2;Ljdk8/byte_code/Server;)V
 #6 = Methodref          #30.#31      // jdk8/byte_code/WaitFor.waitFor:
(Ljdk8/byte_code/Condition;)V
 #7 = Class              #32          // jdk8/byte_code/Example2
 #8 = Class              #33          // java/lang/Object
 #9 = Utf8               InnerClasses
 #10 = Utf8              <init>
 #11 = Utf8              ()V
 #12 = Utf8              Code
 #13 = Utf8              LineNumberTable
 #14 = Utf8              LocalVariableTable
 #15 = Utf8              this
 #16 = Utf8              Ljdk8/byte_code/Example2;
 #17 = Utf8              example
 #18 = Utf8              server
 #19 = Utf8              Ljdk8/byte_code/Server;
 #20 = Utf8              Exceptions
 #21 = Class             #34            // java/lang/InterruptedException
 #22 = Utf8              SourceFile
 #23 = Utf8              Example2.java
 #24 = NameAndType       #10:#11        // "<init>":()V
 #25 = Class             #35            // jdk8/byte_code/Server
 #26 = Utf8              jdk8/byte_code/Server$HttpServer
 #27 = Utf8              HttpServer
 #28 = Utf8              jdk8/byte_code/Example2$1
 #29 = NameAndType       #10:#36        // "<init>":
(Ljdk8/byte_code/Example2;Ljdk8/byte_code/Server;)V
 #30 = Class             #37            // jdk8/byte_code/WaitFor
 #31 = NameAndType       #38:#39        // waitFor:(Ljdk8/byte_code/Condition;)V
 #32 = Utf8              jdk8/byte_code/Example2
 #33 = Utf8              java/lang/Object
 #34 = Utf8              java/lang/InterruptedException
 #35 = Utf8              jdk8/byte_code/Server
 #36 = Utf8              (Ljdk8/byte_code/Example2;Ljdk8/byte_code/Server;)V
 #37 = Utf8              jdk8/byte_code/WaitFor
 #38 = Utf8              waitFor
 #39 = Utf8              (Ljdk8/byte_code/Condition;)V
{
 public jdk8.byte_code.Example2();
 descriptor: ()V
 flags: ACC_PUBLIC
 Code:
   stack=1, locals=1, args_size=1
     0: aload_0
     1: invokespecial #1         // Method java/lang/Object."<init>":()V
     4: return
     LineNumberTable:
       line 6: 0
     LocalVariableTable:
     Start Length Slot Name Signature
         0      5    0 this Ljdk8/byte_code/Example2;
 void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=4, locals=2, args_size=1
         0: new           #2   //class jdk8/byte_code/Server$HttpServer
         3: dup
         4: invokespecial #3   //Method jdk8/byte_code/Server$HttpServer."
<init>":()V
         7: astore_1
         8: new           #4   // class jdk8/byte_code/Example2$1
        11: dup
        12: aload_0
        13: aload_1
        14: invokespecial #5   // Method jdk8/byte_code/Example2$1."<init>":
(Ljdk8/byte_code/Example2;Ljdk8/byte_code/Server;)V
        17: invokestatic  #6   // Method jdk8/byte_code/WaitFor.waitFor:
(Ljdk8/byte_code/Condition;)V
        20: return
      LineNumberTable:
        line 10: 0
        line 11: 8
        line 17: 20
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      21     0  this   Ljdk8/byte_code/Example2;
            8      13     1 server   Ljdk8/byte_code/Server;
    Exceptions:
      throws java.lang.InterruptedException
}
```

Example 3

```
package jdk8.byte_code;

import static jdk8.byte_code.WaitFor.waitFor;

public class Example3 {
    // simple lambda
    void example() throws InterruptedException {
        waitFor(() -> true);
    }
}
```

```
Classfile Example3.class
  Last modified 08-May-2014; size 1155 bytes
  MD5 checksum 22e120de85528efc921bb158588bbaa1
  Compiled from "Example3.java"
public class jdk8.byte_code.Example3
  SourceFile: "Example3.java"
  InnerClasses:
      public static final #50= #49 of #53; //Lookup=class
java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #23 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
  Method arguments:
    #24 ()Ljava/lang/Boolean;
    #25 invokestatic jdk8/byte_code/Example3.lambda$example$25:
()Ljava/lang/Boolean;
    #24 ()Ljava/lang/Boolean;
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
 #1 = Methodref         #6.#21     // java/lang/Object."<init>":()V
 #2 = InvokeDynamic     #0:#26     // #0:isSatisfied:()Ljdk8/byte_code/Condition;
 #3 = Methodref         #27.#28    // jdk8/byte_code/WaitFor.waitFor:
(Ljdk8/byte_code/Condition;)V
 #4 = Methodref         #29.#30    // java/lang/Boolean.valueOf:
(Z)Ljava/lang/Boolean;
 #5 = Class             #31        // jdk8/byte_code/Example3
 #6 = Class             #32        // java/lang/Object
 #7 = Utf8              <init>
 #8 = Utf8              ()V
 #9 = Utf8              Code
 #10 = Utf8             LineNumberTable
 #11 = Utf8             LocalVariableTable
 #12 = Utf8             this
 #13 = Utf8             Ljdk8/byte_code/Example3;
 #14 = Utf8             example
 #15 = Utf8             Exceptions
 #16 = Class              #33       // java/lang/InterruptedException
 #17 = Utf8               lambda$example$25
 #18 = Utf8               ()Ljava/lang/Boolean;
 #19 = Utf8               SourceFile
 #20 = Utf8               Example3.java
 #21 = NameAndType        #7:#8     // "<init>":()V
 #22 = Utf8               BootstrapMethods
 #23 = MethodHandle       #6:#34    // invokestatic
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #24 = MethodType         #18       // ()Ljava/lang/Boolean;
 #25 = MethodHandle       #6:#35    // invokestatic
jdk8/byte_code/Example3.lambda$example$25:()Ljava/lang/Boolean;
 #26 = NameAndType        #36:#37   // isSatisfied:()Ljdk8/byte_code/Condition;
 #27 = Class              #38       // jdk8/byte_code/WaitFor
 #28 = NameAndType        #39:#40   // waitFor:(Ljdk8/byte_code/Condition;)V
 #29 = Class              #41       // java/lang/Boolean
 #30 = NameAndType        #42:#43   // valueOf:(Z)Ljava/lang/Boolean;
 #31 = Utf8               jdk8/byte_code/Example3
 #32 = Utf8               java/lang/Object
 #33 = Utf8               java/lang/InterruptedException
 #34 = Methodref          #44.#45   //
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #35 = Methodref          #5.#46    // jdk8/byte_code/Example3.lambda$example$25:
()Ljava/lang/Boolean;
 #36 = Utf8               isSatisfied
 #37 = Utf8               ()Ljdk8/byte_code/Condition;
 #38 = Utf8               jdk8/byte_code/WaitFor
 #39 = Utf8               waitFor
 #40 = Utf8               (Ljdk8/byte_code/Condition;)V
 #41 = Utf8               java/lang/Boolean
 #42 = Utf8               valueOf
 #43 = Utf8               (Z)Ljava/lang/Boolean;
 #44 = Class              #47       // java/lang/invoke/LambdaMetafactory
 #45 = NameAndType        #48:#52   // metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #46 = NameAndType        #17:#18  // lambda$example$25:()Ljava/lang/Boolean;
 #47 = Utf8               java/lang/invoke/LambdaMetafactory
 #48 = Utf8               metafactory
 #49 = Class              #54      // java/lang/invoke/MethodHandles$Lookup
 #50 = Utf8               Lookup
 #51 = Utf8               InnerClasses
 #52 = Utf8
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #53 = Class              #55        // java/lang/invoke/MethodHandles
 #54 = Utf8               java/lang/invoke/MethodHandles$Lookup
 #55 = Utf8             java/lang/invoke/MethodHandles
{
public jdk8.byte_code.Example3();
 descriptor: ()V
 flags: ACC_PUBLIC
 Code:
  stack=1, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1           // Method java/lang/Object."<init>":()V
    4: return
 LineNumberTable:
 line 6: 0
 LocalVariableTable:
 Start Length Slot Name Signature
     0      5    0 this Ljdk8/byte_code/Example3;
void example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=1, locals=1, args_size=1
         0: invokedynamic #2,  0    // InvokeDynamic #0:isSatisfied:
()Ljdk8/byte_code/Condition;
         5: invokestatic  #3       // Method jdk8/byte_code/WaitFor.waitFor:
(Ljdk8/byte_code/Condition;)V
         8: return
      LineNumberTable:
        line 10: 0
        line 11: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Ljdk8/byte_code/Example3;
    Exceptions:
      throws java.lang.InterruptedException
}
```

Example 4

```
package jdk8.byte_code;

import static jdk8.byte_code.Server.HttpServer;
import static jdk8.byte_code.WaitFor.waitFor;

public class Example4 {
    // lambda with arguments
    void example() throws InterruptedException {
        waitFor(new HttpServer(), (server) -> server.isRunning());
    }
}
```

```
Classfile Example4.class
   Last modified 08-May-2014; size 1414 bytes
   MD5 checksum 7177f97fdf30b0648a09ab98109a479c
   Compiled from "Example4.java"
 public class jdk8.byte_code.Example4
   SourceFile: "Example4.java"
   InnerClasses:
     public static #21= #2 of #29; //HttpServer=class
jdk8/byte_code/Server$HttpServer of class jdk8/byte_code/Server
     public static final #65= #64 of #67; //Lookup=class
java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 Method arguments:
   #33 (Ljava/lang/Object;)Z
   #34 invokestatic jdk8/byte_code/Example4.lambda$example33 : (L
j
d
k8/b
y
t
e
c
o
d
e/S
e
r
v
e
rHttpServer;)Z
   #35 (Ljdk8/byte_code/Server$HttpServer;)Z
 minor version: 0
 major version: 52
 flags: ACC_PUBLIC, ACC_SUPER
 Constant pool:
```

```
 #1 = Methodref          #9.#28     //java/lang/Object."<init>":()V
 #2 = Class              #30        //jdk8/byte_code/Server$HttpServer
 #3 = Methodref          #2.#28     //jdk8/byte_code/Server$HttpServer."<init>":
()V
 #4 = InvokeDynamic      #0:#36     // #0:test:()Ljava/util/function/Predicate;
 #5 = Methodref          #37.#38    // jdk8/byte_code/WaitFor.waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
 #6 = Methodref          #2.#39    // jdk8/byte_code/Server$HttpServer.isRunning:
()Ljava/lang/Boolean;
 #7 = Methodref          #40.#41   // java/lang/Boolean.booleanValue:()Z
 #8 = Class              #42       // jdk8/byte_code/Example4
 #9 = Class              #43       //java/lang/Object
 #10 = Utf8              <init>
 #11 = Utf8              ()V
 #12 = Utf8              Code
 #13 = Utf8              LineNumberTable
 #14 = Utf8              LocalVariableTable
 #15 = Utf8              this
 #16 = Utf8              Ljdk8/byte_code/Example4;
 #17 = Utf8              example
 #18 = Utf8              Exceptions
 #19 = Class             #44       // java/lang/InterruptedException
 #20 = Utf8              lambda$example$33
 #21 = Utf8              HttpServer
 #22 = Utf8              InnerClasses
 #23 = Utf8              (Ljdk8/byte_code/Server$HttpServer;)Z
 #24 = Utf8              server
 #25 = Utf8              Ljdk8/byte_code/Server$HttpServer;
#26 = Utf8               SourceFile
#27 = Utf8               Example4.java
#28 = NameAndType        #10:#11   // "<init>":()V
#29 = Class              #45       // jdk8/byte_code/Server
#30 = Utf8               jdk8/byte_code/Server$HttpServer
#31 = Utf8               BootstrapMethods
#32 = MethodHandle       #6:#46    // invokestatic
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#33 = MethodType         #47       // (Ljava/lang/Object;)Z
#34 = MethodHandle       #6:#48    // invokestatic
jdk8/byte_code/Example4.lambda$example33 : (L
j
d
k8/b
y
t
e
c
o
d
e/S
e
r
```

```
v
e
rHttpServer;)Z
#35 = MethodType         #23       // (Ljdk8/byte_code/Server$HttpServer;)Z
#36 = NameAndType        #49:#50   // test:()Ljava/util/function/Predicate;
#37 = Class              #51       // jdk8/byte_code/WaitFor
#38 = NameAndType        #52:#53   // waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
#39 = NameAndType        #54:#55   // isRunning:()Ljava/lang/Boolean;
#40 = Class              #56       // java/lang/Boolean
#41 = NameAndType        #57:#58   // booleanValue:()Z
#42 = Utf8               jdk8/byte_code/Example4
#43 = Utf8               java/lang/Object
#44 = Utf8               java/lang/InterruptedException
#45 = Utf8               jdk8/byte_code/Server
#46 = Methodref          #59.#60   //
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#47 = Utf8               (Ljava/lang/Object;)Z
#48 = Methodref          #8.#61    // jdk8/byte_code/Example4.lambda$example33 :
(L
j
d
k8/b
y
t
e
c
o
d
e/S
e
r
v
e
rHttpServer;)Z
#49 = Utf8               test
#50 = Utf8               ()Ljava/util/function/Predicate;
#51 = Utf8               jdk8/byte_code/WaitFor
#52 = Utf8               waitFor
#53 = Utf8            (Ljava/lang/Object;Ljava/util/function/Predicate;)V
#54 = Utf8               isRunning
#55 = Utf8               ()Ljava/lang/Boolean;
#56 = Utf8               java/lang/Boolean
#57 = Utf8               booleanValue
#58 = Utf8               ()Z
#59 = Class              #62       // java/lang/invoke/LambdaMetafactory
#60 = NameAndType #63:#66 // metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#61 = NameAndType        #20:#23   // lambda$example33 : (L
j
d
k8/b
y
t
e
c
o
d
e/S
e
r
v
e
rHttpServer;)Z
#62 = Utf8               java/lang/invoke/LambdaMetafactory
#63 = Utf8               metafactory
#64 = Class              #68       // java/lang/invoke/MethodHandles$Lookup
#65 = Utf8               Lookup
#66 = Utf8
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#67 = Class              #69       // java/lang/invoke/MethodHandles
#68 = Utf8               java/lang/invoke/MethodHandles$Lookup
#69 = Utf8               java/lang/invoke/MethodHandles
{
 public jdk8.byte_code.Example4();
 descriptor: ()V
 flags: ACC_PUBLIC
 Code: stack=1, locals=1, args_size=1
 0: aload_0 1:
 invokespecial #1                  // Method java/lang/Object."":()V
 4: return
 LineNumberTable:
 line 9: 0
 LocalVariableTable:
 Start Length Slot Name Signature
     0      5    0 this Ljdk8/byte_code/Example4;
 void example() throws java.lang.InterruptedException;
   descriptor: ()V
   flags:
   Code:
     stack=2, locals=1, args_size=1
         0: new           #2      //class jdk8/byte_code/Server$HttpServer
         3: dup
         4: invokespecial #3      //Method jdk8/byte_code/Server$HttpServer."
<init>":()V
         7: invokedynamic #4,  0  //InvokeDynamic #0:test:
()Ljava/util/function/Predicate;
        12: invokestatic  #5      // Method jdk8/byte_code/WaitFor.waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
        15: return
       LineNumberTable:
        line 13: 0
        line 15: 15
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      16     0  this   Ljdk8/byte_code/Example4;
    Exceptions:
      throws java.lang.InterruptedException
}
```

Example 4 (with Method Reference)

```
package jdk8.byte_code;

import static jdk8.byte_code.Server.HttpServer;
import static jdk8.byte_code.WaitFor.waitFor;

@SuppressWarnings("all")
public class Example4_method_reference {
    // lambda with method reference
    void example() throws InterruptedException {
        waitFor(new HttpServer(), HttpServer::isRunning);
    }

}
```

```
Classfile Example4_method_reference.class
  Last modified 08-May-2014; size 1271 bytes
  MD5 checksum f8aef942361f29ef599adfec7a594948
  Compiled from "Example4_method_reference.java"
public class jdk8.byte_code.Example4_method_reference
 SourceFile: "Example4_method_reference.java"
 InnerClasses:
   public static #23= #2 of #21; //HttpServer=class
jdk8/byte_code/Server$HttpServer of class jdk8/byte_code/Server
   public static final #52= #51 of #56; //Lookup=class
java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #26 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
       #27 (Ljava/lang/Object;)Z
       #28 invokevirtual jdk8/byte_code/Server$HttpServer.isRunning:
()Ljava/lang/Boolean;
       #29 (Ljdk8/byte_code/Server$HttpServer;)Z
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
 #1 = Methodref       #7.#20         //java/lang/Object."<init>":()V
 #2 = Class           #22            //jdk8/byte_code/Server$HttpServer
 #3 = Methodref       #2.#20         //  jdk8/byte_code/Server$HttpServer."
<init>":()V
 #4 = InvokeDynamic   #0:#30         // #0:test:()Ljava/util/function/Predicate;
 #5 = Methodref       #31.#32        // jdk8/byte_code/WaitFor.waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
 #6 = Class           #33            // jdk8/byte_code/Example4_method_reference
 #7 = Class           #34            // java/lang/Object
 #8 = Utf8            <init>
 #9 = Utf8            ()V
 #10 = Utf8           Code
 #11 = Utf8           LineNumberTable
 #12 = Utf8           LocalVariableTable
 #13 = Utf8           this
 #14 = Utf8           Ljdk8/byte_code/Example4_method_reference;
 #15 = Utf8           example
 #16 = Utf8           Exceptions
 #17 = Class          #35            // java/lang/InterruptedException
 #18 = Utf8           SourceFile
 #19 = Utf8           Example4_method_reference.java
 #20 = NameAndType    #8:#9          // "<init>":()V
 #21 = Class          #36            // jdk8/byte_code/Server
 #22 = Utf8           jdk8/byte_code/Server$HttpServer
 #23 = Utf8           HttpServer
 #24 = Utf8           InnerClasses
 #25 = Utf8           BootstrapMethods
 #26 = MethodHandle   #6:#37         // invokestatic
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#27 = MethodType      #38            // (Ljava/lang/Object;)Z
#28 = MethodHandle    #5:#39         // invokevirtual
jdk8/byte_code/Server$HttpServer.isRunning:()Ljava/lang/Boolean;
#29 = MethodType      #40            // (Ljdk8/byte_code/Server$HttpServer;)Z
#30 = NameAndType     #41:#42        // test:()Ljava/util/function/Predicate;
#31 = Class           #43            //jdk8/byte_code/WaitFor
#32 = NameAndType     #44:#45        // waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
#33 = Utf8            jdk8/byte_code/Example4_method_reference
#34 = Utf8            java/lang/Object
#35 = Utf8            java/lang/InterruptedException
#36 = Utf8            jdk8/byte_code/Server
#37 = Methodref       #46.#47        //
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#38 = Utf8            (Ljava/lang/Object;)Z
#39 = Methodref       #2.#48         //
jdk8/byte_code/Server$HttpServer.isRunning:()Ljava/lang/Boolean;
#40 = Utf8            (Ljdk8/byte_code/Server$HttpServer;)Z
#41 = Utf8            test
#42 = Utf8            ()Ljava/util/function/Predicate;
#43 = Utf8            jdk8/byte_code/WaitFor
#44 = Utf8            waitFor
#45 = Utf8           (Ljava/lang/Object;Ljava/util/function/Predicate;)V
#46 = Class           #49            // java/lang/invoke/LambdaMetafactory
#47 = NameAndType     #50:#53        // metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#48 = NameAndType     #54:#55        // isRunning:()Ljava/lang/Boolean; #49 =
Utf8             java/lang/invoke/LambdaMetafactory
#50 = Utf8            metafactory
#51 = Class           #57            // java/lang/invoke/MethodHandles$Lookup
```

```
#52 = Utf8           Lookup
#53 = Utf8
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
#54 = Utf8           isRunning
#55 = Utf8           ()Ljava/lang/Boolean;
#56 = Class          #58 // java/lang/invoke/MethodHandles
#57 = Utf8           java/lang/invoke/MethodHandles$Lookup
#58 = Utf8           java/lang/invoke/MethodHandles
{
 public jdk8.byte_code.Example4_method_reference();
   descriptor: ()V
   flags: ACC_PUBLIC
   Code:
     stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1           // Method java/lang/Object."":()V
      4: return
LineNumberTable:
 line 7: 0
LocalVariableTable:
 Start Length Slot Name Signature
     0      5    0 this Ljdk8/byte_code/Example4_method_reference;  void
example() throws java.lang.InterruptedException;
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: new           #2                  // class
jdk8/byte_code/Server$HttpServer
         3: dup
         4: invokespecial #3                  // Method
jdk8/byte_code/Server$HttpServer."<init>":()V
         7: invokedynamic #4,  0              // InvokeDynamic #0:test:
()Ljava/util/function/Predicate;
        12: invokestatic  #5                  // Method
jdk8/byte_code/WaitFor.waitFor:
(Ljava/lang/Object;Ljava/util/function/Predicate;)V
        15: return
      LineNumberTable:
        line 11: 0
        line 12: 15
      LocalVariableTable:
      Start Length Slot Name Signature
      0     16     0    this Ljdk8/byte_code/Example4_method_reference;
    Exceptions:
      throws java.lang.InterruptedException
  }
```

Example 5

```
package jdk8.byte_code;

import static jdk8.byte_code.Server.*;
import static jdk8.byte_code.WaitFor.waitFor;

public class Example5 {

    // closure
    void example() throws InterruptedException {
        Server server = new HttpServer();
        waitFor(() -> !server.isRunning());
    }
}
```

```
Classfile Example5.class
 Last modified 08-May-2014; size 1470 bytes
 MD5 checksum 7dd0f577d4b4b903500264acf9649c30
 Compiled from "Example5.java"
 public class jdk8.byte_code.Example5
 SourceFile: "Example5.java"
 InnerClasses:
   public static #31= #2 of #29; //HttpServer=class
jdk8/byte_code/Server$HttpServer of class jdk8/byte_code/Server
   public static final #68= #67 of #70; //Lookup=class
java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
 BootstrapMethods:
   0: #34 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 Method arguments:
 #35 ()Ljava/lang/Boolean; // <-- signature and return type of the SAM method to
be implemented by the lambda
 #36 invokestatic jdk8/byte_code/Example5.lambda$example$35:
(Ljdk8/byte_code/Server;)Ljava/lang/Boolean;
 #35 ()Ljava/lang/Boolean; // <-- signature and return type to be enforced at
invocation time
 minor version: 0
 major version: 52
 flags: ACC_PUBLIC, ACC_SUPER
 Constant pool:
 #1 = Methodref           #10.#28   //java/lang/Object."<init>":()V
 #2 = Class               #30       //jdk8/byte_code/Server$HttpServer
 #3 = Methodref           #2.#28    // jdk8/byte_code/Server$HttpServer."
<init>":()V
 #4 = InvokeDynamic       #0:#37    // #0:isSatisfied:
(Ljdk8/byte_code/Server;)Ljdk8/byte_code/Condition;
 #5 = Methodref           #38.#39   // jdk8/byte_code/WaitFor.waitFor:
(Ljdk8/byte_code/Condition;)V
 #6 = InterfaceMethodref  #29.#40   // jdk8/byte_code/Server.isRunning:
()Ljava/lang/Boolean;
```

```
 #7 = Methodref             #41.#42   // java/lang/Boolean.booleanValue:()Z
 #8 = Methodref             #41.#43   // java/lang/Boolean.valueOf:
(Z)Ljava/lang/Boolean;
 #9 = Class                 #44       // jdk8/byte_code/Example5
 #10 = Class                #45       // java/lang/Object
 #11 = Utf8                 <init>
 #12 = Utf8                 ()V
 #13 = Utf8                 Code
 #14 = Utf8                 LineNumberTable
 #15 = Utf8                 LocalVariableTable
 #16 = Utf8                 this
 #17 = Utf8                 Ljdk8/byte_code/Example5;
 #18 = Utf8                 example
 #19 = Utf8                 server
 #20 = Utf8                 Ljdk8/byte_code/Server;
 #21 = Utf8                 Exceptions
 #22 = Class                #46 // java/lang/InterruptedException
 #23 = Utf8                 lambda$example$35
 #24 = Utf8                 (Ljdk8/byte_code/Server;)Ljava/lang/Boolean;
 #25 = Utf8                 StackMapTable
 #26 = Utf8                 SourceFile
 #27 = Utf8                 Example5.java
 #28 = NameAndType          #11:#12   // "<init>":()V
 #29 = Class                #47       // jdk8/byte_code/Server
 #30 = Utf8                 jdk8/byte_code/Server$HttpServer
 #31 = Utf8                 HttpServer
 #32 = Utf8                 InnerClasses
 #33 = Utf8                 BootstrapMethods
 #34 = MethodHandle         #6:#48    // invokestatic
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #35 = MethodType           #49       // ()Ljava/lang/Boolean;
 #36 = MethodHandle         #6:#50    // invokestatic
jdk8/byte_code/Example5.lambda$example$35:
(Ljdk8/byte_code/Server;)Ljava/lang/Boolean;
 #37 = NameAndType          #51:#52   // isSatisfied:
(Ljdk8/byte_code/Server;)Ljdk8/byte_code/Condition;
 #38 = Class               #53        // jdk8/byte_code/WaitFor
 #39 = NameAndType         #54:#55    // waitFor:(Ljdk8/byte_code/Condition;)V
 #40 = NameAndType         #56:#49    // isRunning:()Ljava/lang/Boolean;
 #41 = Class               #57        // java/lang/Boolean
 #42 = NameAndType         #58:#59    // booleanValue:()Z
 #43 = NameAndType         #60:#61    // valueOf:(Z)Ljava/lang/Boolean;
 #44 = Utf8                jdk8/byte_code/Example5
 #45 = Utf8                java/lang/Object
 #46 = Utf8                java/lang/InterruptedException
 #47 = Utf8                jdk8/byte_code/Server
 #48 = Methodref           #62.#63    //
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #49 = Utf8                ()Ljava/lang/Boolean;
 #50 = Methodref           #9.#64     //
jdk8/byte_code/Example5.lambda$example$35:
(Ljdk8/byte_code/Server;)Ljava/lang/Boolean;
 #51 = Utf8                isSatisfied
 #52 = Utf8           (Ljdk8/byte_code/Server;)Ljdk8/byte_code/Condition;
 #53 = Utf8                jdk8/byte_code/WaitFor
 #54 = Utf8                waitFor
 #55 = Utf8                (Ljdk8/byte_code/Condition;)V
 #56 = Utf8                isRunning
 #57 = Utf8                java/lang/Boolean
 #58 = Utf8                booleanValue
 #59 = Utf8                ()Z
 #60 = Utf8                valueOf
 #61 = Utf8                (Z)Ljava/lang/Boolean;
 #62 = Class               #65        // java/lang/invoke/LambdaMetafactory
 #63 = NameAndType         #66:#69    // metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #64 = NameAndType         #23:#24    // lambda$example$35:
(Ljdk8/byte_code/Server;)Ljava/lang/Boolean;
 #65 = Utf8                java/lang/invoke/LambdaMetafactory
 #66 = Utf8                metafactory
 #67 = Class               #71        // java/lang/invoke/MethodHandles$Lookup
 #68 = Utf8                Lookup
 #69 = Utf8
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Meth
odType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/in
voke/MethodType;)Ljava/lang/invoke/CallSite;
 #70 = Class               #72        //java/lang/invoke/MethodHandles
 #71 = Utf8                java/lang/invoke/MethodHandles$Lookup
 #72 = Utf8                java/lang/invoke/MethodHandles
{
  public jdk8.byte_code.Example5();
   descriptor: ()V
   flags: ACC_PUBLIC
   Code:
     stack=1, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1            // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 6: 0
    LocalVariableTable:
      Start Length Slot Name Signature
          0      5    0 this Ljdk8/byte_code/Example5;
void example() throws java.lang.InterruptedException;
  descriptor: ()V
  flags:
  Code:
    stack=2, locals=2, args_size=1
       0: new           #2            //class jdk8/byte_code/Server$HttpServer
       3: dup
       4: invokespecial #3            // Method
jdk8/byte_code/Server$HttpServer."<init>":()V
       7: astore_1
       8: aload_1

       9: invokedynamic #4,  0       // InvokeDynamic #0:isSatisfied:
(Ljdk8/byte_code/Server;)Ljdk8/byte_code/Condition;
       14: invokestatic  #5                  // Method
jdk8/byte_code/WaitFor.waitFor:(Ljdk8/byte_code/Condition;)V
       17: return
      LineNumberTable:
        line 10: 0
        line 11: 8
        line 12: 17
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      18     0  this   Ljdk8/byte_code/Example5;
            8      10     1 server   Ljdk8/byte_code/Server;
    Exceptions:
      throws java.lang.InterruptedException
}
```