Lambda Expressions in Java

Tutorial

Angelika Langer & Klaus Kreft

Angelika Langer Training/Consulting - www.AngelikaLanger.com

Lambda Expressions in Java - A Tutorial

by Angelika Langer & Klaus Kreft

ISBN

Table of Contents

Questions & Answers

What is the target type of a lambda expression? A type to which the lambda expression can be converted in a given context; the target type must be a functional interface type.	**target type**	17
What is a functional interface? An interface with a single abstract method.	**functional interface**	17
How does the syntax of lambda expressions differ from the syntax of anonymous inner class? The syntax of lambda expressions is less verbose and more readable.	**syntax**	20
For an anonymous inner class type definition and instance creation are tied together. How are lambda expression translated? Creation of the lambda object and its type is implicitly taken care of by the virtual machine; it is done at runtime.	**runtime representation**	21
Anonymous inner class can have bindings to variables of the enclosing scope. Is the same true for lambdas? Yes, lambda expressions can capture effectively final variables from their enclosing scope and can bind to the enclosing instance via `this` and `super`.	**variable binding**	22
An anonymous inner class is a name scope of its own. How about lambda expressions? Lambda expressions are part of the scope in which they appear; they are not scopes of their own.	**lexical scoping**	23
What does `this` refer to in a lambda expression? It refers to the enclosing instance (different from an anonymous class where `this` refers to the inner class's instance).	**meaning of this/super**	23
What do we need lambda expressions in Java for? To enable convenient use of the overhauled collection framework in general and it parallel bulk operations in particular.	**collection framework extension**	24
What is a bulk operation? An operation that concern many or all elements in a sequence.	**bulk operations**	24
What is internal and external iteration? External iteration uses an iterator for access to all	**internal iteration**	25

Yes, a class can inherit non-abstract methods from one superclass and multiple interfaces.	

Lambda Expressions

Background and a Bit of Trivia

Many popular programming languages nowadays have a language feature known as "lambdas" aka "closures". Such languages do no only include classic functional languages such as Lisp or Scheme, but also younger ones such as JavaScript, Python, Ruby, Groovy, Scala, C#, and even C++ has lambdas.[1] Some of these lambda-supporting languages run on the Java Virtual Machine, and, naturally, Java as the most prominent programming language on the JVM did not want to be left behind. A discussion regarding the addition of lambdas/closures began in 2006, shortly after Java 5 had been released. Three perfectly reasonable proposals for closures emerged, but unfortunately they did not converge. Instead there was a heated debate regarding the pros and cons of the various diverging ideas[2]. In 2009 the effort grounded to a halt and it ultimately looked like Java would never ever be extended by a lambda or closure feature. Consequently, Java's immanent decline as a vibrant programming language was predicted and the slogan of "Java is the new Cobol" was born.

Then, in 2010, Oracle declared that Java's death is not an option and that - quite the converse - Java is supposed to stay around for many years to come as a popular language in wide-spread use[3]. For that to happen, Java is required to adapt to the needs of modern hardware and in particular modern and future multi-core platforms. On a platform with multiple CPU cores many operations can and should be executed in parallel. While concurrent programs keep a couple of dozens of CPU cores busy with the fairly coarse-grained parallelization that is common practice today, it is obvious that it would take a more fine-grained approach to keep hundreds of CPU cores busy. In order to support fine-grained parallelization, it was decided that the JDK collections framework must undergo a major overhaul. Collections need bulk operations that apply functionality to all elements in a sequence and do so with several threads in parallel.

---

[1] See Wikipedia for examples of the syntax in which these languages express lambdas: http://en.wikipedia.org/wiki/Lambda_calculus#Lambda_calculus_and_programming_languages.

[2] See the "Closure Debate" at http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html for an overview.

[3] See Mark Reinhold's blog at https://blogs.oracle.com/mr/entry/closures.

Implementation of parallel bulk operations for collections requires a better separation of concerns, namely separating "what" is applied to all sequence elements from "how" it is applied to the sequence elements. And this is what lambdas are for: they provide a convenient and concise notation for functionality, which can be passed as an argument to a bulk operation of a collection, which in turn applies this functionality in parallel to all its elements.

What are Lambda Expressions?

The term *lambda* (short for: *lambda expression*) stems from *Lambda calculus*, which is a theoretical framework for describing functions and their evaluation. Rather than delving into the theoretical background let us keep it simple: a lambda expression denotes a piece of functionality. Here is an example of a lambda expression in Java:

```
(int x) -> { return x+1; }
```

Merely looking at it tells that a lambda expression is something like a method without a name. It has everything that a method has: an argument list, which is the `(int x)` part of the sample lambda, and a body, which is the `{ return x+1; }` part after the arrow symbol. Compared to a method a lambda definition lacks a return type, a throws clause, and a name. Return type and exceptions are inferred by the compiler from the lambda body; in our example the return type is `int` and the throws clause is empty. The only thing that is really missing is the name. In this sense, a lambda is a kind of anonymous method. ad-hoc functionality

Lambda Expressions vs. Anonymous Inner Classes

Lambdas are implemented "ad hoc". That is, where and when they are needed. To this regard they are similar to a Java language feature that we have been using all along, namely anonymous inner classes. Anonymous inner classes, too, are "ad hoc" implementations - not just of a single method, but of an entire class with one or several methods. Both lambdas and anonymous inner classes are typically defined for the purpose of passing them as an argument to a method, which takes the anonymous functionality and applies it to something.

Here is a method from the JDK that takes a piece of functionality and applies it; it is the `listFiles(FileFilter)` method from class `java.io.File`.

```
public File[] listFiles(FileFilter filter) {
        String ss[] = list();
        if (ss == null) return null;
```

```
        ArrayList<File> files = new ArrayList<>();
        for (String s : ss) {
            File f = new File(s, this);
            if ((filter == null) || filter.accept(f))
                files.add(f);
        }
        return files.toArray(new File[files.size()]);
}
```

The `listFiles(FileFilter)` method takes a piece of functionality as an argument. The required functionality must be an implementation of the `FileFilter` interface:

```
public interface FileFilter {
    boolean accept(File pathname);
}
```

A file filter is a piece of functionality that takes a `File` object and returns a `boolean` value. The `listFiles` method applies the filter that it receives as an argument to all `File` objects in the directory and returns an array containing all those `File` objects for which the filter returned `true`.

For invocation of the `listFiles` method we must supply the file filter functionality, i.e. an implementation of the `FileFilter` interface. Traditionally, anonymous inner classes have been used to provide ad hoc implementations of interfaces such as `FileFilter`.

Using an anonymous inner class the invocation of the `listFiles` method looks like this:

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
   File[] files = myDir.listFiles(
     new FileFilter() {
      public boolean accept(File f) { return f.isFile(); }
     }
   );
}
```

Using a lambda expression it looks like this:

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
   File[] files = myDir.listFiles(
     (File f) -> { return f.isFile(); }
   );
}
```

The example illustrates a typical use of lambdas. We need to define functionality on the fly for the purpose of passing it to a method. Traditionally, we used anonymous inner classes in such situations, which means: a class is defined, an instance of the class is created and passed to

the method. Since Java 8, we can use lambda expressions instead of anonymous inner classes. In both cases we pass functionality to a method like we usually pass objects to methods: This concept is known as "code-as-data", i.e. we use code like we usually use data. Using anonymous inner classes, we do in fact pass an object to the method. Using lambda expressions object creation is no longer required[4]; we just pass the lambda to the method. That is, with lambda expression we really use "code-as-data".

In addition to getting rid of instance creation, lambda expressions are more convenient. Compare the alternatives:

Using an anonymous inner class it looks like this:

```
File[] files = myDir.listFiles(
   new FileFilter() {
     public boolean accept(File f) { return f.isFile(); }
   }
);
```

Using a lambda expression it looks like this:

```
File[] files = myDir.listFiles(
   (File f) -> { return f.isFile(); }
);
```

We will later[5] see, that in addition to lambda expressions there is a feature called *method references*. They can be used similar to lambdas. With a method reference the example looks even simpler.

Using a method reference it looks like this:

```
File[] files = myDir.listFiles( File::isFile );
```

Clearly, use of a lambda expression or a method reference reduces the file filter to its essentials and much of the syntactic overhead that definition of an anonymous inner class requires is no longer needed.

---

[4] More precisely, object creation is no longer explicit. Under the hood, a lambda expression is eventually translated into a synthetic class type and an instance thereof. The key difference is that the class definition and instance creation is explicitly done by the programmer, when anonymous inner classes are used, whereas the class definition and instance creation is implicitly done by the runtime system. Details regarding the translation process can be found in the section on "Lambda Translation" in the *Lambda Reference* document.

[5] See the section on "Method References" in the *Lambda Reference* document.

Methods vs. Functions

Lambda expressions are a key concept in so-called *functional programming languages*. Java in contrast is an object-oriented programming language. Adding an element from a functional language to Java allows for programming techniques that were previously uncommon due to lack of support. In order to understand what lambdas are and how they can be used we want to look into some of the principles of functional programming. An important issue is the difference between a *method* and a *function*.

Note, that in the following, we use the terms "method" and "function" to denote concepts and to point out the difference between the two concepts. In practice, the terms "method", "function", and "procedure" are often used interchangeably to describe similar principles. Also, in practice, the difference between the concepts may be not be very pronounced. Since we want to explore how lambdas differ from other concepts in Java that we are familiar with, we will focus on the differences between the concepts of a method and a function.

In order to grasp the difference between methods and functions, we take a brief dip into the history and principles of programming languages.

Both methods and functions represent a piece of reusable functionality. They both take arguments, have some kind of body, which is executable code and represents the actual functionality, and they may produce a result and/or side effects. In Java, a *method* always belongs to a class; Java has no concept of *free functions* outside of a class. A method has read and write access to fields of the class to which it belongs and may modifies these fields. A method can also modify the arguments it receives and produce other side effects, but most of the time methods modify fields or produce a result. A *function*, in contrast, operates solely on the arguments it receives. It typically does not modify any data; often it does not even have access to any mutable data.

Methods and functions are used in different ways. The idea of methods (or procedures) stems from *procedural* (aka *imperative*) languages such as Fortran, Pascal, and C. In those languages procedures are invoked in a specific order and may operate on data which they read and modify. The term "procedural" stems from the use of "procedures"; the notion of an "imperative" language stems from the fact that the programmer dictates how things are done, i.e., in which order which procedures are applied to which data. In an imperative language the order of applying procedures matters. This is obvious when we consider that procedures may modify data. After all, it makes a substantial difference in which order

modifications happen and whether data is read before or after a modification.

Object-oriented languages such as C++, Smalltalk, Eiffel, and Java extend the procedural approach. They bundle data and procedures into objects with state (the fields) and behaviour (the methods). The principle of using methods is still the same as in procedural languages: methods are invoked and operate on data (the fields of the class to which the method belongs), which they read and may alter. For this reason, object-oriented languages are imperative languages, too.

The idea of functions stems from *functional* languages such as Lisp, Haskell, Closure, and Scala. These languages have pure functions that behave as described above: they do not mutate data, instead they produce results from the arguments they receive. In addition, functions are used differently. A function, too, is invoked, but the key idea is that functions are passed around as arguments to an operation or as the result of an operation. This is similar to passing around data or objects in procedural and object-oriented languages - hence the previously mentioned notion of "code-as-data".

Functional languages are *declarative* as opposed to *imperative*. They describe *what* a program should accomplish, rather than describing *how* to go about accomplishing it. In a declarative language the programmer does not dictate which steps must be performed in which order. Instead, there is a clear separation between *what* is done (this is what the programmer declares) and *how* it is done (this is determined elsewhere, by the language or the implementation). In *pure functional* languages such as Haskell the functions do not have any side effects. In particular, they do not modify data or objects. If no side effects occur, order does not matter. Consequently it is easy to achieve a clear separation between "how" and "what" is done.

Hence there are the following differences between the concept of methods and pure functions :

	Method	**Pure Function**
Mutation	A method *mutates*, i.e., it modifies data and produces other changes and side effects.	A pure function *does not mutate*; i.e., it just inspects data and produces results in form of new data.
How vs. What	Methods describe *how* things are done.	Pure functions describe *what* is done rather than how it is done.

Invocation Order	Methods are used *imperatively*, i.e., order of invocation matters.	Pure functions are used *declaratively*; i.e., order of invocation does not matter.
Code vs. Data	Methods are *code*, i.e., they are invoked and executed.	Functions are *code-as-data*; i.e., they are not only executable, but are also passed around like data.

The bottom line is:

- We can express *methods* in Java. These are the methods of Java classes that we are familiar with. They are executed, they may mutate data, the order of their invocation matters, and they describe how things are done.
- We can express *pure functions* in Java. Traditionally, functions were expressed as anonymous inner classes and now can be expressed as lambda expressions. Lambdas and anonymous classes are passed around like data, typically as arguments to a method. If they do not modify data they are *pure* functions. In this case their invocation order does not matter and they describe what is done rather than how it is done.

Since Java is a hybrid language (object-oriented with functional elements), the concepts are blurred. For instance, a method in Java need not be a mutator. If it does not modify anything, but purely inspects data, then it has properties of a function. Conversely, a function in Java (i.e., an anonymous inner class or lambda expression) need not be pure. It may modify data or produce other side effects, in which case it has properties of a method. Despite of the fuzziness in practice, it may be helpful to keep the concepts of "method" and "function" in mind.

Let us revisit the previous example of a file filter to check what the concepts look like in practice:

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
   File[] files = myDir.listFiles(
     (File f) -> { return f.isFile(); }
   );
}
```

The `listFiles` method of class `java.io.File` is a *method*. It operates on the static and non-static data of the `File` object it is invoked on. A `File` object contains fields that give access to properties of the file system

element it represents, e.g. its pathname, whether it is a file or a directory, which other elements it contains, etc. These fields are accessed by the `listFiles` method.

The `listFiles` method takes a file filter as an argument. The file filter is a *function*, here expressed as a lambda expression. It is passed to the `listFiles` method which then applies it to all files and directories in `myDir`. The example illustrates the code-as-data principle: the function (i.e. the file filter lambda) is passed to an operation (i.e. the `listFiles` method). It also illustrates the separation between "what" and "how": the function describes "what" is supposed to be applied to all files and directory and the receiving method controls "how" the filter is applied, e.g. in which order. In addition, the example illustrates the principle of a pure function: the file filter does not produce side effects; it just takes a `File` and returns a `boolean`.

In contrast, here is an example of an *impure* function:

```
File myDir = new File("\\user\\admin\\deploy");
final LongAdder dirCount = new LongAdder();
final LongAdder filCount = new LongAdder();
if (myDir.isDirectory()) {
   myDir.listFiles( (File f) -> {
                        if (f.isDirectory()) {
                            dirCount.increment();
                            return false;
                        }
                        if (f.isFile()) {
                            filCount.increment();
                            return false;
                        }
                        return false;
                    }
   );
}
```

The file filter is again expressed as a lambda, but this time it produces side effects, namely incrementing two counters.

After this excursion into the realm of programming language principles, we know that lambda expressions in Java denote (pure or impure) functions. As Java is an object-oriented and not a functional programming language the question is: How are lambdas aka functions integrated into the Java programming language?

Representation of Lambda Expressions

As we have seen in previous examples, we use lambda expressions to define a *function*, which we can pass as an argument to an operation. On

the receiving side, i.e., in the operation that takes the lambda, it is used like an *object*.

For illustration we will again use the previous example of the `listFiles` method from class `java.io.File`. Below we see where the lambda expression is defined and passed on as an argument to the `listFiles` method.

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
   File[] files = myDir.listFiles(
     (File f) -> { return f.isFile(); }
   );
}
```

Here is the receiving side, i.e., the implementation of the `listFiles` method from class `java.io.File`:

```
public File[] listFiles(FileFilter filter) {
    String ss[] = list();
    if (ss == null) return null;
    ArrayList<File> files = new ArrayList<>();
    for (String s : ss) {
        File f = new File(s, this);
        if ((filter == null) || filter.accept(f))
            files.add(f);
    }
    return files.toArray(new File[files.size()]);
}
```

The `listFiles` method take a `FileFilter` object and invokes its `accept` method.

The example demonstrates that a lambda expression has properties in common with functions and objects, depending on the point of view.

- Conceptually, a lambda expression is a *function*. It is an unnamed piece of reusable functionality. It has a signature and a body, but no name. It may or may not be pure, i.e., may or may not have side effects. We pass it around as an argument to a method or as a return value from a method.
- When a lambda expression is passed as an argument to a method the receiving method treats it like an *object*. In the example the lambda expression (or more precisely, a reference to the lambda expression) is received as the argument of the `listFiles` method. Inside the `listFiles` method the lambda expression is a reference to an object of a subtype of the `FileFilter` interface. This *lambda object* is a regular object; e.g. it has an address and a type.

Basically, you think in terms of a function when you define a lambda expression and in terms of a method when you use it. The lambda object that links definition and use together is taken care of by the compiler and the runtime system. We as users need not know much about it.

Practically all decisions regarding the representation and creation of the lambda object (short for: the object that represents a lambda expression at runtime) are made by the compiler and the runtime system. This is good, because it enables optimizations under the hood that we as users neither want to deal with nor care about. For instance, from the context in which the lambda expression is defined the compiler captures all information that is needed to create a synthetic type for a lambda object, but the compiler does not yet create that type; it just prepares the type generation. The actual creation of the lambda object's synthetic type and the creation of the lambda object itself are done dynamically at runtime by the virtual machine. For this dynamic creation of the synthetic lambda type and the lambda object the `invokedynamic` byte code is used, which was added to Java in release 7. Using these dynamic features it is possible to delay the creation until first use, which is an optimization: if you just define, but never use, the lambda expression then neither its type nor the lambda object will ever be created.[6]

Functional Interfaces

Part of the magic of tying the definition of a lambda expression to the use of a lambda expression is inference of the type by which the lambda expression is used. This type is called the *target type*. The synthetic type that the runtime system dynamically creates, as mentioned above, is a subtype of the target type.

In the previous file filter example we have seen that the target type of our lambda expression is `FileFilter`. In the example we define a lambda expression, pass it to the `listFiles` method, in which it is then used like an object of a subtype of `FileFilter`. In a way, this is surprising. After all, we did not specify that our lambda expression implements the `FileFilter` interface. In fact, we did not say anything regarding the lambda expression's type. Similarly, the receiving `listFiles` method never indicated that it happily accepts lambda expressions. Instead it requires an object of a subtype of `FileFilter`. How does it work?

---

[6] If you are interested in the details of the creation of the lambda object please read the section on "Lambda Translation" in the *Lambda Reference* document or take a look at http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html.

The underlying magic that the compiler performs to make it happen is *type inference*. The compiler takes a look at the context in which the lambda expression is defined and figures out which type is required. Then it takes a look at the lambda expression itself and figures out whether the lambda is compatible to the required type (plus minus a couple of adjustments if needed).

If Java were a functional language then the most natural type for a lambda expression would be some kind of *function type*, which is a special category of type that is reserved for description of functions. A function type merely describes the signature and could for example look similar to `(int,int)->boolean` if it is meant to denote the type of a function that takes two `int` arguments and returns a `boolean` value.

Java is no functional language and traditionally had no such thing as a function type. So, the language designers had the choice to add function types as new category of type. Since they did not want to introduce major changes to Java's type system they tried to find a way to integrate lambda expressions into the language without resorting to function types. They found a way to do it by using special interface types instead: these interfaces as so-called *functional interfaces*, previously known as *SAM* types (where SAM stands for Single Abstract Method).

A functional interface essentially is an interface with one method.[7] Countless such interfaces exist in the JDK already, some of them since its first release. The interface `Runnable` is a classic example of a functional interface. It demands the implementation of exactly one method: `void r un()`. There are many more: `Readable`, `Callable`, `Iterable`, `Closeable`, `Flushable`, `Formattable`, `Comparable`, `Comparator`, or the `FileFilter` interface, which we have been using in the previous example.

As functional interfaces and lambda expressions both deal with a single method, the language designers decided that the compiler would convert each lambda expression to a matching functional interface type. This conversion usually happens automatically, because lambda expressions may only appear in a context where such a conversion is feasible.

Let us reconsider the previous example:

```
File[] files = myDir.listFiles(
```

[7] Details regarding functional interfaces can be found in the section on "Functional Interfaces" in the *Lambda Reference* document. These details include, for instance, that functional interfaces can under certain circumstances have more than one method. Also, there is a `@FunctionalInterface` annotation that enables compiler checks for interfaces that are supposed to be used as functional interfaces.

```
    (File f) -> { return f.isFile(); }
);
```

The high-lighted part is the lambda expression. It appears as the argument to the invocation of the `listFiles` method of class `java.io.File`. The compiler knows the signature of method `listFiles` and finds that the method declares `java.io.FileFilter` as its parameter type. Hence the required type in this particular context is `FileFilter`.

Here is what the `FileFilter` interface looks like:

```
public interface FileFilter { boolean accept(File pathname); }
```

`FileFilter` requires exactly one method and for this reason is a functional interface type. Our lambda expression has a matching signature: it takes a `File` argument, returns a `boolean` value and throws no checked exception. Hence the compiler converts the lambda expression to the functional interface type `FileFilter`. Inside the `listFiles` method the lambda is then indistinguishable from an object of type `FileFilter`. Here is the implementation of method `File.listFiles`:

```
public File[] listFiles(FileFilter filter) {
        String ss[] = list();
        if (ss == null) return null;
        ArrayList<File> files = new ArrayList<>();
        for (String s : ss) {
            File f = new File(s, this);
            if ((filter == null) || filter.accept(f))
                files.add(f);
        }
        return files.toArray(new File[files.size()]);
}
```

Inside this method the lambda is bound to the `filter` argument. The lambda's functionality is triggered when the functional interface's method is invoked, i.e. when `filter.accept()` is called.

The conversion to a functional interface can have funny effects, demonstrated in the following example:

Say, we have two functional interfaces:

```
public interface FileFilter { boolean accept(File pathname); }
```

and

```
public interface Predicate<T> { boolean test(T t); }
```

Our lambda is convertible to both functional interface types[8]:

```
FileFilter         filter = (File f) -> { return f.isFile(); };
Predicate<File> predicate = (File f) -> { return f.isFile(); };

filter = predicate;    // << error: incompatible types
```

If, however, we try to assign the two resulting variables to each other the compiler would flag it as an error, although the two variables represent the same lambda expression. The reason is that the two variables to which the lambda is bound are of two different, incompatible types.

Also, there might occasionally be situations in which the compiler cannot figure out a matching functional interface type, like in this example:

```
Object ref  = (File f) -> { return f.isFile(); };
```

The assignment context does not provide enough information for a functional type conversion and the compiler reports an error. Adding a type cast easily solves the problem:

```
Object ref  = (FileFilter) (File f) -> { return f.isFile(); };
```

By and large, functional interface conversion is a way to keep matters simple and avoid the addition of function types to Java's type system.[9]

Comparing Lambdas to Anonymous Inner Classes

Lambda expressions appear in situations where we traditionally used anonymous inner classes, namely in places where an operation asks for a piece of functionality (i.e. for a function). While lambda expressions and anonymous inner classes are somewhat exchangeable they differ in many ways from each other.

---

[8] Expressions that have different type depending on the context in which they appear are called *poly expression*. The actual type of a poly expression is always inferred by the compiler. Lambda expressions and method references are examples of poly expression. Poly expressions also occur in conjunction with generic; for example instance creation expressions that use a diamond <> like `new ArrayList<>()` are poly expressions.

[9] Details regarding functional interface conversion can be found in the sections on "Target Typing" and "Type Inference" in the *Lambda Reference* document. There you find details regarding which context is suitable for functional interface conversion and how conversion to generic functional interfaces works.

Syntax

Quite obviously the syntax is different. The notation for a lambda expression is more concise. Here are examples illustrating how concise a lambda notation can be compared to an anonymous inner class:

Anonymous inner class:

```
File[] fs = myDir.listFiles(
  new FileFilter() {
    public boolean accept(File f) { return f.isFile(); }
  }
);
```

Lambda expression in various forms including method reference:

```
File[] files = myDir.listFiles( (File f) -> {return f.isFile();} );
File[] files = myDir.listFiles( f -> f.isFile() );
File[] files = myDir.listFiles( File::isFile );
```

Runtime Overhead

Anonymous inner classes come with a certain amount of overhead compared to lambda expressions. Use of an anonymous inner class involves creation of a new class type and creation of a new object of this new type and eventually invocation of a non-static method of the new object.

At runtime anonymous inner classes require:

- class loading,
- memory allocation and object initialization, and
- invocation of a non-static method.

A lambda expression needs functional interface conversion and eventually invocation. The type inference required for the functional interface conversion is a pure compile-time activity and incurs no cost at runtime. As mentioned earlier, the creation of the lambda object and its synthetic type is performed via the `invokedynamic` byte code. This allows to defer all decisions regarding type representation, the instance creation, and the actual invocation strategy to runtime. This way the JVM can choose the optimal translation strategy. For instance, the JVM can reduce the effort to the invocation of a constant static method in a constant pool, which eliminates the need for a new class and/or a new object entirely.

The bottom line is that lambda expressions have the potential for optimizations and reduced runtime overhead compared to anonymous inner classes.[10]

Variable Binding

Occasionally, a function needs access to variables from the enclosing context, i.e., variables that are not defined in the function itself, but in the context in which the lambda appears. Anonymous inner classes have support for this kind of *variable binding* (also called *variable capture*): an inner class has access to all `final` variables of its enclosing context. Here is an example:

```
void method() {
        final int cnt = 16;

        Runnable r = new Runnable() {
            public void run() {
                System.out.println("count: " + cnt);
            }
        };
        Thread t = new Thread(r);
        t.start();

        cnt++;   // error: cnt is final
}
```

The anonymous inner class that implements the `Runnable` interface accesses the `cnt` variable of the enclosing method. It may do so, because the `cnt` variable is declared as `final`. Of course, the final `cnt` variable cannot be modified.

Lambda expressions, too, support variable binding. Here is the same example using a lambda expression:

```
void method() {
        int cnt = 16;

        Runnable r = () -> { System.out.println("count: " + cnt);
};

        Thread t = new Thread(r);
        t.start();

        cnt++;   // error: cnt is implicitly final
}
```

[10] Details regarding the translation of lambda expressions can be found in the sections on "Translation of Lambda Expressions" in the *Lambda Reference* document.

The difference is that the `cnt` variable need not be declared as `final` in the enclosing context. The compiler automatically treats it as `final` as soon as it is used inside a lambda expression. In other words, variables from the enclosing context that are used inside a lambda expression are *implicitly final* (or *effectively final*). Naturally, you can add an explicit `final` declaration, but you do not have to; the compiler will automatically add it for you.

Note, since Java 8, the explicit `final` declaration is no longer needed for anonymous inner classes as well. Since Java 8, both lambda expressions and anonymous inner classes are treated the same regarding variable binding. Both can access all effectively final variables of their respective enclosing context.

Scoping

An anonymous inner class is a class, which means that it introduces a scope for names defined inside the inner class. In contrast, a lambda expression is *lexically scoped*, which means that the lambda expression is not a scope of it own, but is part of the enclosing scope. Here is an example that illustrates the difference:

Anonymous inner class:

```
void method() {
  int cnt = 16;
  Runnable r = new Runnable() {
    public void run() { int cnt = 0;  // fine
                        System.out.println("cnt is: " + cnt); }
  };
  ...
}
```

Lambda expression:

```
void method() {
  int cnt = 16;
  Runnable r = () -> { int cnt = 0;  // error: cnt has already been defined
                       System.out.println("cnt is: " + cnt);
                     };
  ...
}
```

Both the anonymous class and the lambda define a variable named `cnt` while there already is a variable with this name defined in the enclosing method. Since the anonymous class establishes a scope of it own it may define a second `cnt` variable which is tied to the scope of the inner class. The lambda, in contrast, is part of the enclosing scope and the compiler

considers the definition of a `cnt` variable in the lambda body a colliding variable definition. In the scope of the enclosing context there already is a definition of a `cnt` variable and the lambda must not define an additional `cnt` variable of its own.

Similarly, the different scoping rules have an impact on the meaning of keywords such as `this` and `super` when used inside an anonymous class or a lambda expression. In an anonymous class `this` denotes the reference to the object of the inner class type and `super` refers to the anonymous class's super class. In a lambda expression `this` and `super` mean whatever they mean in the enclosing context; usually `this` will refer to the object of the enclosing type and `super` will refer to the enclosing class's super class. Essentially it means, that it might not always be trivial to replace an anonymous class by a lambda expression. IDEs might help and offer functionality that refactors anonymous classes into lambda expressions.

Why do we need lambdas?

The key reason for adding lambdas to the Java programming language is the need to evolve the JDK and in particular the JDK's collection framework.[11] The traditional API design of the Java collections in package `java.util` renders certain optimizations impossible. At the same time, these optimizations are badly needed. Considering the current and future hardware architecture that we design our software for, we need support for increased parallelism and improved data locality. Let us see why.

Multi-core and multi-CPU platforms are the norm. While we deal with a couple of cores today, we will have to cope with hundreds of cores some time in the future. Naturally, we need to increase the effort for replacing serial, single-threaded execution by parallel, multi-threaded execution. The JDK intends to support this effort by offering *parallel bulk operations for collections*. An example of a parallel bulk operation is the application of a transformation to each element in a collection. Traditionally such a transformation is done in a loop, which is executed by a single thread, accessing one sequence element after the other. The same can be accomplished in parallel with multiple threads: break down the task of transforming *all* elements in a sequence into many subtasks each of which operates on a *subset* of the elements. Naturally, this can be done without extending the collection framework, e.g. by use of existing JDK abstractions such as the fork-join-framework for instance, but it is quite a

[11] See Mark Reinhold's blog at https://blogs.oracle.com/mr/entry/closures.

bit of an effort and requires a thorough understanding of concurrency issues. Parallel execution of bulk operations on collection is not a triviality. Hence, a key goal of evolving the JDK collections is support for parallelism in a way that is easy and convenient to use. In order to make this happen the collections' API must be extended.

A key point of the extension is to separate "what" operations are applied to sequence elements in a collection from "how" the operations are applied to these elements. Without a separation of concerns the collections have no chance to shield us from the complications of parallel execution of bulk operations. Here is where we come full circle and the previously discussed principles of functional programming including lambdas come into play.

New abstractions have been added to the collection framework in package `java.util`, most prominently the *stream* abstractions in package `java.util.stream`. Collections can be converted to streams and streams, different from collections, access the sequence elements via *internal iteration* (as opposed to *external iteration*.)

Internal vs. External Iteration

Iteration is the process of traversing a sequence. It can be performed in two ways: as an *internal* or *external* iteration.[12] Internal iteration means that the sequence itself controls all details of the iteration process. External iteration means that the sequence only supports iteration, usually via a so-called *iterator*, but many aspects of the iteration are controlled by the collection's user. Traditionally, Java collections offer external iteration. Since Java 8, internal iteration is supported via streams. Let us see, what the difference between internal and external iteration is.

Traditionally in Java, elements in a collection are accessed via an iterator. For illustration we explore an example that uses a bank account abstraction named **`Account`**:

```
class Account {
  private long balance;
  ... constructors and more methods ...
  public long balance() { return balance; }
```

[12] The iterator pattern is one of the so-called Gang of Four design patterns (Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995). Design patterns: Elements of reusable object-oriented software . Addison-Wesley.)

```
}
```

We iterate over a list of bank accounts using a for-each loop:

```
private static void checkBalance(List<Account> accList) {
   for (Account a : accList)
      if (a.balance() < a.threshold) a.alert();
}
```

The for-each-loop internally uses an iterator and basically looks like this:

```
Iterator iter = accList.iterator();
while (iter.hasNext()) {
  Account a = iter.next();
  if (a.balance() < a.threshold) a.alert();
}
```

This loop has potential for improvement in several areas:

- The logic of traversing the sequence is intermingled with the functionality that is applied to each element in the sequence. Separating the concerns would allow for optimizations regarding how the elements are accessed.
- The execution is single-threaded. Multi-threaded execution is possible by means of the fork-join thread pool, but it would be substantially more complex.
- There is a certain amount of redundancy. The iterator's `hasNext` and `next` method have overlap; they both figure out whether there is a next element. The overhead could be eliminated.

The traditional access to elements in a sequence via an iterator is known as *external iteration* in contrast to *internal iteration*.

The idea of internal iteration is: the sequence itself determines how elements in a sequence are accessed and the sequence's user is only responsible for specifying what has to be applied to each element in the sequence. While the external iteration intermingles the logic of sequence traversal with the functionality applied to each of the sequence elements, internal iteration separates the concerns. Here is what internal iteration might look like. Since Java 8, a collection has a `forEach` method that looks like this:

```
public void forEach(Consumer<? super E> consumer);
```

It uses the `Consumer` interface, which looks like this:

```
public interface Consumer<T> { public void accept(T t); }
```

Given this `forEach` method the sequence's user only has to supply the functionality that is to be applied to each element in the sequence.

```
private static void checkBalance(List<Account> accList) {
   accList.forEach(
     (Account a) -> { if (a.balance() < a.threshold)
                          a.alert();
                    }
   );
}
```

Now the concerns are separated: the collection is responsibility for figuring out how to access all elements and how to apply the specified functionality to them. The user is responsible for supply of a reasonable functionality to be applied to the sequence elements. Given this separation, the implementation of the collection and its `forEach` method have the liberty to optimize away the overhead of the iterator's `hasNext` and `next` method, to split the task into subtasks and execute them in parallel, and several further optimizations such as lazy evaluation, pipeline optimizations for consecutive operations, or out-of-order execution. The traditional intermingled external iteration prevents all of these optimizations.

Given the separation of concerns it is now easy to have the loop executed in parallel by multiple threads. Since Java 8, collections can be turned into so-called *streams*, of which there is a sequential and a parallel variety. The parallel version of the example above would look like this:

```
private static void checkBalance(List<Account> accList) {
   accList.parallelStream().forEach(
     (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
   );
}
```

In the following section we take a closer look at streams, their relationship to collections, and their support for bulk operations via internal iteration.

Streams and Bulk Operations

Since Java 8 the JDK has a new abstraction named `Stream<E>` which is a view to a collection and has bulk operations that perform internal iteration. These bulk operations include `forEach`, `filter`, `map`, `reduce`, and many more. Thanks to the internal iteration, streams support sequential as well as parallel execution in a very convenient way. Here is an example:

```
List<Account> accountCol = … ;
accountCol
     .stream()
     .filter(a -> a.balance() > 1_000_000_00)          // intermediate
```

```
    .map(Account::balance)                                  // intermediate
    .forEach(b -> {System.out.format("%d\t",b);});  // terminal
```

Using method `stream()`, we create a stream on top of a collection. Different from a collection a stream does not contain any elements. Instead it is a view to a collection that gives access to all elements in the collection by means of bulk operations with internal iteration. A stream has operations such `filter()`, `map()` and `forEach()`, as illustrated in the code snippet above.

The `filter()` method takes a predicate, which is a function that takes an element from the sequence and returns a `boolean` value. In our example the predicate is the lambda expression `a->a.balance()>1_000_000_00`. This predicate is applied to all elements in the sequence and only the "good" elements, i.e. those for which the predicate returns `true`, appear in the resulting stream.

The `map()` method takes a mapper, which is a function that takes an element from the sequence and returns something else. In our example the mapper maps the accounts to their respective balance using the `balance()` method of class `Account`. The result is a stream of `long` values.

The `forEach()` method takes an arbitrary function, which receives an element from the sequence as an argument and is not supposed to return anything. In our example it prints the element.

Some of these methods (`filter()` and `map()` in the example) return a stream; other ones (`forEach()` in our example) do not. The methods that return streams are *intermediate* (aka *lazy*) operations, which means that they are not immediately executed, but delayed until a *terminal* (aka *eager*) operation is applied. In the example above, each bank account in the list is accessed only once, namely when the terminal `forEach` operation is executed. For each bank account the sequence of operations, i.e., filtering, mapping, and printing, is applied.

So far, the sequence of `filter-map-forEach` is executed sequentially, but it is easy to enable parallel execution. We simply need to create a parallel stream. Compare the sequential to the parallel version:

Sequential execution:

```
List<Account> accountCol = … ;
accountCol
  .stream()
  .filter( a -> a.balance() > 1_000_000_00 )          // intermediate
  .map(Account::balance)                                  // intermediate
  .forEach(b -> {System.out.format("%d\t",b);});      // terminal
```

Parallel execution:

```
List<Account> accountCol = … ;
accountCol
  .parallelStream()
  .filter( a -> a.balance() > 1_000_000_00 )          // intermediate
  .map(Account::balance)                              // intermediate
  .forEach(b -> {System.out.format("%d\t",b);});      // terminal
```

We create a parallel stream via the collection's `parallelStream()` method. This is all we need to say and the rest is taken care of by the implementation of the stream abstraction.

The extended collections framework with its streams and bulk operations as outlined above is just one area of Java and its JDK where lambda expressions are helpful and convenient. Independently of the JDK, lambdas can be useful for our own purposes. Before we explore a couple of "functional patterns" that necessitate lambdas we take a closer look at the lambda language feature, its syntax and its properties.

Programming with Lambdas

Lambdas are a concept that stems from functional programming languages and for this reason lambdas support a functional programming style as opposed to the imperative, object-oriented style Java programmers are used to. In the following we will first demonstrate the functional programming style using a simple case study related to the JDK collections and streams. Subsequently, we will explore an example of a functional programming pattern known as the *Execute-Around-Method Pattern*.

Fluent Programming

In the following we intend to explore the declarative programming style that lambda expressions and the JDK 8 streams support. We will be using the following abstractions:

```
public interface Person {
    public enum Gender { MALE, FEMALE };
    public int    getAge();
    public String getName();
    public Gender getGender();
}
public interface Employee extends Person {
    public long   getSalary();
}
```

```
public interface Manager extends Employee {
}
public interface Department {
    public enum Kind {SALES, DEVELOPMENT, ACCOUNTING,
                      HUMAN_RESOURCES}
    public Department.Kind getKind();
    public String          getName();
    public Manager         getManager();
    public Set<Employee>   getEmployees();
}
public interface Corporation {
    public Set<Department> getDepartments();
}
```

For illustration we want to retrieve all managers of all departments in a corporation that have an employee older than 65.

Imperative Approach

Here is the traditional imperative approach:

```
/*
 *   find all managers of all departments with an employee older
than 65
*/
Manager[] find(Corporation c) {
       List<Manager> result = new ArrayList<>();
       for (Department d : c.getDepartments()) {
          for (Employee e : d.getEmployees()) {
             if (e.getAge() > 65) {
              resu  lt.add(d.getManager());
             }
          }
 }
       return result.toArray(new Manager[0]);
}
```

The implementation is straight forward. We retrieve all departments from the corporation, then retrieve all employees per department, if we find an employee older than 65 we retrieve the department's manager, and add it to a result collection, which we eventually convert in the array that is returned.

Declarative Approach

Using streams and lambdas it would look like this:

```
/*
 *   find all managers of all departments with an employee older
than 65
```

```
*/
Manager[] find(Corporation c) {
  return
  c.getDepartments().stream()                                    // 1
                    .filter(d -> d.getEmployees().stream()       // 2
                                 .map(Employee::getAge)          // 3
                                 .anyMatch(a -> a>65))           // 4
                    .map(Department::getManager)                 // 5
                    .toArray(Manager[]::new)                     // 6
}
```

In line `//1`, we retrieve all departments of the corporation via the `getDepartments` method. The method yields a collection of `Department` references, which we turn into a stream (of `Department` references).

In line `//2`, we use the `filter` operation on the stream of departments. The `filter` operation needs a predicate. A predicate is a function that takes an element from the stream (a `Department` in our example) and returns a `boolean` value. The `map` operation applies the predicate to each element in the stream and suppresses all elements for which the predicate returns `false`. In other words, the stream returned from the `map` operation only contains those elements for which the filter has been returning `true`.

In our example we need a filter predicate that returns `true` if the department has an employee older than 65. We provide the predicate as a lambda expression. The lambda expression is fairly ambitious; it takes a `Department d`, retrieves all employees of that department via the `getEmployees` method, turns the resulting collection of employees into a stream, maps each employee to its age, and eventually applies the `anyMatch` operation.

In line `//3`, we use the `map` operation on the stream of employees. The `map` operation needs a function that takes an element from the stream (an `Employee` in that case) and returns another object of a potentially different type (the age in our example). The mapper function is provided as the method reference `Employee::getAge`. because the `getAge` method does exactly what we need: it maps an employee to its age.

In line `//4`, we use the `anyMatch` operation of the stream of age values return from the preceding `map` operation. The `anyMatch` operation takes elements from the stream, applies a predicate to each element, and stops as soon as an element is found for which the predicate returns `true`. We supply the required predicate as another lambda that takes the age and returns `true` if the age is greater than 65.

The result of the filter operation and the lengthy lambda expression in line `//2` to `//4` is the stream of all departments with an employee older than 65.

In line `//5`, we map each department to its respective manager. As the mapper function we use the method reference `Department::getManager`. The result is the stream of all managers of all departments with an employee older than 65.

So far, none of the lambdas or referenced methods have been executed because `filter` and `map` are intermediate operations.

In line `//6`, we convert the manager list into an array via the stream's `toArray` method. The `toArray` method needs a generator, which takes a size value and returns an array of the requested size. The generator in our example is a constructor reference, namely the reference to the constructor of manager arrays: `Manager[]::new`.

The two approaches look very different. The imperative style intermingles the logic of iterating (the various nested loops) and the application of functionality (retrieval of departments and employees, evaluation of age values, collecting results). The declarative style, in contrast, separates concerns. It describes which functionality is supposed to be applied and leaves the details of how the functions are executed to the various operations such as `filter`, `map`, `anyMatch`, and `toArray`.

Fluent Programming

The small code sample written in declarative style illustrates what is known as *fluent programming*: the chaining of operations. It is a programming technique where operations return a value that allows the invocation of another operation. With fluent programming it is perfectly natural to end up with one huge statement that is the concatenation of as many operations as you like.

The JDK streams are designed to support fluent programming: all intermediate operations return a result stream to which we can apply the next operation in the chain until we apply a terminal operation as the tail of the chain. The chaining of operations is a key feature of streams as it enables all sorts of optimizations. For instance, the stream implementation can arrange the execution of functions as a pipeline. Instead of looping over all elements in the sequence repeatedly (once for `filter`, then again for `map`, and eventually for `toArray`) the chain of filter-mapper-collector can be applied to each element in just one pass over the sequence.

Here is a brief illustration of the pipelining performed by the implementation of streams:

```
String[] txt =
{"State","of","the","Lambda","Libraries","Edition"};
IntStream is = Arrays.stream(txt)
                     .filter(s -> s.length() > 3)
                     .map(s -> s.length())
                     .forEach(l -> System.out.println(l));
```

In the code snippet we take a string array, turn it into a stream, pick all strings longer than 3 characters, and map them to their length.

The source code suggests that first the filter is applied to all elements in the stream and then the mapper is applied to all elements in a second pass over the stream. In reality, the chain of filter and map operation is postponed until the terminal `forEach` operation is executed. The terminal operation triggers only a single pass over the sequence during which both the filter and the mapper are applied to each element.

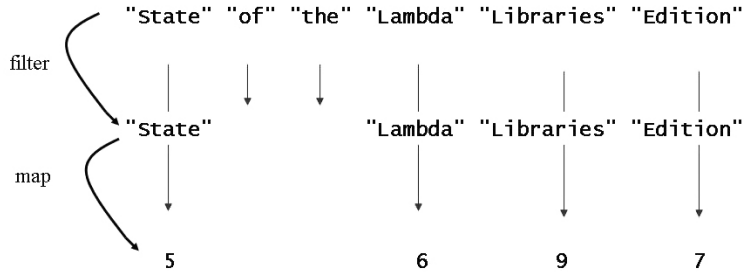

Diagram: Pipelined Execution of filter - map Chain

Creating such a pipeline per element does not only reduce multiple passes over the sequence to a single pass, but also allows further optimizations. For instance, the `length` method need not be invoked twice per string; the repeated call can be eliminated by the JIT compiler.

Which style looks more pleasant and understandable is to the eye of the beholder. Certainly the declarative code looks alien to Java programmer unfamiliar with the declarative or fluent programming. After getting accustomed to the declarative style it will probably appear equally pleasant and understandable.

Execute-Around-Method Pattern

So far, we have been exploring lambdas solely in the context of the JDK collections and streams. However, lambda expressions are useful beyond the JDK stream abstractions and internal iteration. They are convenient and helpful in all places where operations take functionality as an argument. One such occasion is the so-called *Execute-Around-Method pattern*. It is a programming technique for eliminating code duplication.

Occasionally we encounter situations where it is required that some boilerplate code must to be executed both before and after a method (or more generally, before and after another piece of code that varies). Often we simply duplicate the boilerplate code via copy-and-paste and insert the variable functionality manually. Following the DRY (don't repeat yourself) principle you might want to remove the code duplication via copy-and-paste. For this purpose it is necessary to separate the boilerplate code from the variable code. The boilerplate code can be expressed as a method and the variable piece of code can be passed to this method as the method's argument. This is an idiom where functionality (i.e. the variable piece of code) is passed to a method (i.e. the boilerplate code). The functionality can be conveniently and concisely be expressed by means of lambda expressions.

Let us consider an example for the Execute-Around-Method pattern: use of explicit locks. An explicit `ReentrantLock` (from package `java.util.lock`) must be acquired and released before and after a critical region of statements. The resulting boilerplate code looks like this:

```
class SomeClass {
  private ... some data ...
  private Lock lock = new ReentrantLock();
  ...
  public void someMethod() {
    lock.lock();
    try {
       ... critical region ...
    } finally {
       lock.unlock();
    }
  }
}
```

In all places where we need to acquire and release the lock the same boilerplate code of "lock-try-finally-unlock" appears. Following the Execute-Around-Method pattern we would factor out the boilerplate code into a helper method:

```
class Utilities {
  public static void withLock(Lock lock, CriticalRegion cr) {
```

```
      lock.lock();
      try {
        cr.apply();
      } finally {
        lock.unlock();
      }
    }
  }
```

The helper method `withLock` takes the variable code as a method argument of type `CriticalRegion`:

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

Note that `CriticalRegion` is a functional interface and hence a lambda expression can be used to provide an implementation of the `CriticalRegion` interface. Here is a piece of code that uses the helper method `withLock`:

```
private class MyIntStack {
  private Lock lock = new ReentrantLock();
  private int[] array = new int[16];
  private int sp = -1;

  public void push(int e) {
    withLock(lock, () -> {
      if (++sp >= array.length)
        resize();
      array[sp] = e;
    });
  }

  ...
}
```

The boilerplate code is reduced to invocation of the `withLock` helper method and the critical region is provided as a lambda expression. While the suggested `withLock` method indeed aids elimination of code duplication it is by no means sufficient.

Let us consider another method of our sample class `MyIntStack`: the `pop` method, which returns an element from the stack and throws a `NoSuchElementException` exception if the stack is empty. While the `push` method did neither return nor throw anything the `pop` method has to return a value and throws an exception. Neither is allowed by the `CriticalRegion`'s `apply` method: it has a `void` return type and no `throws`

clause. The exception raised in the critical region of the `pop` method does not cause any serious trouble; it is unchecked and need not be declared in a `throws` clause. The lambda that we need for implementing the `pop` method can throw it anyway. The return type, in contrast, causes trouble. In order to allow for lambda expressions with a return type different from `void` we need an additional `CriticalRegion` interface with an `apply` method that returns a result. This way we end up with two interfaces:

```
@FunctionalInterface
public interface VoidCriticalRegion {
  void apply();
}
@FunctionalInterface
public interface IntCriticalRegion {
  int apply();
}
```

Inevitably, we also need additional helper methods.

```
class Utilities {
  public static void withLock(Lock lock,VoidCriticalRegion cr) {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
  }
  public static int withLock(Lock lock, IntCriticalRegion cr) {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
  }
}
```

Given the additional helper method and functional interface the `pop` method can be implemented like this:

```
private class MyIntStack {

  ...

  public int pop() {
   return withLock(lock, () -> {
      if (sp < 0)
        throw new NoSuchElementException();
      else
        return array[sp--];
    });
  }
}
```

In analogy to the return type you might wonder whether we need additional functional interfaces for critical regions with different argument lists. It turns out that arguments to the critical region are usually not an issue. The lambda expression, which implements the critical region interface, can access (implicitly) `final` variable of the enclosing context. Consider the `push` method again; it takes an argument.

```
private class MyIntStack {
  private Lock lock = new ReentrantLock();
  private int[] array = new int[16];
  private int sp = -1;

  public void push(final int e) {
    withLock(lock, () -> {
      if (++sp >= array.length)
        resize();
      array[sp] = e;
    });
  }

  ...
}
```

The critical region lambda accesses the `push` method's argument, which is either explicitly declared `final` or implicitly treated as such. As long as the critical region only reads the argument and does not modify it there is no need for additional helper methods or functional interfaces.

Wrap-Up

Programming with lambda expressions has two aspects:

- *Using lambdas* to define ad-hoc functionality that is passed to existing operations, such as calling the streams' `forEach`, `filter`, `map`, and `reduce` operations.
- *Designing functional APIs* that take lambda expressions as arguments. At the heart of designing such APIs is the execute-around pattern. Even the streams' `forEach`, `filter`, `map`, and `reduce` operations are example of execute-around: they are loops that execute around the lambda that we pass in.

Default Methods

Lambda expressions and method references are not the only features that have been to the language in release 8 of Java. Java 8 also supports a novel feature named *default methods*. In principle, default methods are entirely unrelated to lambda expressions. It is just that they are the other new language feature in Java 8. Both lambda expressions and default methods are part of the Java Specification Request JSR 335[13] and for this reason we mention them in this tutorial.

Interface Evolution

Default methods are needed for interface evolution. From the previous sections on streams and bulk operation we know that the JDK has been radically overhauled in Java 8. Reengineering such an existing framework often involves the modification of the framework's interfaces. As we all know, modifying an interface breaks all classes that implement the interface. In other words, changing any of the interfaces in the JDK collection framework would break millions of lines of code. This is clearly not a viable option for a reengineering effort of the JDK. Hence the JDK implementers had to figure a means of extending interfaces in a backward compatible way and they invented default methods.

Default methods can be added to an interface without breaking the implementing classes because default methods have an implementation. If every additional method in an interface comes with an implementation then no implementing class is adversely affected. Instead of providing their own implementations of additional methods, the implementing classes can simply inherit the implementations offered by the interface's default methods. An implementing class may choose to override the default implementation suggested by the interface. For this reason, the default methods were initially called *virtual extension methods*; they can be overridden like virtual methods inherited from a superclass.

Let us consider an example. As mentioned earlier, the JDK collections have been extended for Java 8 and one of the changes is addition of a `forEach` method to all collection in the JDK. Hence the JDK designers wanted to add the `forEach` method to the `Iterable` interface, which is the topmost interface of all collections in the JDK.

---

[13] The specification request JSR 335 can be found at http://openjdk.java.net/projects/lambda/.

If this addition is made the traditional way (without default methods), the extended `Iterable` interface looks like this:

```
public interface Iterable<T> {
    public Iterator<T> iterator();
    public void forEach(Consumer<? super T> consumer);
}
```

With this modification, every implementing class does no longer compile because it lacks an implementation of the `forEach` method. The point of a default method is that it supplies the missing implementation so that the implementing classes need not be changed. Using a default method the `Iterable` interface looks like this:

```
public interface Iterable<T> {
    public Iterator<T> iterator();
    public default void forEach(Consumer<? super T> consumer) {
        for (T t : this) {
            consumer.accept(t);
        }
    }
}
```

The obvious difference to a regular method in a class is the `default` modifier. Otherwise, the default method has an implementation pretty much like a regular method in a class.

In addition to the `default` modifier there is another notable difference between regular methods in classes and default methods in interfaces: methods in classes can access and modify not only their method arguments but also the fields of their class. A default method in contrast can only use its arguments because interfaces do not have state. (The fields that you can define in an interface are not really state; they are `static final` fields, i.e. symbolic names for compile-time constant values, which the compiler eliminates during compilation.) All that the implementation of a default method can build on are its own method arguments and the other methods declared in the interface.

Take a look at the default implementation of the `forEach` method above. To illustrates the principle we slightly rewrite it; we replace the for-each loop by explicit use of an iterator. Rewritten the `Iterable` interface looks like this.

```
public interface Iterable<T> {
    public Iterator<T> iterator();
    public default void forEach(Consumer<? super T> consumer) {
        Iterator<T> iter = iterator();
        while (iter.hasNext()) {
            consumer.accept(iter.next());
        }
    }
```

```
}
```

The `forEach` method uses

- its `Consumer` argument, which represents the functionality that is to be applied to each element in the collection, and
- the not yet implemented, abstract `iterator` method that the `Iterable` interface declares.

Essentially, default methods are combinations of the other methods declared in the same interface. multiple inheritance

Multiple Inheritance and Ambiguities

Since classes in Java can implement multiple interfaces and each interface can have default methods, the inherited methods may be in conflict with each other if they have matching signatures. For instance, a class `C` might inherit a method `foo` from both an interface `I1` and an interface `I2`. It raises the question: which method does class `C` inherit?

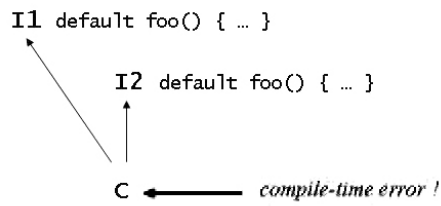

Diagram: Ambiguous Multiple Inheritance - Needs explicit resolution.

In such a situation the compiler cannot resolve the ambiguity and reports an error. In order to enable the programmer to resolve the ambiguity there is syntax for explicitly stating which method class `C` is supposed to inherit. A resolution could look like this:

```
class C implements I1, I2 {
   public void foo() { I1.super.foo(); }
}
```

This is just one example of a conceivable ambiguous inheritance of default methods. There are numerous further examples. In some situation the compiler can resolve the situation because the language has a resolution rule for that situation. In those few cases where the compiler

reports an error (like in the example above) there is syntax for explicit resolution. If you are interested in a more elaborate discussion of multiple inheritance in Java and/or details regarding the resolution of ambiguous multiple inheritance of default methods, please consult the section on "Default Methods" in the *Lambda Reference* document.

Reference to Related Reading

Documentation & Specification

Lambda Expressions - Reference

This tutorial aims to provide a first glance at the language features that were added to the Java programming language in release 8 of Java, namely lambda expression, method reference, and default methods. Comprehensive coverage of the details can be found in the "Lambda Reference".
URL: to be provided

Streams - Tutorial & Reference

Equally interesting is the context for which these features were designed, namely the streams and bulk operations in JDK 8. An overview is given in the "Stream Tutorial", further details in the "Stream Reference".
URL: to be provided

Oracle's Java Tutorial: Section on "Lambda Expressions"

http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

JSR 335 "Project Lambda"

The official OpenJDK project page.
http://openjdk.java.net/projects/lambda/

Brian Goetz on "State of the Lambda", 4th edition, December 2011

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

Brian Goetz on "Translation of Lambda Expressions", April 2012

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html

Conference Presentations

Brian Goetz: The Road to Lambda, JavaOne 2012

https://oracleus.activeevents.com/connect/sessionDetail.ww?SESSION_ID=4862

Brian Goetz: Lambda: A Peek Under the Hood, JavaOne 2012

https://oracleus.activeevents.com/connect/sessionDetail.ww?SESSION_ID=6080

Mike Duigou & Stuart Marks: Jump Starting Lambda Programming, JavaOne 2012

https://oracleus.activeevents.com/connect/sessionDetail.ww?SESSION_ID=5089

Angelika Langer: Lambdas in Java 8, JFokus 2012

http://www.angelikalanger.com/Conferences/Slides/jf12_LambdasInJava8-1.pdf
(video) http://www.jfokus.se/jfokus/video.jsp?v=3072

Angelika Langer: Lambdas in Java 8, JavaZone 2012

(video) http://vimeo.com/49385450

Daniel Smith: Project Lambda in Java SE 8, Strange Loop, September 2012

https://github.com/strangeloop/strangeloop2012/blob/master/slides/sessions/Smith-ProjectLambda(notes).pdf?raw=true
(video) http://www.infoq.com/presentations/Project-Lambda-Java-SE-8

Joe Darcy: On the Road to JDK 8, Devoxx 2012

https://blogs.oracle.com/darcy/resource/Devoxx/Devoxx2012_ProjectLambda.pdf

José Paumard: JDK 8 and lambdas, parallel programming made (too ?) easy, Devoxx 2012

http://www.slideshare.net/slideshow/embed_code/15339485

Maurice Naftalin: Collections After Eight, Devoxx 2012

http://naftalin.org/maurice/professional/talks/cAfter8_devoxx2012.pdf

Tool support

EAP versions of Intellij IDEA

provide amazingly good support for lambda expressions and other parts of the Java 8 feature set.
http://confluence.jetbrains.com/display/IDEADEV/IDEA+12+EAP

Nightly builds of NetBeans 8

provide experimental lambda support.
http://bertram2.netbeans.org:8080/job/jdk8lambda/lastSuccessfulBuild/artifact/nbbuild/

Miscellaneous

Angelika Langer & Klaus Kreft, The Closure Debate, June 2008

An overview of the debate that led to the development of lambda expressions for Java.
http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html for an overview

Mark Reinhold, Closures for Java, November 2009

A blog that announces Project Lambda and explains why it is needed.
https://blogs.oracle.com/mr/entry/closures

Brian Goetz, Interview on Project Lambda, in the *Java Magazine* for September/October 2012

(either register (free) as a subscriber to download the magazine as PDF, or get it via the Newsstand app on iPhone or iPad
http://www.oraclejavamagazine-digital.com/javamagazine/20120910#pg1

Anton Arhipov, Blog at Zero Turnaround on "Java 8: The First Taste of Lambdas", February 2013

A blog entry that explores how lambdas are represented at the runtime and what bytecode instructions are involved during method dispatch. http://zeroturnaround.com/labs/java-8-the-first-taste-of-lambdas/#!/

Appendix

Source Code of Fluent Programming Case Study

Abstractions Used in the Example

```
public interface Person {
    public enum Gender { MALE, FEMALE };
    public int    getAge();
    public String getName();
    public Gender getGender();
}
public interface Employee extends Person {
    public long   getSalary();
}
public interface Manager extends Employee {
}
public interface Department {
    public enum Kind {SALES, DEVELOPMENT,
                      ACCOUNTING, HUMAN_RESOURCES}
    public Department.Kind getKind();
    public String          getName();
    public Manager         getManager();
    public Set<Employee>   getEmployees();
}
public interface Corporation {
    public Set<Department> getDepartments();
}
```

Imperative & Sequential

```
/*
 *  find all managers of all departments with an employee
 *  older than 65
*/
Manager[] find(Corporation c) {
       List<Manager> result = new ArrayList<>();
       for (Department d : c.getDepartments()) {
          for (Employee e : d.getEmployees()) {
             if (e.getAge() > 65) {
              resu  lt.add(d.getManager());
             }
          }
 }
       return result.toArray(new Manager[0]);
}
```

Declarative & Sequential

```
/*
 *   find all managers of all departments with an employee
 *   older than 65
*/
Manager[] find(Corporation c) {
  return
  c.getDepartments().stream()                                    // 1
                    .filter(d -> d.getEmployees().stream()       // 2
                                 .map(Employee::getAge)          // 3
                                 .anyMatch(a -> a>65))           // 4
                    .map(Department::getManager)                 // 5
                    .collect(Collectors.toList())                // 6
                    .toArray(new Manager[0]);                    // 7
}
```

Declarative & Parallel

```
/*
 *   find all managers of all departments with an employee
 *   older than 65
*/
Manager[] find(Corporation c) {
  return
  c.getDepartments().parallelStream()                            // 1
                    .filter(
                      d -> d.getEmployees().parallelStream()     // 2
                    .map(Employee::getAge)                       // 3
                    .anyMatch(a -> a>65))                        // 4
  .map(Department::getManager)                                   // 5
  .collect(Collectors.toList())                                  // 6
  .toArray(new Manager[0]);                                      // 7
}
```

Imperative & Parallel

```
/*
 *   find all managers of all departments with an employee
 *   older than 65
*/
    private static Manager[] find(Corporation c) {
        class FinderTask extends RecursiveTask<List<Manager>> {
            private final Department[] departments;
            private final int from, to;
```

```
        private final int targetBatchSize = 2;

        public FinderTask(Department[] departments,
                          int from, int to)   {
            this.departments = departments;
            this.from = from;
            this.to = to;
        }
        private List<Manager> findSequentially() {
            List<Manager> result = new ArrayList<>();
            for (int i=from;i<to;i++) {
                Department d =  departments[i];
                for (Employee e : d.getEmployees()) {
                    if (e.getAge() > 65) {
                        result.add(d.getManager());
                    }
                }
            }
            return result;
        }
        public List<Manager> compute() {
            if (to-from < targetBatchSize)
                return findSequentially();
            int half = (to-from)/2;
            FinderTask task1
              = new FinderTask(departments,from,from+half);
            FinderTask task2
              = new FinderTask(departments,from+half,to);
            invokeAll(task1,task2);
            try {
                List<Manager> result = task1.get();
                result.addAll(task2.get());
                return result;
            } catch (final InterruptedException |
                          ExecutionException e) {
                throw new RuntimeException(e);
            }
        }
    }
    Department[] departments
       = c.getDepartments().toArray(new Department[0]);
    return ForkJoinPool
          .commonPool()
          .invoke(new FinderTask(departments,0,
                                 departments.length))
          .toArray(new Manager[0]);
}
```

Source Code of Execute-Around-Method Pattern Case Study

Helper Methods and Functional Interfaces

```
public class Utilities {
    @FunctionalInterface
    public static interface VoidCriticalRegion {
        void apply();
    }
    public static void withLock(
      Lock lock,
      VoidCriticalRegion region) {
        lock.lock();
        try {
            region.apply();
        } finally {
            lock.unlock();
        }
    }
    @FunctionalInterface
    public static interface IntCriticalRegion {
        int apply();
    }
    public static int withLock(
      Lock lock,
      IntCriticalRegion region) {
        lock.lock();
        try {
            return region.apply();
        } finally {
            lock.unlock();
        }
    }
    @FunctionalInterface
    public static interface VoidIOECriticalRegion {
        void apply() throws IOException;
    }
    public static void withLockAndIOE(
      Lock lock,
      VoidIOECriticalRegion region)
      throws IOException {
        lock.lock();
        try {
            region.apply();
        } finally {
            lock.unlock();
        }
    }
}
```

IntStack Class

```
public class IntStack {
        private Lock lock = new ReentrantLock();
        private int[] array = new int[16];
        private int sp = -1;

        private void resize() {
            // todo later - for now throw index out of bounds
            array[sp] = 0;
        }
        public void push(int e) {
            withLock(lock, () -> {
                if (++sp >= array.length)
                    resize();
                array[sp] = e;
            });
        }
        public int pop() {
            return withLock(lock, () -> {
                if (sp < 0)
                    throw new NoSuchElementException();
                else
                    return (array[sp--]);
            });
        }
}
```

Index

R

S

T

V