Lambda Expression in Java 8

BTI7311 (Computer Science Seminar), Autumn Term 2015/16, Topic No. 16

Table of Contents

Written by
Strub, Thomas Reto (strut1@bfh.ch)
Touw, Marc (touwm1@bfh.ch)

Evaluated by
Haenni, Prof Dr Rolf (hnr1@bfh.ch)

Handed in
1 November 2015

Presented
by Touw, Marc (touwm1)
5 November 2015

1 Abstract

Lambda expressions or anonymous functions, as they are also known, are extensively used in functional programming languages such as Lisp, Scheme, Clojure and F#. Functional programming languages themselves descend from the mathematical concept of lambda calculus which predates modern computer science by decades. However, lambda expressions are up-to-date like never before. Several procedural programming languages have recently introduced anonymous methods and more specifically, lambda expressions to their syntax to allow developers to use the best from two worlds – functional and procedural programming – in one product.
Oracle scheduled lambda expressions to be included in version 7 of Java SE but due to technical difficulties the introduction had to be deferred to version 8 released in 2014. The introduction of lambda expressions caused a number of proven frameworks to be extended in order to benefit from the new possibilities the useful construct brings along for the language.

Now, more than one year after one of the major improvements in the history of the Java programming language, we want to give both a theoretical and practical introduction into the topic and highlight the advantages lambda expressions have brought by showing a series of short Java source code examples.
This paper was written as part of the course *Computer Science Seminar* at the *Bern University of Applied Sciences* by *Strub, Thomas Reto* (strut1) and *Touw, Marc* (touwm1).

The introduction of lambda expressions also brings along a new framework: Java (Object) Streams have been included in Java 8 and heavily rely on lambda expressions. However, different groups have been assigned to write papers dedicated to the topics of *Streams in Java 8*, *Lisp und Scheme* as well as *Functional Programming*. So these topics will not be discussed in-depth but instead addressed only shortly and left to the other groups to explore.

Readers of this paper are expected to be familiar with the Java programming language and have a basic knowledge of mathematical logic. Nevertheless, certain **emphasised terms** *are explained in the Glossary in chapter 6 on page 13.*

All code examples in this paper are available on GitHub under the following link:
https://github.com/StrubT/CSSemLambdaExpressions.

2 Historical Background

Although the term *anonymous function* is common, especially in theoretical computer science the term *lambda expression* and the like are even more popular. The Greek letter λ emphasises the fact that lambda expressions root in the mathematical concept of *lambda calculus* presented in 1936 by Alonzo Church, an American mathematician. [1] [2]

2.1 Lambda Calculus

The lambda calculus (also written λ-calculus) predates modern computer science and the computer as we know it. The lambda calculus was developed as a purely mathematical concept and only later showed to be a computational model with an expression power equivalent to that of *Turing Machines*, *general recursive functions* and other models.
The lambda calculus consists of only two keywords: λ and . (the baseline dot). Albeit almost trivial, this syntax has been proven tremendously useful in mathematical applications.
The mathematical square function

$$\text{square}: \mathbb{R} \to \mathbb{R}$$

can be written either

$$\text{square}(x) = x^2$$

or anonymously

$$x \mapsto x^2$$

or using lambda calculus

$$\lambda x.\, x^2.$$

The λ signifies there is an anonymous function taking the parameter named x, the . separates the parameter from the function body x^2 calculating the square of the parameter.

Function applications can be written either

$$\text{square}(2) = 2^2 = 4$$

or anonymously

$$(x \mapsto x^2)2 = 2^2 = 4$$

or using lambda calculus

$$(\lambda x.\, x^2)2 = 2^2 = 4$$

The expression $\lambda x.\, x^2$ is applied to the number 2 yielding the result 4.

Using lambda calculus we can write the identity function

$$\lambda x.\, x$$

but a more interesting example is the function

$$\lambda x.\, y.$$

This function, applied to any value for variable x, always returns the value of variable y.
This example is the first one using multiple variables. The variable x (the parameter) is said to be bound by the function, the variable y is said to be free. Whenever the function $\lambda x.\, y$ is applied to a value, x will be bound to this value; while y is still free, it has no defined value yet.

Lambda expressions can be nested, so that

$$\lambda x.\, (\lambda y.\, x + y) = \lambda xy.\, x + y$$

represents an addition of x and y. The second form is simply a convention used when writing nested functions to further shorten lambda expressions.
In this example x is bound by the outer function while y remains free until bound by the inner function. Applying the outer function to the value 1 yields $(\lambda y.\, x + y)[x \coloneqq 1]$. This means the outer function returns the inner function with the captured variable x and the value bound to it by the outer function. This resulting function can be applied to the value 2 resulting in the operation

$$\Big(\big(\lambda x.\, (\lambda y.\, x + y)\big)(1)\Big)(2) = \big((\lambda y.\, x + y)[x \coloneqq 1]\big)(2) = (x + y)[x \coloneqq 1, y \coloneqq 2] = 1 + 2 = 3.$$

2.2 Functional Programming Languages

Programming languages descending from the mathematical concept of lambda calculus are characterised by their excessive use of functions as opposed to mere use of subroutines in *procedural programming languages*. They are therefore known as *functional programming languages*.
Strictly functional languages completely lack of mutable data. A function's return value only depends on the value(s) passed as parameters. This eliminates any side effects from state-dependent behaviour typically present in procedural programming.
Hint: Functional programming is covered by a separate presentation.
Functional programming languages are a great way to develop and test standalone algorithms and have therefore traditionally been used almost exclusively in academic applications. Yet recently, less strict and thus more flexible languages such as Lisp, Scheme, Clojure and F# gain growing importance in commercial applications as well.
Hint: Lisp and Scheme are covered by a separate presentation.
Although functional programming languages descend from the mathematical concept of lambda calculus, this does not mean all of these languages strictly and relentlessly follow it. Many of these languages include named functions alongside anonymous ones as well as modifiable variables, some of them even support immutable or mutable classes.
Having said that, many of the theoretical concepts used in lambda calculus still have a very practical application in functional programming languages. For example, it is imperative to distinguish between bound and free variables. If a function, whether anonymous or named, captures free variables from the surrounding environment, it is also known as a closure. This naming ambiguity leads many to ask whether to call them anonymous functions, lambda expressions or closures when in fact, in most languages they are all of the above. Some programming languages such as C++11 or the syntactically simpler PHP require to explicitly capture the free variables used.
So what exactly is a closure? Just an anonymous function using free variables? Not quite. Just as theoretically described in the previous chapter, a closure is a record containing an anonymous function together with a list of free variables and their captured values from the environment surrounding the function.

2.3 Procedural and Object-oriented Programming Languages

For many years, language architects have been working on incorporating interesting features from other languages into their own. In Java, especially GUI programming has been painful for many years as Java does not have a dedicated functionality to handle events – in this case, events triggered by the user when interacting with the GUI components.
The complexity involved in managing closures may have prevented an earlier introduction of anonymous functions to procedural programming languages. While anonymous functions were introduced to C# with version 2.0 in 2005 [3] and lambda expression with version 3.0 in 2007 [4], it took Java until version 8 released in 2014 [12] to introduce them. Until then, anonymous classes implementing interfaces or extending abstract adapter classes were needed to capture free variables from the environment.
There are, of course, exceptions to this general rule of not including functional aspects in initial releases of procedural or object-oriented programming languages. JavaScript supported anonymous as well as nested functions from the very beginning. They are used in many ways and routinely passed as parameters to other functions to be used as callbacks.

3 Lambda Expressions in Java

As mentioned before, Java did not introduce lambda expressions before version 8 in 2014 [12] while most comparable object-oriented programming languages introduced them earlier on. Lambda expressions were originally scheduled to be included in version 7, but were delayed [11] [13] because of difficulties implementing them. The main problem with lambda expressions in Java is the lack of methods to be first-class citizens: while other languages allow functions or function references to be stored in variables and passed into and returned out of functions, Java methods are not objects on their own. The following chapter outlines how lambda expressions were eventually included in Java.

3.1 Functional Interfaces

Because methods cannot exist alone, the concept of *functional interfaces* was introduced to Java. A functional interface is any interface with exactly one abstract method (except for methods implemented by `java.lang.Object` [10]). Since Java 8 it is also possible for any interface to include *default* or *static methods*. Functional interfaces impose no restrictions on them.

```
@FunctionalInterface
public interface SampleInterface {

 void abstractMethod();                      // one abstract method allowed
 boolean equals(Object obj);                 // defined in Object, allowed

 default void defaultMethod() {              // default methods allowed
  System.out.println("Hello, World! by default function");
 }

 static void staticMethod() {                // static methods allowed
  System.out.println("Hello, World! by static function");
 }
}
```

The annotation `@FunctionalInterface` is recommended but optional. Whether or not an interface is annotated that way does not make any semantic difference. Yet it instructs the compiler to verify the interface contains one and only one abstract method.

But what are these functional interfaces for? Any instance of a class implementing a functional interface can be replaced with a lambda expression. It is no longer necessary to introduce *anonymous classes* implementing them. This new syntax is much more concise.

```
public class SampleClass {

 public static void main(String[] args) {

  myMethod(new SampleInterface() {           // anonymous class (pre-Java 8)
   @Override
   public void abstractMethod() {
    System.out.println("Hello, World! by anonymous class");
   }
  });
                                             // lambda expression (Java 8)
  myMethod(() -> System.out.println("Hello, World! by lambda expression"));
 }

 public static void myMethod(SampleInterface sample) {
  sample.abstractMethod();
 }
}
```

3.2 Lambda Expression Syntax

Java follows the popular pattern and uses the C/C++ *structure dereference* operator also known as *element selection through pointer*. This operator is written as a *minus sign* followed by a *greater than sign* (`->`). An example lambda expression is the square function: `x -> x * x`. In C#, this function would be written as `x => x * x`. C++11, ECMAScript 6 (e.g. next generation JavaScript) and many more use a similar syntax. Python for example uses a more text-based syntax and this function would be written as `lambda x: x * x`.
This example is only a very simple one-line expression. Lambda expressions can get much more complicated but nevertheless, simple expressions are likely the ones most commonly used.

A lambda expression consists of three parts: the parameter list, the separating *arrow token* and either a *single expression* or a *statement block* forming the body. [7]
The parameter list in its completest form looks just like the parameter list of a (named) method.

```
join(elements, (String a, String b) -> a + separator + b);
```

However, most of the time the parameter types can be omitted because the types can be inferred from the *deduction context* [15], somewhat similar to generic types being omitted using the *diamond operator*. This shorthand form of a parameter list looks as follows.

```
join(elements, (a, b) -> a + separator + b);
```

Under certain circumstances, e.g. if the abstract method is generic the parameter types cannot not be inferred. In such a case, the types have to be written for the code to successfully compile.

In the example below, instead of a single expression, a whole statement block with multiple statements forms the expression body.

```
filter(numbers, (Integer n) -> { int mod = n % 2; return mod == 0; });
```

Although this block may even span over several lines, it is generally easier to read if there is only a single expression without braces or a return statement.

```
filter(numbers, (Integer n) -> n % 2 == 0);
```

By omitting the parameter type and eventually the parentheses as well, the original lambda expression can be shortened significantly without negatively influencing the readability. On the contrary, the concise code is even easier to read than the cluttered original.

```
filter(numbers, (n) -> n % 2 == 0);
filter(numbers, n -> n % 2 == 0);
```

The parentheses can only be omitted if there is a single parameter without a type declaration.

Intuitively, if there are no parameters, the parentheses remain empty but cannot be omitted.

```
println(() -> String.format("Hello, World! at %s", LocalDateTime.now()));
```

If the expression body is too complicated to be written inline, it may be a good idea to move it to a separate (named) method. However, one might still prefer not to use an anonymous class. In that case, a different approach can be taken. The same approach can be taken if a lambda expression is only calling a predefined method and optionally returning its output value.
Together with the lambda expression syntax, a second syntax has been added to Java. By using *method references*, it is possible to tell the compiler to call the referenced method in the lambda expression body. [8]

```
List<String> elements = Arrays.asList("Hello", "strut1", "and", "touwm1.");
join(elements, Class::concatStatic);          // static method
join(elements, instance::concatInstance);     // instance method (v1)
join(elements, String::concat);               // instance method (v2)
constructor = Class::new;                     // constructor reference
```

The new double-colon operator (`::`) selects a class' or object's method and represents a lambda expression with the parameter list copied from and a body calling the referenced method. The following method reference is semantically equivalent to the lambda expression below.

```
join(elements, String::concat);              // instance method (v2)
join(elements, (_, str) -> _.concat(str));   // equivalent lambda expression
```

Method references can either reference an *instance method of a particular object*, in which case the implicit instance reference is kept; a static method, in which case the method is simply called with the parameters given; an *instance method of an arbitrary object of a particular type* (as above), in which case the first lambda expression parameter is the instance the method is called on and the remaining parameters are passed on; or a *constructor*, in which case the class constructor is called and the constructed instance is returned. [15]

3.3 Lambda Expression Implementation

Even though anonymous classes were once used in place of lambda expressions and method references, lambda expressions are not just a syntactic sugar but work completely different from anonymous classes. A new instruction had to be added to the *Java Virtual Machine* in order to deal with lambda expressions. [18]
Using anonymous (inner) classes has a number of downsides. As with all *nested classes*, their bytecode is stored in a separate *.class* file. In the example from chapter 3.1 (Functional Interfaces) on page 5, a file named *SampleClass.class* contains the bytecode for the outer class and a second file named *SampleClass$1.class* contains the bytecode for the anonymous class. This separate file then has to be loaded into the application when used, which reduces performance. Furthermore, as with all *inner* (non-static nested) *classes*, they maintain a reference to the instance of the outer class they were instantiated with. And last but not least, a lot of the flexibility towards future changes would have been lost had lambda expressions simply been implemented as anonymous classes.
Instead, a fifth JVM bytecode invocation instruction called `invokedynamic` was added with Java 7 to dynamically invoke methods. [17] The existing four methods are: `invokestatic` for invoking static methods, `invokevirtual` for virtual methods (i.e. overridable instance methods), `invokeinterface` for implemented interface methods and `invokespecial` for special methods such as private methods, overridden virtual methods or constructors and initialisers. The newly added invocation instruction defers the selection of the *translation strategy* [14] until runtime. Depending on whether a lambda expression captures any free variables or not, this strategy may significantly differ. A non-capturing lambda expression may simply be translated into a static method while a lambda expression capturing an instance reference as well as several local variables might be translated into either an instance or a static method with additional parameters for passing the captured variable values along with the explicitly defined lambda expression parameters.

When it comes to capturing free local variables, Java 8 has become much more flexible. Captured function parameters or local variables in nested or *local classes* no longer need to be explicitly declared *final* using the meaningful keyword `final`. Variables assigned to once and only once are now implicitly considered *effectively final* by the compiler. These effectively final variables can now be used just as if they were explicitly declared final.
Fields (a.k.a. instance variables) and static fields (a.k.a. class variables) can be used even if they are not final or effectively final because they can be accessed using a reference to the enclosing class or one of its instances. Methods – static or not – can be accessed similarly. However, it may not be desirable to retain a reference to the instance enclosing the lambda expression in case the lambda expression is expected to be used outside the defining class and to exceed the enclosing instance's lifetime. Accessing static fields and methods is usually not much of a problem as references to the different class types are kept in memory by the *class loader* anyway.

4 Usage

With the introduction of lambda expressions, default methods and other inventions in Java 8, a number of existing classes and interfaces were significantly extended to make use of the new possibilities. Moreover, new classes and interfaces were created to further simplify programming in Java. The following chapter shows some examples of how lambda expression can be used in Java 8.

4.1 Collections

Working with the ever popular *Collections Framework* is much more comfortable than working with legacy arrays yet has always been painful because many operations required repeated iteration over the collection. Several default methods have been added to different interfaces in the framework to further simplify working with collections.
The high-level interface `List` now contains the methods `replaceAll` and `sort`. The following example shows how a list of points scored in an examination are converted into marks using the method `replaceAll` to calculate the marks according to the linear formula used in Switzerland

$$\text{mark} = \frac{\text{points achieved} \cdot 5}{\text{total points}} + 1$$

and the method `sort` to sort the marks in ascending order.

```
List<Double> pointsToMarks = Arrays.asList(6.5, 9.75, 8.5, 3.0, ..., 5.5);
pointsToMarks.replaceAll(p -> p * 5.0 / 10.0 + 1.0);
pointsToMarks.sort(Double::compare);
```

The mid-level interface `Collection` has been extended with the method `removeIf` to allow the developer to remove elements not just by position or known value but also using a *predicate*. The following example removes all elements above 20.

```
Collection<Integer> numbers = Arrays.asList(1, 3, 52, 47, 8, ..., 31, 43, 5);
numbers.removeIf(n -> n > 20);
```

But also the low-level interface `Iterable` was not forgotten: a new method `forEach` has been added allowing a *consumer* to be called on each element as it was previously done with a for-each loop. This method is not expected to replace the for-each loop but to allow for a substitution in simple cases. The following example shows how to print out student logins to the console. As only one statement is needed to achieve that, it is predestined to be turned into a `forEach` call.

```
Iterable<String> students = Arrays.asList("strut1", "touwm1");

for (String student : students)
 System.out.printf("By for loop: %s%n", student);

students.forEach(n -> System.out.printf("By lambda expression: %s%n", n));
```

Numerous methods have been added to the interface `Map` as well. The method `putIfAbsent` adds a value only to the map if the key is not yet present, `merge` merges a present value with a new one while `replaceAll` can be used to completely replace the present values. `getOrDefault` can be used to shorten the conditional access with a default value if the key is not present.

```
Map<String, String> studentNames = new HashMap<>(2);
studentNames.put("touwm1", "Touw, Marc");
studentNames.put("strut1", "Strub, Thomas");
studentNames.putIfAbsent("strut1", "not added");
studentNames.merge("strut1", " Reto", String::concat);
studentNames.replaceAll((k, v) -> String.format("%s (%s)", v, k));
studentNames.forEach((k, v) -> System.out.printf("%s: %s; ", k, v));

System.out.printf("weidj1: %s%n", studentNames.getOrDefault("weidj1", "n/a"));
```

While all of these methods are great additions to the proven interfaces and their implementations, a new framework called *streams* was created "to express sophisticated data processing queries" [16]. Streams allow to create advanced queries querying both collections and arrays and are much more flexible and elaborate when compared to collections.

Hint: Streams are covered by a separate presentation.

The following example shows how to query a list of given names, remove the duplicate names, convert the names to upper case, filter for names starting with the letter H, sort them, implement a *paging*, and create a list containing the result.

```
int pageSize = 3;
int pageIndex = 1;
Collection<String> names = Arrays.asList("Heinrich", "Marc", ..., "Thomas");
Collection<String> namesH = names .stream().parallel()
                                  .distinct()
                                  .map(String::toUpperCase)
                                  .filter(l -> l.startsWith("H"))
                                  .sorted()
                                  .skip(pageIndex * pageSize).limit(pageSize)
                                  .collect(Collectors.toList());
```

Although this example combines many of the possibilities of streams, it is still simple to read and understand because the naming and usage of the methods is self-explanatory. This new framework is something many developers waited years for and has finally been added when functional-style programming had been made possible by the introduction of lambda expressions.

4.2 Concurrency

Although no big changes have been made to the Java *Concurrency API* [5] [9], it can still benefit from the introduction of lambda expressions. Its two most widely used interfaces `Runnable` and `Callable` are both functional interfaces and can therefore be implemented by lambda expressions.

The following example shows three ways how new threads can be created: either by extending the class `Thread`, by anonymously implementing the interface `Runnable` and passing it to a new `Thread` or by using a lambda expression to create a `Runnable` and passing it to a new `Thread`.

```
Thread threadAnonymousExtended = new Thread() {
 @Override // create an anonymous class extending the class
 public void run() {
  System.out.printf("Hello! from %s%n", Thread.currentThread().getName());
 }
};

Thread threadAnonymousImplemented = new Thread(new Runnable() {
 @Override // create an anonymous class implementing the interface
 public void run() {
  System.out.printf("Hello! from %s%n", Thread.currentThread().getName());
 }
});

Thread threadLambda = new Thread( // lambda expression "implementing" interface
 () -> System.out.printf("Hello! from %s%n",
Thread.currentThread().getName()));
```

The lambda expression syntax is the most concise one and certainly to be preferred if there is only a single expression or maybe just a couple simple lines. In case the runnable code is more complex or should be executed multiple times, it may be a better choice to move it to a method and reference it. If the code should be so complex as to justify the creation of a dedicated class possibly including instance members and parameterised constructors, the class should whenever possible implement the `Runnable` interface and only extend the `Thread` class if it is absolutely necessary.

Alongside the simple `Runnable` interface, a second and slightly more powerful interface exists. The `Callable` interface allows the called method to return a value. This return value can then be retrieved and further used by the caller. Callables are most often used with *executor services*. Executor services are basically services managing a pool of threads to which different tasks can be assigned. When done, a `Future` is returned representing the return value being made available sometime in the future.
The following example shows there are multiple implementations of `ExecutorService`. Some contain only a single thread used to sequentially execute all submitted tasks, other managing a pool of several threads using different strategies optimised for different usage scenarios. The example also shows two ways of creating a `Callable`: first by anonymously implementing the interface and then using a lambda expression.
Furthermore, it shows an alternative way to use executor services: it is possible not to pass a `Callable` but a `Runnable` instead. A `Runnable` cannot return a value, so a predefined return value can be specified where appropriate or `null` will be used instead. Although the lambda expressions look the same, the compiler can determine which interface the lambda expression is supposed to implement by comparing the signatures (i.e. the parameter and return types).

```
ExecutorService executor1 = Executors.newSingleThreadExecutor();
ExecutorService executor2 = Executors.newCachedThreadPool();

Future<String> future11 = executor1.submit(new Callable<String>() {
 @Override
 public String call() throws Exception {
  String threadName = Thread.currentThread().getName(); // get name of thread
  System.out.printf("Hello! from future 1 on pool 1 on %s%n", threadName);
  return "return value from future 1 in pool 1"; //return value to caller
 }
});

Future<String> future12 = executor1.submit(() -> {
 String threadName = Thread.currentThread().getName();
 System.out.printf("Hello! from future 2 in pool 1 on %s%n", threadName);
 return "return value from future 2 in pool 1";
});

Future<String> future21 = executor2.submit(() -> {
 String threadName = Thread.currentThread().getName();
 System.out.printf("Hello! from future 1 in pool 2 on %s%n", threadName);
}, "return value from future 2 in pool 2");

Future<?> future22 = executor2.submit(() -> {
 String threadName = Thread.currentThread().getName();
 System.out.printf("Hello! from future 2 in pool 2 on %s%n", threadName);
}); // no return value (null will be returned)
```

4.3 JavaFX

JavaFX is a relatively new framework for developing modern, (F)XML-based desktop applications. It was maintained separate from the Java Standard Edition before its newest version 8 has been incorporated into Java SE 8.
Both *AWT* and *Swing* use `EventListener` interfaces and sometimes abstract *adapter classes* to handle user-initiated events. Some simple interfaces like `ActionListener` and `ChangeListener` are functional interfaces and benefit from lambda expressions, others like `MouseListener` and `WindowListener` are not. If not all of the abstract methods need to be implemented, the abstract classes `MouseAdapter` and `WindowAdapter`, respectively, can be used and only those empty method implementations which are actually needed can be overridden. In JavaFX, a different approach has been taken: there are only a handful of functional listener and handler interfaces which are reused throughout the framework. [6]

To handle the event when a user presses a button, an `EventHandler<ActionEvent>` has to be set to the `onActionProperty` of the `Button`. This can either be done directly on the property itself.

```
this.counterButton.onActionProperty().set(e -> this.counterLabel.setText(
                          String.format("counter value: %d", ++this.counter)));
```

Often however, helper methods exist so as not to violate the *Law of Demeter*.

```
this.counterButton.setOnAction(e -> this.counterLabel.setText(
                          String.format("counter value: %d", ++this.counter)));
```

As mentioned in chapter 4.2 (Concurrency) on page 9, if the handler is more complicated, one might choose to move the code to a separate method. One advantage of a separate method is that the different methods can be kept short but by moving it to a separate method instead of a dedicated anonymous or named class, the structure of the program is kept simple.

```
this.asyncButton.setOnAction(this::asyncButtonHandler);
```

If a lambda expression needs to be used at several different locations in a local scope, it can be stored to a local variable and the variable can then for example be passed on to different methods. This prevents writing the same lambda expression over and over again but keeps it in the same scope instead of moving it to a dedicated method.

```
ChangeListener<? super Number> widthHeightListener = (v, o, n)
      -> this.stageResized(primaryStage.getWidth(), primaryStage.getHeight());
primaryStage.widthProperty().addListener(widthHeightListener);
primaryStage.heightProperty().addListener(widthHeightListener);
```

A usage linked to both JavaFX and concurrency likewise is passing a lambda expression to the method `Platform.runLater(Runnable)`. This invokes the runnable on the *JavaFX Application Thread*. This is necessary because the JavaFX controls are not thread-safe by default. Asynchronous, potentially long-running or even blocking code should be executed in a separate thread and subsequent access to JavaFX controls should be invoked on the JavaFX Application Thread.

```
Platform.runLater(() -> {
 this.asyncLabel.setText(String.format("finished at %s",
                                finished.format(JavaFXClass.TIME_FORMATTER)));
 this.asyncButton.setDisable(false);
});
```

The method `SwingUtilities.invokeLater(Runnable)` provides the same functionality for Swing, `SwingUtilities.invokeAndWait(Runnable)` works similarly but blocks the calling thread until the *AWT event dispatching thread* is available and the code has finished and thus allows the calling code to ensure the invoked code updated the necessary controls before continuing.

5 Conclusion

Lambda expressions are most definitely the biggest single improvement since the introduction of *generics* in Java 5. Although the Java designers decided not to break with Java traditions and thus not to introduce a new class of function types beside the already existing primitive, class, interface, enumerable, array and annotation types; Java developers need to familiarise themselves not so much with the underlying changes to the virtual machine but even more so with the very visible syntactical changes.

Java has always been famous for its diverse and to some extent even divided community. Some Java developers might never use lambda expressions for the pointer history of its arrow token or because they do not have full control over the interface implementation created behind the scenes. But the presented examples should convince almost every one of them of the massive advantages lambda expressions bring along.

But why have lambda expressions not been part of Java from version 1.0? Because Java is an object-oriented language and in fact a strictly class-based one. Lambda expressions, on the other hand, are a feature of traditionally non-object-oriented functional programming languages. The Java language designers had to invest countless hours of evaluating different implementation options, listening to the community for requirements, ideas and the overall feeling towards this new addition.

By listening to the community, fulfilling their requirements and integrating their contributions the designers did a tremendous job in merging the strictly object-based environment with the concept of first-class anonymous functions. Lambda expressions look just like first-class anonymous functions while they are actually boxed in a generated class implementing a predefined interface defining the expression.

This introduction of a much asked-for feature brings Java closer to other competing programming languages and will help the language defend its position amongst the most popularly used programming languages today.

6 Glossary

This glossary defines the terms only in the sense they are used in this document. They may have very different meanings in different contexts.

Term	Explanation
Adapter class	(Abstract) class implementing a listener interface but only providing empty method bodies. Used if only some methods are needed.
Anonymous class	Local class implementing an interface or extending a class defined without a class name. Can only be instantiated at declaration time.
Anonymous function	Function defined without a name. Can be stored in variables.
Arrow token	Sequence of symbols used in lambda expression: `->`.
AWT	*Abstract Window Toolkit*: Framework for graphical user interfaces.
AWT event dispatching thread	Thread used to handle AWT-related events. AWT objects should only be accessed and modified from within this thread.
Class	Base concept of OOP. Named container for fields and methods.
Collections Framework	Includes lists, sets, maps and more advanced collection types.
Concurrency API	Java API for developing concurrent or multi-threaded applications.
Constructor	Specialised method for creating and initialising class instances.
Consumer	Method or object taking in and processing objects without output.
Deduction context	Context of a lambda expression the signature is deduced from.
Default method	Implemented method in an interface. Can be called on any instance.
Diamond operator	Empty pair of inequality signs with inferred generic types: `Class<>`.
Executor service	Thread pool used to execute numerous short code sequences.
Functional / procedural programming language	Types of programming languages: based on interrelated functions and sequences of computational instructions, respectively.
Functional interface	Special name for an interface with only one abstract method.
Instance method of a particular object	Referable instance method of an object. If used in a method reference, the object it is used on is used as its instance reference.
Instance method of an arbitrary object of a particular type	Referable instance method of an object instance. If used in a method reference, the instance reference is implicitly added as an additional method parameter before any other parameters.
Java Virtual Machine	Computing machine, environment and sandbox for Java bytecode.
JavaFX	Modern framework for (F)XML-based graphical user interfaces.
JavaFX Application Thread	Thread used to create and handle JavaFX-related objects.
Law of Demeter	Limiting access to directly known objects: *don't talk to strangers*.
Local class	Named class defined inside method, same scope as local variables.
Method reference	Reference to named method, used instead of anonymous function.
Nested class / static nested class / inner class	Small class nested inside encapsulating class. A static class is called *static nested class*, a non-static class is simply called *inner class*.
Paging	Splitting large collections in pages, showing only one at a time.
Predicate	Function taking arbitrary inputs, always returning Boolean value.
Producer / supplier	Method or object without input returning created objects.
Single expression	Single evaluable expression statement used in lambda expressions.
Statement block	Sequential code block composed of several statements.
Static method	Method belonging to a class, rather than a particular instance.
Stream	Framework for traversing and modifying streams of objects.
Structure dereference / element selection through pointer	Specialised syntax introduced in C/C++ for accessing elements of a referenced structure with implicit dereferencing: the shorthand syntax `p->e` is equivalent to dereference and access `(*p).e`.
Swing	Framework for creating platform-independent graphical interfaces.

7 References

The following official publications were used:

[1] Alama, J., Edward N. Zalta (ed.): The Lambda Calculus, The Stanford Encyclopedia of Philosophy, Spring 2015 Edition, http://plato.stanford.edu/archives/spr2015/entries/lambda-calculus/
[2] Rojas, R.: A Tutorial Introduction to the Lambda Calculus, FU Berlin, WS-97/98, http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
[3] Anonymous Methods (C# Programming Guide), Microsoft Developer Network, 13 October 2015, https://msdn.microsoft.com/en-us/library/0yw3tz5k(v=vs.80).aspx
[4] Hejlsberg, A., Torgersen, M., Overview of C# 3.0, Chapter 26.3 Lambda Expressions, Microsoft Developer Network, March 2007, https://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7
[5] Java™ Platform, Standard Edition 8, API Specification, 27 October 2015, http://docs.oracle.com/javase/8/docs/api/
[6] Java™ Platform, Standard Edition 8, JavaFX API Specification, 27 October 2015, http://docs.oracle.com/javase/8/javafx/api/
[7] Lambda Expressions, Oracle Java Documentation, 19 October 2015, https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[8] Method References, Oracle Java Documentation, 19 October 2015, https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html
[9] Concurrency Utilities Enhancements in Java SE 8, Oracle Java Documentation, 25 October 2015, https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html
[10] FunctionalInterface, Java™ Platform, Standard Edition 8, API Specification, 31 October 2015, https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html
[11] JDK 7 Features, 13 October 2015, http://openjdk.java.net/projects/jdk7/features/
[12] JDK 8 Features, 13 October 2015, http://openjdk.java.net/projects/jdk8/features
[13] Project Lambda, 13 October 2015, http://openjdk.java.net/projects/lambda/
[14] Goetz, B., Translation of Lambda Expressions, April 2012, http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html
[15] Kreft, K., Langer, A., Effective Java - Java 8 - Lambda Expressions & Method References Java Magazine, November 2013, http://www.angelikalanger.com/Articles/EffectiveJava/71.Java8.Lambdas/71.Java8.Lambdas.html
[16] Urma, R.-G., Processing Data with Java SE 8 Streams, Java Magazine, March/April 2014 (part 1) / May/June (part 2), http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html (part 1) / http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html (part 2)
[17] Friesen, J., invokedynamic 101, 16 December 2014, http://www.javaworld.com/article/2860079/scripting-jvm-languages/invokedynamic-101.html
[18] Warburton, R., Urma, R., Fusco, M., Java 8 Lambdas - A Peek Under the Hood, 07 October 2014, http://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood

Some illustrating examples were taken from or inspired by the following Wikipedia pages:

[19] Lambda calculus, 7 October 2015, https://en.wikipedia.org/wiki/Lambda_calculus
[20] Anonymous function, 7 October 2015, https://en.wikipedia.org/wiki/Anonymous_function
[21] Closure (computer programming), 7 October 2015, https://en.wikipedia.org/wiki/Closure_(computer_programming)
[22] Functional programming, 7 October 2015, https://en.wikipedia.org/wiki/Functional_programming
[23] Procedural programming, 7 October 2015, https://en.wikipedia.org/wiki/Procedural_programming
[24] First-class function, 19 October 2015, https://en.wikipedia.org/wiki/First-class_function
[25] Dereference operator, 28 October 2015, https://en.wikipedia.org/wiki/Dereference_operator
[26] Law of Demeter, 28 October 2015, https://en.wikipedia.org/wiki/Law_of_Demeter