

Principles of Programming Languages

<http://www.di.unipi.it/~andrea/Didattica/PLP-14/>

Prof. Andrea Corradini

Department of Computer Science, Pisa

Lesson 30

- Lambdas and streams in Java 8

Java 8: language extensions

Java 8 is the biggest change to Java since the inception of the language. Main new features:

- Lambda expressions
 - Method references
 - Default methods in interfaces
 - Improved type inference
- Stream API

A big challenge was to introduce lambdas without requiring recompilation of existing binaries

Benefits of Lambdas in Java 8

- Enabling functional programming
 - Being able to pass behaviors as well as data to functions
 - Introduction of lazy evaluation with stream processing
- Writing cleaner and more compact code
- Facilitating parallel programming
- Developing more generic, flexible and reusable APIs

Lambda expression syntax:

Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
intSeq.forEach(x -> System.out.println(x));
```

- `x -> System.out.println(x)`

is a lambda expression that defines an *anonymous function (method)* with one parameter named `x` of type `Integer`

```
// equivalent syntax
```

```
intSeq.forEach((Integer x) -> System.out.println(x));
```

```
intSeq.forEach(x -> {System.out.println(x);});
```

```
intSeq.forEach(System.out::println); //method reference
```

- Type of parameter inferred by the compiler if missing

Multiline lambda, local variables, variable capture, no new scope

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
// multiline: curly brackets necessary  
intSeq.forEach(x -> {  
    x += 2;  
    System.out.println(x);  
});  
// local variable declaration  
intSeq.forEach(x -> {  
    int y = x + 2;  
    System.out.println(y);  
});  
// variable capture  
[final] int y = 2; // must be [effectively] final  
intSeq.forEach(x -> {  
    System.out.println(x + y);  
});  
// no new scope!!!  
int x = 0;  
intSeq.forEach(x -> { //error: x already defined  
    System.out.println(x + 2);  
});
```

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {  
    System.out.println(x);  
}
```
- But what type should be generated for this function? How should it be called? What class should it go in?

Functional Interfaces

- Design decision: Java 8 lambdas are instances of *functional interfaces*.
- A **functional interface** is a Java interface with exactly one abstract method. E.g.,

```
public interface Comparator<T> { //java.util
    int compare(T o1, T o2);
}
```

```
public interface Runnable { //java.lang
    void run();
}
```

```
public interface Callable<V> { //java.util.concurrent
    V call() throws Exception;
}
```

Functional interfaces and lambdas

- Functional Interfaces can be used as *target type* of lambda expressions, i.e.
 - As type of variable to which the lambda is assigned
 - As type of formal parameter to which the lambda is passed
- The compiler uses type inference **based on target type**
- Arguments and result types of the lambda must match those of the unique abstract method of the functional interface
- Lambdas can be interpreted as instances of anonymous inner classes implementing the functional interface
- The lambda is invoked by calling the only abstract method of the functional interface

An example: From inner classes...

```
public class Calculator1 { // Pre Java 8
    interface IntegerMath { // (inner) functional interface
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    } // parameter type is functional interface
    // inner class implementing the interface
    static class IntMath$Add implements IntegerMath{
        public int operation(int a, int b){
            return a + b;
        }
    }
    public static void main(String... args) {
        Calculator1 myApp = new Calculator1();
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, new IntMath$Add()));
        // anonymous inner class implementing the interface
        IntegerMath subtraction = new IntegerMath(){
            public int operation(int a, int b){
                return a - b;
            }
        };
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

... to lambda expressions

```
public class Calculator {  
  
    interface IntegerMath { // (inner) functional interface  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    } // parameter type is functional interface  
  
    public static void main(String... args) {  
        Calculator myApp = new Calculator();  
        // lambda assigned to functional interface variables  
        IntegerMath addition = (a, b) -> a + b;  
        System.out.println("40 + 2 = " +  
            myApp.operateBinary(40, 2, addition));  
        // lambda passed to functional interface formal parameter  
        System.out.println("20 - 10 = " +  
            myApp.operateBinary(20, 10, (a, b) -> a - b));  
    }  
}
```

Other examples of lambdas: Runnable

```
public class ThreadTest {// using functional interface Runnable
    public static void main(String[] args) {
        Runnable r1 = new Runnable() { // anonymous inner class
            @Override
            public void run() {
                System.out.println("Old Java Way");
            }
        };

        Runnable r2 = () -> { System.out.println("New Java Way"); };

        new Thread(r1).start();
        new Thread(r2).start();
    }
}
```

```
// constructor of class Thread

public Thread(Runnable target)
```

Other examples of lambdas: Listener

```

JButton button = new JButton("Click Me!");

// pre Java 8
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Handled by anonymous class listener");
    }
});

// Java 8
button.addActionListener(
    e -> System.out.println("Handled by Lambda listener"));

```

New Functional Interfaces in package java.util.function

```
public interface Consumer<T> { //java.util.function
    void accept(T t);
}
public interface Supplier<T> { //java.util.function
    T get();
}
public interface Predicate<T> { //java.util.function
    boolean test(T t);
}
public interface Function <T,R> { //java.util.function
    R apply(T t);
}
```

Other examples of lambdas

```
List<Integer> intSeq = new ArrayList<>(Arrays.asList(1,2,3));  
  
// sort list in descending order using Comparator<Integer>  
intSeq.sort((x,z) -> z - x); // lambda with two arguments  
intSeq.forEach(System.out::println);  
  
// remove odd numbers using a Predicate<Integer>  
intSeq.removeIf(x -> x%2 == 1);  
intSeq.forEach(System.out::println); // prints only '2'
```

```
// default method of Interface List<E>  
default void sort(Comparator<? super E> c)  
// default method of Interface Collection<E>  
default boolean removeIf(Predicate<? super E> filter)  
// default method of Interface Iterable<T>  
default void forEach(Consumer<? super T> action)
```

Default Methods

Adding new abstract methods to an interface breaks existing implementations of the interface

Java 8 allows interface to include

- Abstract (instance) methods, as usual
- **Static methods**
- **Default methods**, defined in terms of other possibly abstract methods

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve *backward compatibility* with existing published interfaces

Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Streams in Java 8

The new **java.util.stream** package provides utilities to support functional-style operations on streams of values. Streams differ from ***collections*** in several ways:

- **No storage**. A stream is not a data structure that stores elements; instead, it conveys elements from a ***source*** (a data structure, an array, a generator function, an I/O channel,...) through a ***pipeline*** of computational operations.
- **Functional in nature**. An operation on a stream produces a result, but does not modify its source.

Streams in Java 8 (cont'd)

- **Laziness-seeking**. Many stream operations, such as *filtering*, *mapping*, or *duplicate removal*, can be implemented lazily, exposing opportunities for optimization. Stream operations are divided into **intermediate** (stream-producing) operations and **terminal** (value- or side-effect-producing) operations. *Intermediate operations are always lazy*.
- **Possibly unbounded**. While collections have a finite size, streams need not. Short-circuiting operations such as *limit(n)* or *findFirst()* can allow computations on infinite streams to complete in finite time.
- **Consumable**. The elements of a stream are only visited once during the life of a stream. Like an *Iterator*, a new stream must be generated to revisit the same elements of the source.

Pipelines

- A typical pipeline contains
 - A source, producing (by need) the elements of the stream
 - Zero or more *intermediate operations*, producing streams
 - A *terminal operation*, producing side-effects or non-stream values
- Example of typical pattern: filter / map / reduce

```
double average = listing // collection of Person
    .stream()             // stream wrapper over a collection
    .filter(p -> p.getGender() == Person.Sex.MALE) // filter
    .mapToInt(Person::getAge) // extracts stream of ages
    .average()            // computes average (reduce/fold)
    .getAsDouble();      // extracts result from OptionalDouble
```

Anatomy of the Stream Pipeline

- A Stream is processed through a pipeline of operations
- A Stream starts with a source
- Intermediate methods are performed on the Stream elements. These methods produce Streams and are not processed until the terminal method is called.
- The Stream is considered consumed when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards
- A Stream pipeline contains some short-circuit methods (which could be intermediate or terminal methods) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

Stream sources

Streams can be obtained in a number of ways. Some examples include:

- From a **Collection** via the **stream()** and **parallelStream()** methods;
- From an array via **Arrays.stream(Object[])**;
- From static factory methods on the stream classes, such as **Stream.of(Object[])**, **IntStream.range(int, int)** or **Stream.iterate(Object, UnaryOperator)**;
- The lines of a file can be obtained from **BufferedReader.lines()**;
- Streams of file paths can be obtained from methods in **Files**;
- Streams of random numbers can be obtained from **Random.ints()**;
- Numerous other stream-bearing methods in the JDK...

Intermediate Operations

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- Several intermediate operations have arguments of *functional interfaces*, thus *lambdas* can be used

```
Stream<T> filter(Predicate<? super T> predicate) // filter
```

```
IntStream mapToInt(ToIntFunction<? super T> mapper) // map f:T -> int
```

```
<R> Stream<R> map(Function<? super T,? extends R> mapper) // map f:T->R
```

```
Stream<T> peek(Consumer<? super T> action) //performs action on elements
```

```
Stream<T> distinct() // remove duplicates - stateful
```

```
Stream<T> sorted() // sort elements of the stream - stateful
```

```
Stream<T> limit(long maxSize) // truncate
```

```
Stream<T> skip(long n) // skips first n elements
```

Terminal Operations

- A **terminal operation** must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.
- Typical: collect values in a data structure, reduce to a value, print or other side effects.

```
void forEach(Consumer<? super T> action)
```

```
Object[] toArray()
```

```
T reduce(T identity, BinaryOperator<T> accumulator) // fold
```

```
Optional<T> reduce(BinaryOperator<T> accumulator) // fold
```

```
Optional<T> min(Comparator<? super T> comparator)
```

```
boolean allMatch(Predicate<? super T> predicate) // short-circuiting
```

```
boolean anyMatch(Predicate<? super T> predicate) // short-circuiting
```

```
Optional<T> findAny() // short-circuiting
```

Infinite Streams

- Streams wrapping collections are finite
- Infinite streams can be generated with:
 - iterate
 - generate

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

```
// Example: summing first 10 elements of an infinite stream
```

```
int sum = Stream.iterate(0, x -> x+1).limit(10).reduce(0, (x, s) -> x+s);
```

```
static <T> Stream<T> generate(Supplier<T> s)
```

```
// Example: printing 10 random numbers
```

```
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>  
mapper)
```


Parallelism & Streams from Collections

- Streams facilitate parallel execution
- Stream operations can execute either in serial (default) or in parallel
- A stream wrapping a collection uses a **Splititerator** over the collection
- Does not provide methods for **returning** elements but
 - For **applying an action** to the next or to all remaining elements
 - For **splitting**: If parallel, the `split()` method creates a new Splititerator and partitions the stream

Critical issues

- Non-interference
 - Behavioural parameters (like lambdas) of stream operations should not affect the source (*non-interfering behaviour*)
 - Risk of `ConcurrentModificationExceptions`, even if in single thread
- Stateless behaviours
 - Stateless behaviour for intermediate operations is encouraged, as it facilitates parallelism, and functional style, thus maintenance
- Parallelism and thread safety
 - For parallel streams, ensuring thread safety is the programmers' responsibility

Monads in Java: Optional and Stream

```
public static <T> Optional<T> of(T value)
// Returns an Optional with the specified present non-null value.

<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
/* If a value is present, apply the provided Optional-bearing mapping
function to it, return that result, otherwise return an empty
Optional. */
```

```
static <T> Stream<T> of(T t)
// Returns a sequential Stream containing a single element.

<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
/* Returns a stream consisting of the results of replacing each element
of this stream with the contents of a mapped stream produced by applying
the provided mapping function to each element. */
```

References

- The Java Tutorials, <http://docs.oracle.com/javase/tutorial/java/index.html>
- Lambda Expressions, <http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
- Adib Saikali, Java 8 Lambda Expressions and Streams, https://www.youtube.com/watch?v=8pDm_kH4YKY
- Brian Goetz, Lambdas in Java: A peek under the hood. <https://www.youtube.com/watch?v=MLksirK9nnE>