Interfaces y Lambda

ESTEBAN VÁZQUEZ

Lambda Expressions

- Now you are ready to learn about lambda expressions, the most exciting change to the Java language in many years.

- You will see how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

Lambda Expressions

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.

- Before getting into the syntax, let's see an example of sorting with a custom comparator.

Lambda Expressions

```
class LengthComparator
implements
Comparator<String> { public
int compare(String first, String
second) {
return first.length() - second.length();
}
}
…
Arrays.sort(strings, new LengthComparator());
```

- The compare method isn't called right away.
  - Instead, the sort method keeps calling the compare method, rearranging the elements if they are out of order, until the array is sorted.

Lambda Expressions

You give the sort method a snippet of code needed to compare elements, and that code is integrated into the rest of the sorting logic, which you'd probably not care to reimplement.

A block of code is passed to someone (a sort method in our example).

That code block was called at some later time.

Lambda Expressions

Up to now, giving someone a block of code hasn’t been easy in Java.

You couldn’t just pass code blocks around.

Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.

In other languages, it is possible to work with blocks of code directly.

Lambda Expressions

+ Better APIs

- In Java, one could have written similar APIs that take objects of classes implementing a particular function, but such APIs would be unpleasant to use.
- For some time now, the question was not whether to augment Java for functional programming, but how to do it.
- It took several years of experimentation before a design emerged that is a good fit for Java.

The Syntax of Lambda Expressions

- Consider again the sorting example from the preceding section.
  - We pass code that checks whether one string is shorter than another.

```
first.length() - second.length()
```

- What are first and second? They are both strings.
  - Java is a strongly typed language, and we must specify that as well:

```
(String first, String second) -> first.length() - second.length()
```

The Syntax of Lambda Expressions

- You have just seen your first lambda expression.
  - Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

```
(String first, String second) -> first.length() - second.length()
```

The Syntax of Lambda Expressions

- Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable.
- He used the Greek letter lambda (λ) to mark parameters.

  λfirst.λsecond.first.length() - second.length()

The Syntax of Lambda Expressions

- You have just seen one form of lambda expressions in Java: parameters, the -> arrow, and an expression

- If the code carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method:
  - Enclosed in { } and with explicit return statements.

The Syntax of Lambda Expressions

```
(String first, String second) -> {
if (first.length() < second.length()) return -1;
else if (first.length() > second.length()) return 1;
else return 0;
}
```

- If a lambda expression has no parameters, you still supply empty parentheses, just as with a parameterless method:

```
() -> {
for (int i = 100; i >= 0; i--)
System.out.println(i);
}
```

The Syntax of Lambda Expressions

- If the parameter types of a lambda expression can be inferred, you can omit them.

```
Comparator<String> comp
    = (first, second) // Same as (String first, String second)
    -> first.length() - second.length();
```

- Here, the compiler can deduce that first and second must be strings because the lambda expression is assigned to a string comparator.

The Syntax of Lambda Expressions

- If a method has a single parameter with inferred type, you can even omit the parentheses:

```
ActionListener listener = event
        ->
        System.out.println("The
        time is " + new Date()");
// Instead of (event) -> ... or (ActionEvent event) -> ...
```

The Syntax of Lambda Expressions

➢You never specify the result type of a lambda expression.

- It is always inferred from context.

```
(String first, String second) -> first.length() - second.length()
```

➢This expression can be used in a context where a result of type int is expected.

- NOTE: It is illegal for a lambda expression to return a value in some branches but not in others. For example this is invalid:

```
(int x) -> { if (x >= 0) return 1; }
```

Functional Interfaces

- As we discussed, there are many existing interfaces in Java that encapsulate blocks of code, such as Comparator.
  - Lambdas are compatible with these interfaces.
- ➢ You can supply a lambda expression whenever an object of an interface with a single abstract method is expected.
  - Such an interface is called a **functional interface**.

Functional Interfaces

- To demonstrate the conversion to a functional interface, consider the Arrays.sort method.
  - Its second parameter requires an instance of Comparator, an interface with a single method. Simply supply a lambda:

```
Arrays.sort( words, (first, second) -> first.length() - second.length() );
```

- Behind the scenes, the Arrays.sort method receives an object of some class that implements Comparator<String>.
  - Invoking the compare method on that object executes the body of the lambda expression.

Functional Interfaces

- The management of these objects and classes is completely implementation dependent, and it can be much more efficient than using traditional inner classes.
- It is best to think of a lambda expression as a function, not an object, and to accept that it can be passed to a functional interface.

Functional Interfaces

- This conversion to interfaces is what makes lambda expressions so compelling. The syntax is short and simple.

```
Timer t = new Timer( 1000, event -> {
System.out.println("At the tone, the time is " + new Date());
Toolkit.getDefaultToolkit().beep();
});
```

- That’s a lot <u>easier to read</u> than the alternative with a class that implements the ActionListener interface.
  - In fact, conversion to a functional interface is the only thing that you can do with a lambda expression in Java.

Functional Interfaces

- The Java API defines a number of very generic functional interfaces in the java.util.function package.

- One of the interfaces, BiFunction<T, U, R>, describes functions with parameter types T and U and return type R.

```
BiFunction<String, String, Integer> comp
        = (first, second) -> first.length() - second.length();
```

Functional Interfaces

However, that does not help you with sorting.

There is no Arrays.sort method that wants a BiFunction.

If you have used a functional programming language before, you may find this curious.

Functional Interfaces

- But for Java programmers, it's pretty natural.

➢ An interface such as Comparator has a specific purpose, not just a method with given parameter and return types.

- When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

Functional Interfaces

- A particularly useful interface in the java.util.function package is Predicate:

```
public
interface
Predicat
e<T> {
boolean
test(T t);
// Additional default and static methods
}
```

- The ArrayList class has a removeIf method whose parameter is a Predicate.

Functional Interfaces

- It is specifically designed to pass a lambda expression.

  - For example, the following statement removes all null values from an array list:

```
list.removeIf(e -> e == null);
```

Method References

- Sometimes, <u>there is already a method</u> that carries out exactly the action that you'd like to pass on to some other code.
  - For example, suppose you simply want to print the event object whenever a timer event occurs.
- Of course, you could call:

```
Timer t = new Timer(1000, event -> System.out.println(event));
```

It would be nicer if you could just pass the println method to the Timer constructor.

Method References

- Here is how you do that:

```
Timer t = new Timer(1000, System.out::println);
```

- The expression System.out::println is a **method reference** that is equivalent to the lambda expression x -> System.out.println(x)

Method References

- As another example, suppose you want to sort strings regardless of letter case.
- You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

- As you can see from these examples, the :: operator separates the method name from the name of an object or class.

Method References

There are three principal cases:

- ❑ object::instanceMethod
- ❑ Class::staticMethod
- ❑ Class::instanceMethod

- In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method.

  - As already mentioned, System.out::println is equivalent to x -> System.out.println(x)

Method References

There are three principal cases:

- ❑ object::instanceMethod
- ❑ Class::staticMethod
- ❑ Class::instanceMethod

- In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method.

o As already mentioned, System.out::println is equivalent to x -> System.out.println(x)

- Similarly, Math::pow is equivalent to (x, y) -> Math.pow(x, y)

Method References

There are three principal cases:

- ❑ object::instanceMethod
- ❑ Class::staticMethod
- ❑ Class::instanceMethod

- In the third case, the first parameter becomes the target of the method. For example:

  String::compareToIgnoreCase is the same as (x, y) -> x.compareToIgnoreCase(y)

Method References

NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean.

- For example, there are two versions of the Math.max method, one for integers and one for double values.

- Which one gets picked depends on the method parameters of the functional interface to which Math::max is converted.

Method References

- You can capture the *this* parameter in a method reference. For example:

  this::equals is the same as x -> this.equals(x)

- It is also valid to use *super*
  - The method expression super::instanceMethod uses this as the target and invokes the superclass version of the given method.

Method References

```
class Greeter {
public void greet() {
System.out.println("Hello, world!");
}
}
class TimedGreeter extends Greeter {
public void greet() {
Timer t = new Timer(1000, super::greet);
t.start();
}
}
```

Superclass

Subclass

- When the TimedGreeter.greet method starts, a Timer is constructed that executes the super::greet method on every timer tick.
  - That method calls the greet method of the superclass

Constructor References

- Constructor references are just like method references, except that the name of the method is new.
  - For example, Person::new is a reference to a Person constructor.
- Which constructor? It depends on the context.
  - Suppose you have a list of strings.

Constructor References

- Then you can turn it into an array of Person objects, by calling the constructor on each of the strings, with the following invocation:

```
ArrayList<String> names = ...;
Stream<Person> stream =
names.stream().map(Person::new);
List<Person> people =
stream.collect(Collectors.toList());
```

The map method calls the Person(String) constructor for each list element.

Constructor References

- If there are multiple Person constructors, the compiler picks the one with a String parameter because it infers from the context that the constructor is called with a string.

- You can also form constructor references with array types.
  - For example, int[]::new is a constructor reference with one parameter: the length of the array.
    - It is equivalent to the lambda expression x -> new int[x]

Variable Scope

- Often, you want to be able to access variables from an enclosing method or class in a lambda expression.

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

- Consider a call:

```
repeatMessage("Hello", 1000); // Prints Hello every 1,000 milliseconds.
```

Variable Scope

- Often, you want to be able to access variables from an enclosing method or class in a lambda expression.

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

- Now look at the variable text inside the lambda expression.
  - Note that this variable is not defined in the lambda expression.

Variable Scope

- Often, you want to be able to access variables from an enclosing method or class in a lambda expression.

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

- Instead, it is a parameter variable of the repeatMessage method.
  - If you think about it, something nonobvious is going on here.

Variable Scope

The code of the lambda expression may run long after the call to repeatMessage has returned and the parameter variables are gone.

How does the text variable stay around?

To understand what is happening, we need to refine our understanding of a lambda expression.

Variable Scope

A lambda expression has three ingredients:

- A block of code
- Parameters
- Values for the **free variables**

➢That is, the variables that are not parameters and not defined inside the code

○In our example, the lambda expression has one free variable, text.

➢The data structure representing the lambda expression must store the values for the free variables, in our case, the string "Hello".

○We say that such values have been *captured by the lambda expression*.

Variable Scope

- It's an implementation detail how that is done.
- For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.

Lambda Expression -> Object with Single Method -> Instance Variables

Here the free variables

- NOTE: The technical term for a block of code together with the values of the free variables is a **closure**.
- If someone gloats that their language has closures, rest assured that Java has them as well.

Variable Scope

In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope.

In Java, to ensure that the captured value is well-defined, <u>there is an important restriction</u>.

In a lambda expression, you can only reference variables whose value doesn’t change.

Variable Scope

```
public static void countDown(int start, int delay) {
ActionListener listener = event -> {
start--; // Error: Can't mutate captured variable
System.out.println(start);
};
new Timer(delay, listener).start();
}
```

- There is a reason for this restriction…
  - Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently.
    - This won’t happen for the kinds of actions that we have seen so far, but in general, it is a serious problem

Variable Scope

- It is also illegal to refer to variable in a lambda expression that is mutated outside.

```
public static void
    repeat(String
    text, int
    count) { for
    (int i = 1; i <=
    count; i++) {
        ActionListener listener = event -> {
            System.out.println(i + ": " + text); // Error:
            Cannot refer to changing i
        };
        new Timer(1000, listener).start();
    }
}
```

Variable Scope

The rule is that any captured variable in a lambda expression must be

effectively final.

An effectively final variable is a variable that is never assigned a new value after it has been initialized.

In our case, text always refers to the same String object, and it is OK to capture it.

However, the value of i is mutated, and therefore i cannot be captured.

Variable Scope

➢The body of a lambda expression has the same scope as a nested block.

- The same rules for name conflicts and shadowing apply.

➢It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp = (first, second) -> first.length() - second.length();
```

- Obviously, inside a lambda expression you can’t have two local variables with the same name neither.

Variable Scope

➢When you use the *this* keyword in a lambda expression, you refer to the *this* parameter of the method that creates the lambda.

```
public class Application() {
    public void init() {
        ActionListener listener = event -> {
            System.out.println(this.toString());
        }
}
```

- The expression this.toString() calls the toString method of the Application object, not the ActionListener instance.

○ There is nothing special about the use of *this* in a lambda expression.

Processing Lambda Expressions

Now let us see how to write methods that can consume lambda expressions!

- The point of using lambdas is **deferred execution**.
- After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda.

Processing Lambda Expressions

There are many reasons for executing code later, such as:

- Running the code in a separate thread
- Running the code multiple times

Running the code at the right point in an algorithm (for example, the comparison operation in sorting)

Running the code when something happens (a button was clicked, data has arrived, and so on)

- Running the code only when necessary

Processing Lambda Expressions

Suppose you want to repeat an action n times.

The action and the count are passed to a repeat method:

- repeat(10, () -> System.out.println("Hello, World!"));

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface.

In this case, we can use the Runnable interface

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
Runnable	none	void	run	Runs an action without arguments or return value	
Supplier<T>	none	T	get	Supplies a value of type T	
Consumer<T>	T	void	accept	Consumes a value of type T	andThen
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U	andThen
Function<T, R>	T	R	apply	A function with argument of type T	compose, andThen, identity

BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two arguments	and, or, negate

Processing Lambda Expressions

```
public static void repeat(int n,
        Runnable action) { for (int i =
        0; i < n; i++)
                action.run();
}
```

- Note that the body of the lambda expression is executed when action.run() is called.

Processing Lambda Expressions

- Now let’s make this example a bit more sophisticated.
- We want to tell the action in which iteration it occurs.
  - For that, we need to pick a functional interface that has a method with an int parameter and a void return.

Processing Lambda Expressions

- The standard interface for processing int values is:

```
public interface IntConsumer {
void accept(int value);
}
```

- Here is the improved version of the repeat method:

```
public static void repeat(int n, IntConsumer action) {
for (int i = 0; i < n; i++)
action.accept(i);
}
```

- And here is how you call it:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

More about Comparators

- The Comparator interface has a number of convenient static methods for creating comparators.
  - These methods are intended to be used with lambda expressions or method references.
- The static comparing method takes a “key extractor” function that maps a type T to a comparable type (such as String).
  - The function is applied to the objects to be compared, and the comparison is then made on the returned keys.

More about Comparators

- For example, suppose you have an array of Person objects.
- Here is how you can sort them by name:

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

- This is certainly much easier than implementing a Comparator by hand.
- Moreover, <u>the code is clearer</u> since it is obvious that we want to compare people by name.

More about Comparators

- You can chain comparators with the thenComparing method for breaking ties. For example:

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

If two people have the same last name, then the second comparator is used.

More about Comparators

There are a few variations of these methods.

- You can specify a comparator to be used for the keys that the comparing and thenComparing methods extract.

For example, here we sort people by the length of their names: .

```
Arrays.sort(people,
    Comparator.comparing(Person::getName, (s, t) -
    > Integer.compare( s.length(), t.length() )
));
```

More about Comparators

- Moreover, both the comparing and thenComparing methods have variants that avoid boxing of int, long, or double values.

  - An easier way of producing the preceding operation would be:

    ```
    Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
    ```

More about Comparators

- If your key function can return null, you will like the nullsFirst and nullsLast adapters.
  - These static methods take an existing comparator and modify it so that it doesn’t throw an exception when encountering null values but ranks them as smaller or larger than regular values.
    - For example, suppose getMiddleName returns a null when a person has no middle name. Then you can use :

      Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))
  - The nullsFirst method needs a comparator—in this case, one that compares two strings.

More about Comparators

- The naturalOrder method makes a comparator for any class implementing Comparable.

  - A Comparator.<String>naturalOrder() is what we need.

Here is the complete call for sorting by potentially null middle names.

I use a static import of java.util.Comparator.*, to make the expression more legible.

- Note that the type for naturalOrder is inferred.

```
Arrays.sort(people, comparing(Person::getMiddleName,
nullsFirst(naturalOrder())));
```

More about Comparators

- The static reverseOrder method gives the reverse of the natural order.

○ To reverse any comparator, use the reversed instance method.

▪ For example, naturalOrder().reversed() is the same as reverseOrder().

naturalOrder().reversed() = reverseOrder()

What is a lambda?

- Term comes from λ-Calculus
  - Formal logic introduced by Alonzo Church in the 1930's
  - Everything is a function!
  - Equivalent in power and expressiveness to Turing Machine
  - Church-Turing Thesis, ~1934
- A lambda (λ) is an *anonymous* function
  - A function without a corresponding identifier (name)

Does Java have lambdas?

- Yes, it's had them since the beginning
- Yes, it's had them since anonymous classes (1.1)
- Yes, it's had them since Java — spec says so
- No, never had 'em, never will

Function objects in Java 1.0

```
class StringLengthComparator implements Comparator {
    private StringLengthComparator() { }
    public static final StringLengthComparator INSTANCE =
            new StringLengthComparator();

    public int compare(Object o1, Object o2) {
        String s1 = (String) o1, s2 = (String) o2;
        return s1.length() - s2.length();
    }
}

Arrays.sort(words, StringLengthComparator.INSTANCE);
```

Function objects in Java 1.1

```
Arrays.sort(words, new Comparator() {
   public int compare(Object o1, Object o2) {
       String s1 = (String) o1, s2 = (String) o2;
       return s1.length() - s2.length();
   }
});
```

Class Instance Creation Expression (CICE)

Function objects in Java 5

```
Arrays.sort(words, new Comparator<String>()
   { public int compare(String s1, String
   s2) {
       return s1.length() - s2.length();
   }
});
```

CICE with generics

Function objects in Java

```
Arrays.sort(words,
    (s1, s2) -> s1.length() - s2.length());
```

- They feel like lambdas, and they're called lambdas
  - But they're no more anonymous than 1.1 CICE's!
  - Method has name, class does not*
  - But method name does not appear in code ☺

No function types in Java, only *functional interfaces*

- Interfaces with only one explicit abstract method
  - AKA *SAM interface* (Single Abstract Method)
- Optionally annotated with `@FunctionalInterface`
  - Do it, for the same reason you use `@Override`
- Some functional interfaces you know
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
  - `java.util.Comparator`
  - `java.awt.event.ActionListener`
  - Many, many more in package `java.util.function`

Function interfaces in `java.util.function`

`BiConsumer<T,U>`
`BiFunction<T,U,R>`
`BinaryOperator<T>`
`BiPredicate<T,U>`
`BooleanSupplier`
`Consumer<T>`
`DoubleBinaryOperator`
`DoubleConsumer`
`DoubleFunction<R>`
`DoublePredicate`
`DoubleSupplier`
`DoubleToIntFunction`
`DoubleToLongFunction`
`DoubleUnaryOperator`
`Function<T,R>`
`IntBinaryOperator`
`IntConsumer`
`IntFunction<R>`
`IntPredicate`
`IntSupplier`
`IntToDoubleFunction`
`IntToLongFunction`

IntUnaryOperator
LongBinaryOperator
LongConsumer LongFunction<R>
LongPredicate LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T> **Supplier<T>**
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>

Lambda Syntax

Syntax	Example
parameter -> expression	x -> x * x
parameter -> block	s -> { System.out.println(s); }
(parameters) -> expression	(x, y) -> Math.sqrt(x*x + y*y)
(parameters) -> block	(s1, s2) -> { System.out.println(s1 + "," + s2); }
(parameter decls) -> expression	(double x, double y) -> Math.sqrt(x*x + y*y)
(parameters decls) -> block	(List<?> list) -> { Arrays.shuffle(list); Arrays.sort(list); }

Method references – a more succinct alternative to lambdas

- An instance method of a particular object (*bound*)
  - `objectRef::methodName`
- An instance method whose receiver is unspecified (*unbound*)
  - `ClassName::instanceMethodName`
  - The resulting function has an extra argument for the receiver
- A static method
  - `ClassName::staticMethodName`
- A constructor
  - `ClassName::new`

Kind	Examples
Bound instance method	System.out::println
Unbound instance method	String::length
Static method	Math::cos
Constructor	LinkedHashSet<String>::new
Array constructor	String[]::new

Method reference examples

Description	Code
Lambda	`s -> Integer.parseInt(s)`
Lambda w/ explicit param type	`(String s) -> Integer.parseInt(s)`
Static method reference	`Interger::parseInt`
Constructor reference	`Integer::new`
Instance method reference	`String::length`

Anonymous class ICE	New Function<String, Integer>(){
	public Integer apply(String s) {
	return s.length();
	}
	}

Some (not all!) ways to get a Function<String,Integer>

What is a stream?

A bunch of data objects, typically from a collection, array, or input device, for bulk data processing

Processed by a *pipeline*

- **A single *stream generator* (data source)**
- **Zero or more *intermediate stream operations***
- **A single *terminal stream operation***

Supports mostly-functional data processing

Enables painless parallelism

- Simply replace stream with parallelStream
- You may or may not see a performance improvement

Stream examples – Iteration

```
// Iteration over a collection
static List<String> stringList = ...;
stringList.stream()
      .forEach(System.out::println);

// Iteration over a range of integers
IntStream.range(0, 10)
      .forEach(System.out::println);

// Puzzler: what does this print?
"Hello world!".chars()
      .forEach(System.out::print);
```

Puzzler solution

```
"Hello world!".chars()
      .forEach(System.out::print);
```

`Prints "721011081081113211911111410810033"`

Why does it do this?

Puzzler Explanation

```
"Hello world!".chars()
      .forEach(System.out::print);
```

Prints `"721011081081113211911111410810033"`

The chars method on `String` returns an `IntStream`

How do you fix it?

```
"Hello world!".chars()
        .forEach(x -> System.out.print((char) x));
```

Now prints "Hello world"

Moral

- Streams only for object ref types, int, long, and double
    - Minor primitive types are missing
- Type inference can be confusing

Stream examples – mapping, filtering

```
List<String> filteredList = stringList.stream()
        .filter(s -> s.length() > 3)
        .collect(Collectors.toList());

List<String> mappedList = stringList.stream()
        .map(s -> s.substring(0,1))
        .collect(Collectors.toList());

List<String> filteredMappedList =
        stringList.stream()
                .filter(s -> s.length() > 4)
                .map(s -> s.substring(0,1))
                .collect(Collectors.toList());
```

Stream examples – duplicates, sorting

```
List<String> dupsRemoved = stringList.stream()
        .map(s -> s.substring(0,1))
        .distinct()
        .collect(Collectors.toList());

List<String> sortedList = stringList.stream()
        .map(s -> s.substring(0,1))
        .sorted()          // Buffers everything until terminal op
        .collect(Collectors.toList());
```

Stream examples – file input

```
// Prints a file, one line at a time
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
        lines.forEach(System.out::println);
}

// Prints sorted list of unique non-empty, lines in file (trimmed)
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
        lines.map(String::trim).filter(s -> !s.isEmpty()).sorted()
                .forEach(System.out::println);
}

// As above, sorted by line length
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
        lines.map(String::trim).filter(s -> !s.isEmpty())
                .sorted(Comparator.comparingInt(String::length))
                .forEach(System.out::println);
}
```

A subtle difference between lambdas and anonymous class instances

```
class Enclosing {
    Supplier<?> lambda() {
        return () -> this;
    }

    Supplier<?> anon() {
        return new Supplier<Object>() {
            public Object get() { return this; }
        };
    }

    public static void main(String[] args) {
        Enclosing enclosing = new Enclosing();
        Object lambdaThis = enclosing.lambda().get();
        Object anonThis = enclosing.anon().get();
        System.out.println(anonThis == enclosing);   // false
        System.out.println(lambdaThis == enclosing); // true
    }
}
```

Stream examples – bulk predicates

```
boolean allStringHaveLengthThree = stringList.stream()
                .allMatch(s -> s.length() == 3);

boolean anyStringHasLengthThree = stringList.stream()
                .anyMatch(s -> s.length() == 3);
```

Streams are processed *lazily*

- Data is “pulled” by terminal operation, not pushed by source
  - Infinite streams are not a problem
- Intermediate operations can be fused
  - Multiple intermediate operations typically don’t result in multiple traversals
- Intermediate results typically not stored
  - But there are exceptions (e.g., sorted)

A simple parallel stream example

- Consider this for-loop (.96 s runtime; dual-core laptop)

```
long sum = 0;
for (long j = 0; j < Integer.MAX_VALUE; j++) sum += j;
```

- Equivalent stream computation (1.5 s)

```
long sum = LongStream.range(0,
                                        Integer.MAX_VAL
UE).sum();
```

- Equivalent parallel computation (.77 s)

```
long sum = LongStream.range(0,Integer.MAX_VALUE)
           .parallel().sum();
```

- Fastest handcrafted parallel code I could write (.48 s)
  – You don't want to see the code. It took hours.

When to use a parallel stream – loosely speaking

- When operations are independent, and
- Either or both:
  - Operations are computationally expensive
  - Operations are applied to many elements of efficiently splittable data structures
- **Always measure before and after parallelizing!**
  - Jackson’s third law of optimization

When to use a parallel stream – in detail

- Consider s.parallelStream().operation(f) if
  - f, the per-element function, is independent
    - i.e., computation for each element doesn't rely on or impact any other
  - s, the source collection, is efficiently splittable
    - Most collections, and java.util.SplittableRandom
    - NOT most I/O-based sources
  - Total time to execute sequential version roughly > 100µs
    - "Multiply N (number of elements) by Q (cost per element of f), guestimating Q as the number of operations or lines of code, and then checking that N*Q is at least 10,000.
      If you're feeling cowardly, add another zero or two." —DL
    - For details: http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

Stream interface is a monster (1/3)

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {
  // Intermediate Operations
  Stream<T> filter(Predicate<T>);
  <R> Stream<R> map(Function<T, R>);
  IntStream mapToInt(ToIntFunction<T>);
  LongStream
  mapToLong(ToLongFunction<T>);
  DoubleStream mapToDouble(ToDoubleFunction<T>);
  <R> Stream<R>    flatMap(Function<T,    Stream<R>>);
  IntStream flatMapToInt(Function<T, IntStream>);  LongStream
  flatMapToLong(Function<T, LongStream>);  DoubleStream
  flatMapToDouble(Function<T, DoubleStream>); Stream<T>
  distinct();
  Stream<T> sorted();
  Stream<T>
  sorted(Comparator<T>);
  Stream<T>
  peek(Consumer<T>);
  Stream<T>
  limit(long);
  Stream<T>
  skip(long);
```

Stream interface is a monster (2/3)

```
// Terminal Operations
void forEach(Consumer<T>);          // Ordered only for sequential streams
void forEachOrdered(Consumer<T>);   // Ordered if encounter order exists
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> arrayAllocator);
T reduce(T, BinaryOperator<T>);
Optional<T> reduce(BinaryOperator<T>);
<U> U reduce(U, BiFunction<U, T, U>, BinaryOperator<U>);
<R, A> R collect(Collector<T, A, R>); // Mutable Reduction Operation
<R> R collect(Supplier<R>, BiConsumer<R, T>, BiConsumer<R, R>);
Optional<T> min(Comparator<T>);
Optional<T> max(Comparator<T>);
long count();
boolean anyMatch(Predicate<T>);
boolean allMatch(Predicate<T>);
boolean noneMatch(Predicate<T>);
Optional<T> findFirst();
Optional<T> findAny();
```

Stream interface is a monster (2/3)

```
    // Static methods: stream sources
    public static <T> Stream.Builder<T> builder();
    public static <T> Stream<T> empty();
    public static <T> Stream<T> of(T);
    public static <T> Stream<T> of(T...);
    public static <T> Stream<T> iterate(T, UnaryOperator<T>);
    public static <T> Stream<T> generate(Supplier<T>);
    public static <T> Stream<T> concat(Stream<T>, Stream<T>);
}
```

In case your eyes aren't glazed yet

```
public interface BaseStream<T, S extends BaseStream<T, S>>
  extends AutoCloseable {
 Iterator<T> iterator();
 Spliterator<T> spliterator();
 boolean isParallel();
 S sequential();  // May have little or no effect
 S parallel();    // May have little or no effect
 S unordered();   // Note asymmetry wrt sequential/parallel
 S onClose(Runnable);
 void close();
}
```

Optional<T> – a third (!) way to indicate the absence of a result

It also acts a bit like a degenerate stream

```
public final class
    Optional<T> { boolean
    isPresent();
    T get();

     void
    ifPresent(Consumer<T>);
    Optional<T>
    filter(Predicate<T>);
    <U> Optional<U> map(Function<T, U>);
    <U> Optional<U> flatMap(Function<T,
    Optional<U>>); T orElse(T);
    T orElseGet(Supplier<T>);
    <X extends Throwable> T orElseThrow(Supplier<X>) throws X;
}
```

Lambda Expressions In JDK

Simplified Parameterised Behaviour

- Old style, anonymous inner classes

```
new Thread(new Runnable {
  public void run() {
    doSomeStuff();
  }
}).start();
```

- New style, using a Lambda expression

```
new Thread(() -> doSomeStuff()).start();
```

Type Inference

- Compiler can often infer parameter types in a lambda expression
  - Inferrence based on target functional interface's method signature

```
static T void sort(List<T> l, Comparator<? super T> c);

List<String> list = getList();
Collections.sort(list, (String x, String y) -> x.length() > y.length());


Collections.sort(list, (x, y) -> x.length() - y.length());
```

- Fully statically typed (no dynamic typing sneaking in)
  - More typing with less typing

Functional Interface Definition

- Is an interface
- Must have only one abstract method
  - In JDK 7 this would mean only one method (like `ActionListener`)
- JDK introduced default methods
  - Adding multiple inheritance of types to Java
  - These are, by definition, not abstract
- JDK also now allows interfaces to have static methods
- `@FunctionalInterface` to have the compiler check

Is This A Functional Interface?

```
@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
```

Yes. There is only one abstract method

Is This A Functional Interface?

```
@FunctionalInterface
public interface Predicate<T> {
  default Predicate<T> and(Predicate<? super T> p) {…};
  default Predicate<T> negate() {…};
  default Predicate<T> or(Predicate<? super T> p) {…};
  static <T> Predicate<T> isEqual(Object target) {…};
  boolean test(T t);
}
```

Yes. There is still only one abstract method

Is This A Functional Interface?

```
@FunctionalInterface
public interface Comparator {
  // Static and default methods elided
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

Therefore only one abstract method

The `equals(Object)` method is implicit from the Object class

Stream Overview

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

Source

```
int total = transactions.stream()
  .filter(t -> t.getBuyer().getCity().equals("London"))
  .mapToInt(Transaction::getPrice)
  .sum();
```

Intermediate operation

Terminal operation

Stream Sources

Many Ways To Create

- From collections and arrays
  - Collection.stream()
  - Collection.parallelStream()
  - Arrays.stream(T array)or Stream.of()
- Static factories
  - IntStream.range()
  - Files.walk()

Stream Terminal Operations

- The pipeline is only evaluated when the terminal operation is called
  - All operations can execute sequentially or in parallel
  - Intermediate operations can be merged
    - Avoiding multiple redundant passes on data
    - Short-circuit operations (e.g. `findFirst`)
    - Lazy evaluation
  - Stream characteristics help identify optimisations
    - `DISTINT` stream passed to `distinct()` is a no-op

Optional Class

- Terminal operations like `min()`, `max()`, etc do not return a direct result
- Suppose the input Stream is empty?
- `Optional<T>`
  - Container for an object reference (null, or real object)
  - Think of it like a Stream of 0 or 1 elements
  - use `get()`, `ifPresent()` and `orElse()` to access the stored reference
  - Can use in more complex ways: `filter()`, `map()`, etc
    - `gpsMaybe.filter(r -> r.lastReading() < 2).ifPresent(GPSData::display);`

LambdaExpressions And Delayed Execution

Performance Impact For Logging

- Heisenberg's uncertainty principle

Always executed

```
logger.finest(getSomeStatusData());
```

- Setting log level to `INFO` still has a performance impact
- Since `Logger` determines whether to log the message the parameter must be evaluated even when not used

Supplier<T>

- Represents a supplier of results
- All relevant logging methods now have a version that takes a Supplier

```
logger.finest(() -> getSomeStatusData());
```

- Pass a description of how to create the log message
  - Not the message
- If the `Logger` doesn't need the value it doesn't invoke the Lambda
- Can be used for other conditional activities

Avoiding Loops In Streams

Functional v. Imperative

- For functional programming you should not modify state
- Java supports closures over values, not closures over variables
- But state is really useful…

Counting Methods That Return Streams

Still Thinking Imperatively

```
Set<String> sourceKeySet =
  streamReturningMethodMap.keySet();

LongAdder sourceCount = new LongAdder();

sourceKeySet.stream()
  .forEach(c -> sourceCount
    .add(streamReturningMethodMap.get(c).size()));
```

Counting Methods That Return Streams

Functional Way

```
sourceKeySet.stream()
  .mapToInt(c -> streamReturningMethodMap.get(c).size())
  .sum();
```

Printing And Counting Functional Interfaces

Still Thinking Imperatively

```
LongAdder newMethodCount = new LongAdder();

functionalParameterMethodMap.get(c).stream(

)
  .forEach(m -> {
    output.println(m);

    if (isNewMethod(c, m))
      newMethodCount.increment();
  });

  return newMethodCount.intValue();
```

Printing And Counting Functional Interfaces

More Functional, But Not Pure Functional

```
int count = functionalParameterMethodMap.get(c).stream()
  .mapToInt(m  ->  {
    int newMethod = 0;
    output.println(m);

    if (isNewMethod(c, m))
      newMethod = 1;

    return newMethod
  })
  .sum();
```

There is still state being modified in the Lambda

Printing And Counting Functional Interfaces

Even More Functional, But Still Not Pure Functional

```
int count = functionalParameterMethodMap.get(nameOfClass)
  .stream()
  .peek(method -> output.println(method))
  .mapToInt(m -> isNewMethod(nameOfClass, m) ? 1 : 0)
  .sum();
```

Strictly speaking printing is a side effect, which is not purely functional

The Art Of Reduction (Or The Need to Think Differently)

A Simple Problem

- Find the length of the longest line in a file
- Hint: BufferedReaderhas a new method, lines(), that returns a Stream

```
BufferedReader reader = ...

reader.lines()
    .mapToInt(String::length)
    .max()
    .getAsInt();
```

Another Simple Problem

- Find the ~~length of the~~ longest line in a file

Naïve Stream Solution

```
String longest = reader.lines().
      sort((x, y) -> y.length() - x.length()). findFirst().
      get();
```

- That works, so job done, right?
- Not really. Big files will take a long time and a lot of resources
- Must be a better approach

External Iteration Solution

```
String longest = "";

while ((String s = reader.readLine()) != null) if (s.length() >
    longest.length())
        longest = s;
```

- Simple, but inherently serial
- Not thread safe due to mutable state

Functional Approach: Recursion

```
String findLongestString(String longest, List<String> l, int i) { if
    (l.get(i).length() > longest.length())
        longest = l.get(i);

    if (i < l.length() - 1)
        longest = findLongestString(longest, l, i + 1);

    if (longest.length() > l.get(i).length())
        return longest;
    return l.get(i);
}
```

Recursion: Solving The Problem

```
List<String> lines = new ArrayList<>();

while ((String s = reader.readLine()) != null) lines.add(s);

String longest = findLongestString("", lines, 0);
```

- No explicit loop, no mutable state, we’re all good now, right?
- Unfortunately not - larger data sets will generate an OOM exception

A Better Stream Solution

- Stream API uses the well known filter-map-reduce pattern
- For this problem we do not need to filter or map, just reduce

Optional<T> reduce(BinaryOperator<T> accumulator)

- BinaryOperatoris a subclass of BiFunction
  - R apply(T t, U u)
- For BinaryOperatorall types are the same
  - T apply(T x, T y)

A Better Stream Solution

- The key is to find the right accumulator
  - The accumulator takes a partial result and the next element, and returns a new partial result
  - In essence it does the same as our recursive solution
  - But without all the stack frames or Listoverhead

A Better Stream Solution

- Use the recursive approach as an acculator for a reduction

```
String longestLine = reader.lines()
    .reduce((x, y) -> {
  if (x.length() > y.length())
    return x;
  return y;
})
.get();
```

A Better Stream Solution

- Use the recursive approach as an acculultor for a reduction

```
String longestLine = reader.lines()
    .reduce((x, y) -> {

  if (x.length() > y.length())
    return x;
  return y;
})
.get();
```

x in effect maintains state for us, by providing the partial result, which is the longest string found so far

The Simplest Stream Solution

- Use a specialised form of max()
- One that takes a Comparator as a parameter

```
reader.lines()
      .max(comparingInt(String::length))
      .get();
```

- comparingInt()is a static method on Comparator

```
Comparator<T> comparingInt(
        ToIntFunction<? extends T> keyExtractor)
```

Lambdas And Streams And JDK 9

Additional APIs

- `Optional` now has a `stream()` method
  - Returns a stream of one element or an empty stream
- `Collectors.flatMapping()`
  - Returns a `Collector` that converts a stream from one type to another by applying a flat mapping function

Additional APIs

Matcher stream support

Stream<MatchResult> results()

Scanner stream support

Stream<MatchResult> findAll(String pattern)

Stream<MatchResult> findAll(Pattern pattern)

Stream<String> tokens()

Additional Stream Sources

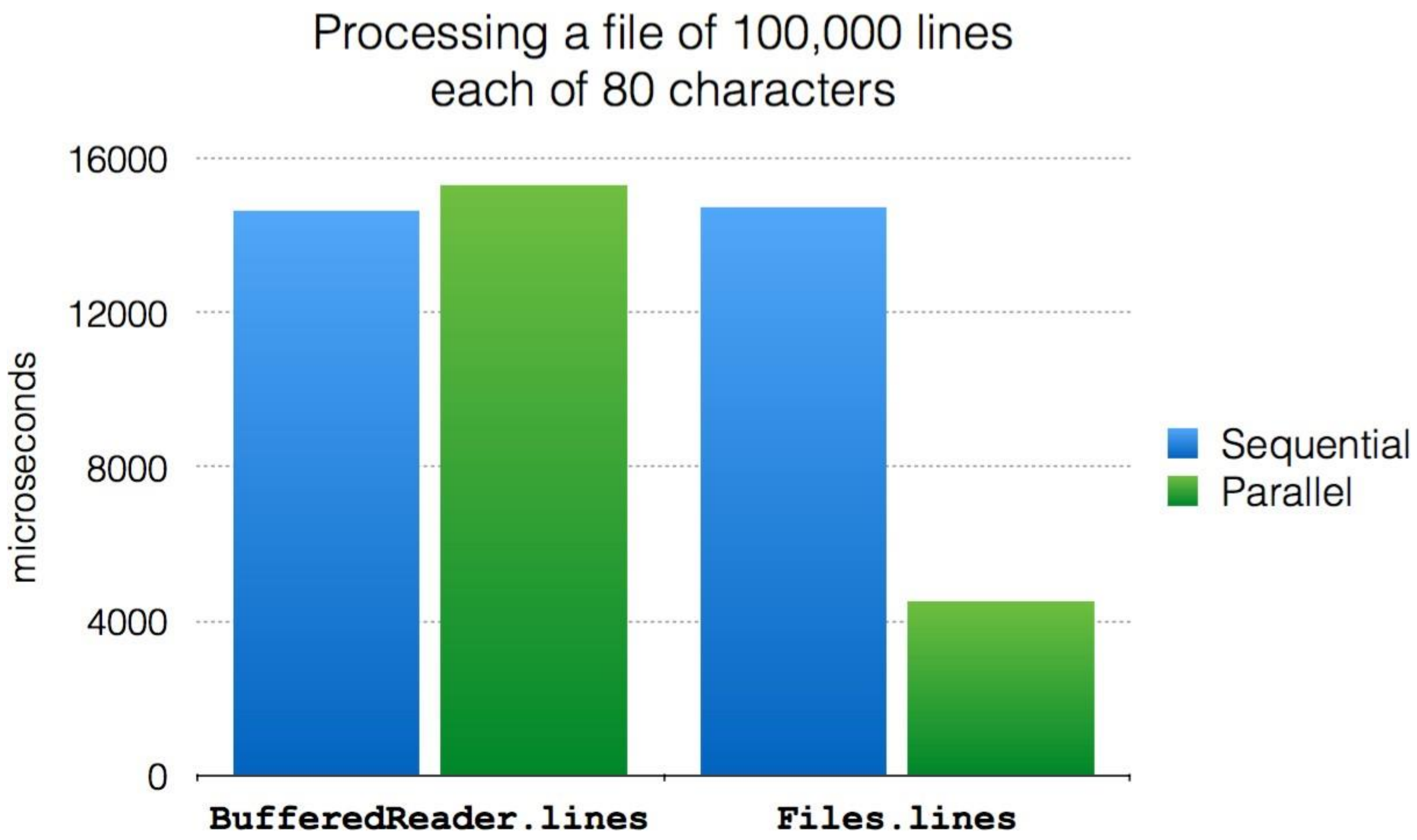
- `java.net.NetworkInterface`
  - `Stream<InetAddress> inetAddresses()`
  - `Stream<NetworkInterface> subInterfaces()`
  - `Stream<NetworkInterface> networkInterfaces()`
    - `static`
- `java.security.PermissionCollection`
  - `Stream<Permission> elementsAsStream()`

Parallel Support For Files.lines()

- Memory map file for UTF-8, ISO 8859-1, US-ASCII
  - Character sets where line feeds easily identifiable
- Efficient splitting of mapped memory region
- Divides approximately in half
  - To nearest line feed

Parallel Lines Performance

133



Results produced using **jmh** on a MacBook Pro (2012 model)

Stream takeWhile

- Stream<T> takeWhile(Predicate<? super T> p)
- Select elements from stream until Predicatematches
- Unordered stream needs consideration

```
thermalReader.lines()
    .mapToInt(i -> Integer.parseInt(i))
    .takeWhile(i -> i < 56)
    .forEach(System.out::println);
```

Stream dropWhile

- Stream<T> dropWhile(Predicate<? super T> p)
- Ignore elements from stream until Predicatematches
- Unordered stream still needs consideration

```
thermalReader.lines()
    .mapToInt(i -> Integer.parseInt(i))
    .dropWhile(i -> i < 56)
    .forEach(System.out::println);
```

Conclusions

Conclusions

- Lambdas provide a simple way to parameterise behaviour
- The Stream API provides a functional style of programming
- Very powerful combination
- Does require developers to think differently
  - Avoid loops, even non-obvious ones!
  - Reductions
- More to come in JDK 9 (and 10)