Functions, Lambda, Streams

minimum & maximum computations

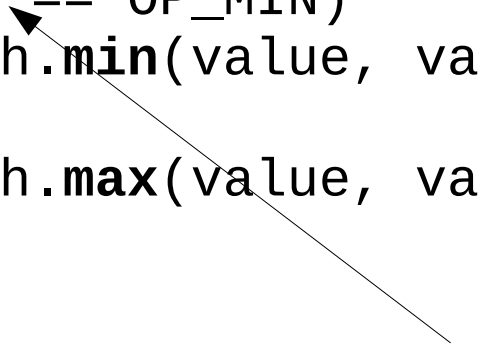
```
public static int min(int[] values) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int min = values[0];
  for(int i = 1; i < values.length; i++) {
    min = Math.min(min, values[i);
  }
  return min;
}
public static int max(int[] values) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int max = values[0];
  for(int i = 1; i < values.length; i++) {
    max = Math.max(max, values[i]);
  }
  return max;
}
```

How to share common code?

We just need a parameter !

```
private static final int OP_MIN = 1;
private static final int OP_MAX = 2;

public static int minOrMax(int[] values, int typeOfOp)
{
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   if (typeOfOp == OP_MIN)
    value = Math.min(value, values[i]);
  } else {
    value = Math.max(value, values[i]);
  }
  return value;
}
```



Ahhhh, TYPE manually defined!!!

Object Oriented approach: reminder

One object for each operation

- 1 object for Math.min and
- 1 object for Math.max

and we pass the object as argument to minOrMax

Then we need 2 classes, and one interface allowing us to commonly handle objects of the two classes

object oriented solution

```
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

public static int minOrMax(int[] values,
                           IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return value;
}
```

Object oriented solution (2)

```
private static final IntBinaryOperator OP_MIN =
  new MinBinaryOperator();


private static final IntBinaryOperator OP_MAX =
  new MaxBinaryOperator();


class MinBinaryOperator implements IntBinaryOperator {
   public int applyAsInt(int left, int right) {
      return Math.min(left, right);
   }
}

class MaxBinaryOperator implements IntBinaryOperator {
   public int applyAsInt(int left, int right) {
      return Math.max(left, right);
   }
}
```

Nice solution but verbose !

Method reference

Java allows an automatic conversion from a method reference to a functional interface

```
IntBinaryOperator op = Math::min;
```

The syntax :: allows to reference a method through its class name followed by its name

- Parameter types are deduced from the functional interface

Functional interface

Conceptually, an interface with a single method is equivalent to a function pointer in C

- We just have name in addition

A functional interface is an interface with only one abstract method

- In our example, `IntBinaryOperator` is a *functional interface*

@FunctionalInterface

Annotation that ask for the compiler to verify that the interface really have only one abstract method

- Works like @Override
- Useful to document

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}
```

How compiler finds the right referenced method ?

IntBinaryOperator op = Math::min;

From the functional interface, compiler finds the abstract method

- int applyAsInt(int left, int right)

It deduces the functional signature

- int(int,int)

Compiler uses the class and the name of the method reference

```
Math.min + int(int,int) = int Math.min(int,int)
```

And what about run time?

Compiler generates some code that ask to create an object of a class implementing the functional interface and whose abstract method implementation invokes the referenced method.

At run time, the virtual machine creates such a class and an instance of it.

(shorter) object oriented solution

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

private static int minOrMax(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return value;
}

public static int min(int[] values) {
  return minOrMax(values, Math::min);
}

public static int max(int[] values) {
  return minOrMax(values, Math::max);
}
```

(shorter) object oriented solution

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

private static int minOrMax(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return value;
}

public static void main(String[] args) {

  int[] values = {4, 5, 2, 8, 4, 9, 3, 7} ;
  System.out.println("min:" + minOrMax(values, Math::min));
  System.out.println("max:" + minOrMax(values, Math::max));
}
```

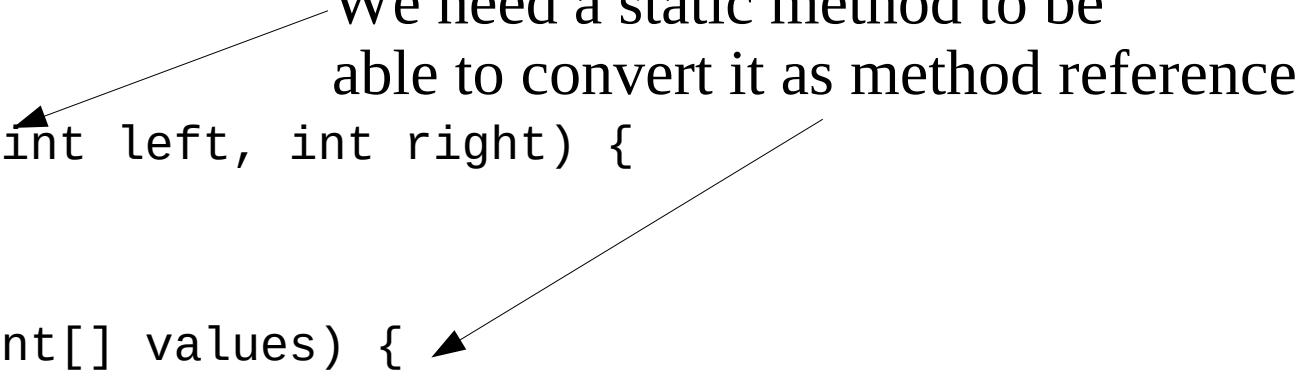
What about computing the sum?

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

private static int reduce(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return value;
}
private static int add(int left, int right) {
  return left + right;
}

public static int sum(int[] values) {
  return reduce(values, EnclosingClass::add);
}
```

We need a static method to be able to convert it as method reference



Lambda

The syntax of lambdas is a shorter syntax to write an anonymous function that will be converted in an object of a class implementing a functional interface

```
public static int sum(int[] values) {
  return reduce( values ,
      (int left, int right) -> left+right);
}
```

A lambda an **anonymous function**

Syntax of lambda

There is two kinds of lambda

- lambda expressions
  - `x -> x + 1`
  - `list.forEach(e -> System.out.println(e));`
- Lambdas blocks
  - ```
    x -> {
      System.out.println("hello lambda");
    }
    ```
  - ```
    list.forEach(e -> {
       map.put(e.getName(), e);
    });
    ```

Type inference of the parameters

Like for method references, it is possible to avoid parameter type specification

```
public static int sum(int[] values) {
  return reduce(values,
                (left, right) -> left + right );
}
```

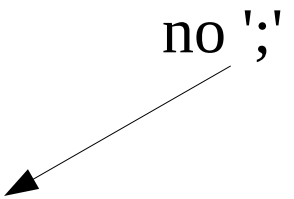
In the same way, compiler finds the functional signature from the functional interface

Syntax of lambdas

Syntax varies according to the number of parameter

Zero parameter

- `() -> System.out.println("hello")`
- `() -> { System.out.println("hello"); }`



One parameter

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

  parenthesis are not mandatory

Two or more parameters

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

And counting the number of ‘3’ ?

```
private static int reduce(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = 0;
  for(int i = 0; i < values.length; i++) {
    value = op.applyAsInt(value, values[i]);
  }
  return value;
}

public static int countNumberOf(int[] values, int value) {
  return reduce(values, (sum, v) -> sum + ((v==value)?1:0));
  /*** or:
  return reduce(values, (sum, v) -> {
    if(v==value)
      return sum+1;
    else
      return sum;
    });
  ***/
}
```

Lambda and local variable

A lambda could use the value of a local variable

```
int countNumberOf(int[] values, int value) {
  return reduce(values,
          (sum,v) - > sum + ((v == value)? 1: 0));
}
```

Since a local variable could die before the code of the lambda is executed, compiler copies its value at the lambda's creation time

Lifetime of local variables

A local variable could be dead when the code of the lambda is executed

```
IntBinaryOperation countByMultiple(int multiple) {
  return ( (sum,v) - > sum + Math.Pow(v,multiple)
);
}

public static void main(String[] args) {
  IntBinaryOperation binOp = countByMultiple(2);
  binOp.applyAsInt(2, 3);
}
```

End of scope

Use of the code of the lambda

Capture of the **value** of a local variable

At the time of conversion into the functional interface, if a lambda is using local variables, their values are saved and transmited as parameters to the lambda

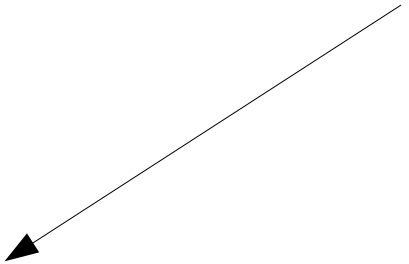
- A lambda that captures value of local variable is distinct at each invocation since car the value could differ
  - Such a lambda cannot be a constant
- Compiler only allows capture of local variables that are assigned only once

Effectively final

It is forbidden to capture the value of a changing variable

Compiler checks the variable is declared final or is effectively final

```
IntBinaryOperation countByMultiple(int multiple) {
  multiple = 7;
  return (v, sum) - > sum + multiple;
}
```



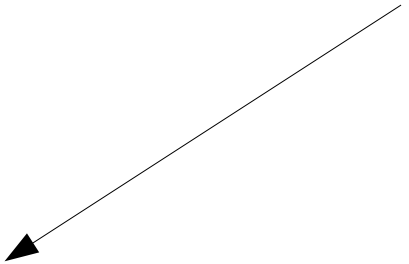
Capture forbidden, because variable is not effectively final

Effectively final

For the same reason, it is not possible to increment or modify the capture of a variable inside the lambda

```
IntBinaryOperation countByMultiple(int multiple) {

  return (v, sum) - > sum + multiple++;
}
```



Capture impossible, variable is not effectively final

Capture of fields

It is also possible to refer to the value of fields…
In this case, the value of “this” is captured

Thus, it is possible to modify the value of the field from the lambda

```
public class Foo {
  private int multiple = 1;

  IntBinaryOperation cumulateBy() {
    return (v, sum) - > sum + multiple++;
    // equivalent à (v, sum) -> sum +
                                this.multiple++;
  }
}
```

Method reference and capture

Method references have also a capture mechanism allowing to capture the value of the reference on which the method will be search

```
ArrayList<String> list = ...
PrintStream out = System.out;
list.forEach(out::println);

list.forEach(System.err::println);
```

Package java.util.function => predefined functional interfaces

- Type of functions
  - from 0 to 2 parameters
    - `Runnable, Supplier, Consumer, Function, BiFunction, ...`
  - Dedicated to primitive types
    - `IntSupplier () → int, LongSupplier () → long, DoubleSupplier () → double`
    - `Predicate<T> (T) → boolean`
    - `IntFunction<T> (int) → T`
    - `ToIntFunction<T> (T) → int`
  - Specialized for same return type as parameter type
    - `UnaryOperator (T) → T ou BinaryOperator (T, T) → T`
    - `DoubleBinaryOperator (double,double) → double`, ...

Runnable and Supplier

java.lang.**Runnable** is equivalent to **() → void**

```
Runnable code = () -> { System.out.println("hello"); }
code.run();
```

Supplier<T> is equivalent to **() → T**

```
Supplier<String> factory = () -> "hello";
System.out.println(factory.get());
```

[Int|Long|Double]Supplier

```
IntSupplier factory = () -> 42;
System.out.println(factory.getAsInt());
```

Consumer

Consumer<T> is equivalent to **(T) → void**

```
Consumer<String> printer =
    s -> System.out.println(s);
printer.accept("hello");
```

[Int|Long|Double]Consumer

```
DoubleConsumer printer =
    d -> System.out.println(d);
printer.accept(42.0);
```

Predicate

Predicate<T> stands for **(T) → boolean**

```
Predicate<String> isSmall = s -> s.length() < 5;
System.out.println(isSmall.test("hello"));
```

[Int|Long|Double]Predicate

```
LongPredicate isPositive = v -> v >= 0;
System.out.println(isPositive.test(42L));
```

Function

Function<T,R> stands for **(T) → R**

```
Function<String, String> fun = s -> "hello " + s;
System.out.println(fun.apply("function"));
```

[Int|Long|Double]Function<R>

```
IntFunction<String[]> arrayCreator =
    size -> new String[size];
System.out.println(arrayCreator.apply(5).length);
```

To[Int|Long|Double]Function<T>

```
ToIntFunction<String> stringLength = s -> s.length();
System.out.println(stringLength.applyAsInt("hello"));
```

UnaryOperator

UnaryOperator<T> stands for **(T) → T**

```
UnaryOperator<String> op = s -> "hello " + s;
System.out.println(op.apply("unary operator"));
```

[Int|Long|Double]UnaryOperator

```
IntUnaryOperator negate = x -> - x;
System.out.println(negate.applyAsInt(7));
```

BiPredicate et BiFunction

BiPredicate<T, U> stands for

(T, U) → boolean

```
BiPredicate<String, String> isPrefix =
    (s, prefix) -> s.startsWith(prefix);
System.out.println(isPrefix.test("hello", "hell"));
```

BiFunction<T, U, R> représente (T, U) → R

- **BiFunction**<String, String, String> concat =
  (s1, s2) -> s1 + " " + s2;
  System.out.println(concat.**apply**("hello", "Bob"));

BinaryOperator

BinaryOperator<T> stands for **(T, T) → T**

```
BinaryOperator<String> concat =
    (s1, s2) -> s1 + " " + s2;
System.out.println(concat.apply("hello", "binop"));
```

[Int|Long|Double]BinaryOperator

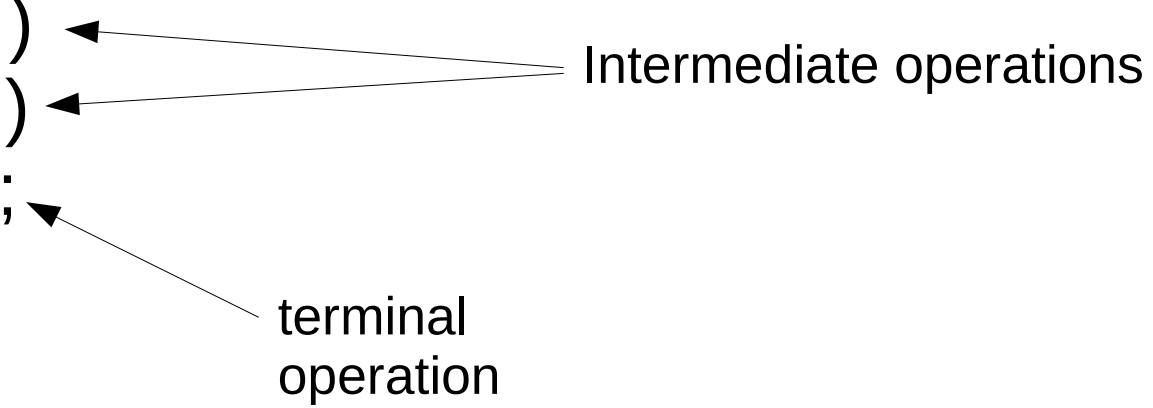
```
IntBinaryOperator add = (a, b) -> a + b;
System.out.println(add.applyAsInt(40, 2));
```

java.util.stream.Stream

API introduced in version 1.8

A Stream does not store data but represents some transformations (intermediate operations) until a terminal operation

```
List<String> list = …
int count = list.stream()
    .map(…)
    .filter(…)
    .count();
```



Example without Stream

Find the list of names of employees older than 50

```
record Employee(String name, int age) { }

List<String> findEmployeeNameOlderThan(
                    List<Employee> employees,
                    int age) {
   var list = new ArrayList<String>();
     for(var employee: employees) {
       if (employee.age() >= age) {
         list.add(employee.name());
       }
     }
   return list;
}
```

Example with Stream

Find the list of names of employees older than 50

```
record Employee(String name, int age) { }

List<String> findEmployeeNameOlderThan(
                    List<Employee> employees,
                    int age) {
    return employees.stream()
            .filter(e -> e.age() >= age)
            .map(Employee::name)
            .collect(Collectors.toList());
}
```

We specify WHAT we want but not HOW to obtain it

General principles of streams

- No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or 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. For example, filtering a `Stream` obtained from a collection produces a new `Stream` without the filtered elements, rather than removing elements from the source collection.
- Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first `String` with three consecutive vowels" need not examine all the input strings. 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.

Intermediate operations

.filter(Predicate<E>): Stream<E>

- Retains only elements for which the predicate is true

.map(Function<E, R>): Stream<R>

- Transforms each element one by one

.flatMap(Function<E, Stream<R>): Stream<R>

- Transforms into 0 to n elements

.skip(long), limit(long)

- forgets/keeps some elements

.distinct(), .sorted(Comparator<E>)

- Without duplicate (cf equals) or sorted (comparator)

Terminal operations

.count(): long

- Number of elements (this is a « reduction »)

.forEach(Consumer<E>): void

- Invoke the consumer on each element

.findAny(), .findFirst(): Optional<E>

- (or Optional.empty())

.toArray(IntFunction<A>): A

- For instance : toArray(String[]::new)

.collect(Collector<...>): …

- Gathers thes elements in a container
  this is a « mutable reduction »

Philosophy

- Parameters of operations must not modify the source (non-interference)
- They must be « stateless »
- It is better if they are side-effect free

  => could pose big problems in case of parallelStream()

- Mutable reductions (reduce/collect) must be preferred rather than side-effect in forEach()

Reduction

A terminal operation that produces a single result value starting from a starting value by processing each element of the stream; more general form:

<U> U reduce(U identity,
BiFunction<U, ? super T, U> accumulator,
BinaryOperator<U> combiner) // for parallelization

for instance, summing characters of a String list:

```
var chars = list.stream()
                .reduce(0,
                        (sum, s) -> sum + s.length(),
                        Integer::sum);
```

Mutable reduction

A reduction that cumulates its result in a mutable container

General form:

```
<R>    R collect(Supplier<R> supplier,
                 BiConsumer<R, ? super T> accumulator,
                 BiConsumer<R, R> combiner);          // for parallelization
```

For instance: creating an ArrayList of String for integers from 0 to 99 :

```
List<String> strings = IntStream.range(0, 99)
    .mapToObj(i -> Integer.toString(i))
    .collect(() -> new ArrayList<>(),                 // supplier
             (c, e) -> c.add(e.toString()),          // accumulator
             (c1, c2) -> c1.addAll(c2));             // combiner
// .collect(Collectors.toList());
// a« Collector » gathers 3: supplier/accumulator/combiner
```

java.util.stream.Collector

Class Collectors defines some predefined Collector

- toList(), toSet(), toUnmodifiableList(), to...
  - stream.collect(toList())
- toCollection(Supplier<Collection<E>>)
  - stream.collect(toCollection(ArrayList::new))
- groupingBy(Function<...> mapper): Map<…, List<...>>
  - Map<String, List<Foo>> map = stream.collect(groupingBy(Foo:name))
- joining(separateur, prefix, suffix): String
  - stream.map(Object::toString()).collect(joining(", ", "[", "]"));

Stream vs SQL

Interface Stream allows some set operations like in SQL

```
SELECT name From persons
  WHERE country = ‘France’ ORDER BY id
```

```
persons.stream()
    .filter(p -> p.country().equals("France"))
    .sorted(Comparator.comparing(Person::id))
    .map(Person::name)
    .collect(toList());
```

Specialization for primitive types

To avoid boxing (allocation), interface Stream is specialized for some primitive types

IntStream, LongStream et DoubleStream

Methode map() is specialized

ex: Stream.mapToInt(Person::age): IntStream

Since types are integers, some methods have been added

- max(), sum(), average(), etc
- IntStream.range(0, 100)

Stream.forEach() vs Collection.forEach()

java.util.Collection already defines a method forEach() (inherited from Iterable)

Instead of

```
collection.stream().forEach(…)
```

it is simpler to write

```
collection.forEach(...)
```