# FUNCTIONAL AND CONCURENT PROGRAMMING

SI4

Pascal URSO

# PROGRAMMING WITH FUNCTIONS

Functional interface, anonymous function, method reference, higher order programming

# FUNCTION AS FIRST CLASS OBJECT

Since Java 8

1. Functional interfaces
2. Interfaces java.util.function
3. Lambda expressions
4. Method references

# FUNCTIONAL INTERFACES

- Interfaces with only one (non-default) method

```
@FunctionalInterface
public interface Foo {
 String method();
 default void defaultMethod() {}
}
```

- Annotation @FunctionalInterface
  - Informative (aka optional)

# INTERFACES JAVA.UTIL.FONCTION

| *Functional interfaces* | *method* | *type* |
|---|---|---|
| • `Function<T,R>` | `apply()` | `T → R` |
| • `BiFunction<T,U,R>` | `apply()` | `T x U → R` |
| • `BinaryOperator<T>` | `apply()` | `T x T → T` |
| • `UnaryOperator<T>` | `apply()` | `T → T` |
| • `Predicate<T>` | `test()` | `T → boolean` |
| • `Consumer<T>` | `accept()` | `T → void` |
| • `Supplier<T>` | `get()` | `void → T` |

- `IntFunction, DoublePredicate, BooleanSupplier, …`

# LAMBDA EXPRESSIONS

- Anonymous function objets, i.e. anonymous fonctional interface instances
- Syntax: `parameter -> expression`
  - Parameter: (), x, (x), (x, y), …
  - Expression: expression or { code block; [return …;] }
- Automatic type resolution
  - `Foo f = () -> “Hello!”;`
  - `Supplier<String> f = () -> “Hello!”;`

# METHOD REFERENCES

- References to existing methods
- Class methods (static), examples:
  - `Integer::max` — `Integer, Integer → Integer`
  - `Collections::emptySet` — `() → Set<>`
  - `String::valueOf` — `xxx → String`
- Instance methods, examples:
  - `Class::method` — adds a first parameter of type Class
    - `String::toUpperCase` — `String → String`
  - `Object::method`
    - `“hello”::toUpperCase` — `() → String`
    - `System.out::println` — `String → ()`

# USAGE EXAMPLE

```
public static <T> void applyAll(T []arr, UnaryOperator<T> f) {
    for (int i = 0; i < arr.length; ++i) {
        arr[i] = f.apply(arr[i]);
    }
}
```

```
Integer[] tab = { 1, 7, -3, 10 };

applyAll(tab, new UnaryOperator<>() {
    public Integer apply(Integer i) { return i * i;} });    // [1, 49, 9, 100]
applyAll(tab, x -> Integer.max(x, b: 5));                   // [5, 7, 5, 10]
applyAll(tab, java.lang.Math::abs);                         // [1, 7, 3, 10]
```

*Without side effect ;)*

# HIGHER-ORDER PROGRAMMING

- Higher-order functions:
  - Either takes a function as argument
  - Or returns a function
- Apply functions on sequences of elements
  - Map
  - Filter
  - Reduce
  - …

# BASIC EXAMPLES

```
static <T> void saysYesOrNo(T e, Predicate<T> f) {
    if (f.test(e)) {
        System.out.println("Yes");
    } else {
        System.out.println("No");
    }
}

static Predicate<String> sameLetters(String x) {
    return y -> y.toLowerCase().equals(x.toLowerCase());
}
```

# FUNCTION COMPOSITION

- Function<T,R>::andThen

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
```

- Function<T,R>::compose

```
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
```

- Examples (*begins with "hello", not considering capitalisation*):

```
Function<String, String> f = String::toLowerCase;
Function<String, Boolean> g = f.andThen(s -> s.startsWith("hello"));

Function<String, Boolean> gg = s -> s.startsWith("hello");
Function<String, Boolean> ff = gg.compose(String::toLowerCase);
```

# MAP<E, F>

- Apply a unary function to each element of a sequence
- **(E → F) × Seq<E> → Seq<F>**
- *Returns a new sequence, respect the order*
- Example:
  - `map(String::toLowerCase, toLST("A","b”,"C")) // => ["a","b","c"]`
- Predefined only for stream in Java (see later)
  - Contrary to python, C++, JavaScript, PHP, all functional languages

# FILTER<E>

- Retains elements that satisfy a predicate
- **(E → boolean) × Seq<E> → Seq<E>**
- *Returns a new sequence, respect the order*
- Example
  - `filter(i -> i > 5, toLST(10, 3, -1, 6)) // => [10, 6]`
- Predefined only for stream in Java (see later)
  - Contrary to python, C++, JavaScript, PHP, all functional languages

# REDUCE<T, R>

- Combine the elements of the sequence together
- **(R × T → R) × Seq<T> × R → R**
- **reduce(f, [n0, n1, …, nN], e) => f(...f(f(f(e, n0), n1), ….), nN)**
- Examples
  - `reduce(Sting::concat, toLST("A", "BB", "CCC"), "") // => "ABBCCC"`
  - `reduce(a,b -> a+b.length(), toLST("Hi", "Ciao", "Salut"), 0) // => 11`
- Predefined only for stream in Java (see later)
  - Contrary to python, C++, JavaScript, PHP, all functional languages
  - Often called Fold / FoldLeft / FoldRight

# COMBINE HIGHER-ORDER FUNCTIONS

```
Lst<Integer> sizes =
        map(String::length, toLST(”Hi", ”Ciao", ”Salut”)
int sumSize = reduce(Math::sum, sizes, 0);
```

- Or less explicitly

```
int sumSize = reduce(Math::sum, map(String::length,
toLST(”Hi", ”Ciao", ”Salut”), 0);
```