EXPLORE NEW PERSPECTIVES

Parallel and concurrent programming in Java 8

Part II - lambda functions

Lambda functions

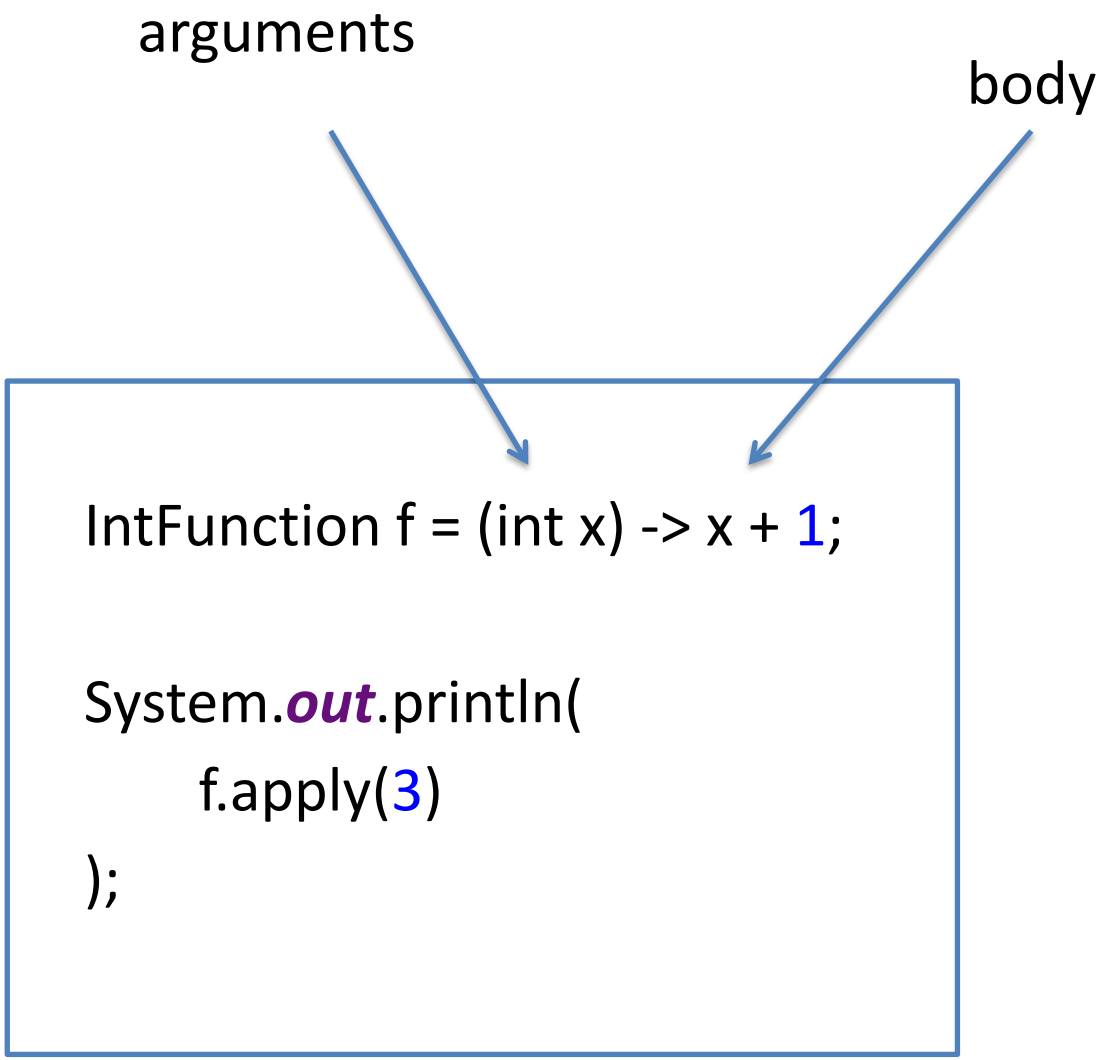
represents a functional interface

implements behavior parametrization

lambda functions

provides lazy evaluation

A first example



Is really an interface?

```
IntFunction g = new IntFunction() {
    @Override
    public Object apply(int x) {
        return x + 1;
    }
};

System.out.println(
        g.apply(3)
);
```

yes!

Are there other interfaces?

yes, many!

```
IntToDoubleFunction h = (int x) -> x * 3.1415;

System.out.println(
        h.applyAsDouble(2)
);
```

Interface definition

Note that there is a generic type in the interface definition!

```
IntFunction<String> m = (int x) -> "OK:" + x;
System.out.println(
        m.apply(3)
);
```

Interface definition

Can we define our own interface?
Yes!

```
interface StringFunction<R> {
    R apply(String value);
};


com.esteco.StringFunction<Integer> o = (String x) -> x.length();
System.out.println(
        o.apply("Hello")
);
```

Simplifications

1. Parameter types can be omitted (all or none)
2. a single parameter does not require parenthesis

```
IntFunction f = x -> x + 1;
IntToDoubleFunction h = x -> x * 3.1415;
com.esteco.StringFunction<Integer> o = x -> x.length();
```

Other interfaces

Is there any general function declaration? Yes!

```
Function<Integer, String> p = x -> ":" + x + ":";
System.out.println(
        p.apply(3)
);
```

Note that there are other method definitions!
compose(), antThen()...

Parameters

Can we get use more than one parameter? Yes, of course

```
interface IntIntFunction<R> {
    R apply(Integer x, Integer y);
}
com.esteco.IntIntFunction q = (x, y) -> x + y;
System.out.println(
        q.apply(2, 3


        )
);
```

Examples

Let's do it also for doubles:

```
interface DoubleDoubleFunction<R> {
    R apply(Double x, Double y);
    //R compare(Double x, Double y);
}
com.esteco.DoubleDoubleFunction r = (x, y) -> x + y;
System.out.println(
        r.apply(3.14, 0.0015)
);
```

Context dependent!

The following two lambda expressions are the same:

```
com.esteco.IntIntFunction q = (x, y) -> x + y;

com.esteco.DoubleDoubleFunction r = (x, y) -> x + y;
```

Note that the type of the lambda expression depends on the context!

Anonymous classes

Lambdas can help when using anonymous classes

```
JButton jb = new JButton();
jb.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hi");
    }
});
```

can be written as:

```
jb.addActionListener(e -> System.out.println("Hi"));
```

Anonymous classes

```
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hi");
    }
});
t1.start();
```

can be written as:

```
Thread t2 = new Thread(() -> System.out.println("hi"));
t2.start();
```

Anonymous classes

anonymous classes
create a new object

variable capture
is different

but there are some differences!

etc.

Functional interfaces

Interface with exactly one abstract method

```
@FunctionalInterface
interface StringFunction<R> {
    R apply(String value);
};
@FunctionalInterface
interface IntIntFunction<R> {
    R apply(Integer x, Integer y);
}
@FunctionalInterface
interface DoubleDoubleFunction<R> {
    R apply(Double x, Double y);
}
```

Functional interfaces

Predicate

BiFunction

BiPredicate

Function

many predefined

Consumer

BinaryOperator

BiConsumer

UnaryOperator

Supplier

Functional interfaces

IntFunction

DoubleFunction

LongFunction

ToLongFunction

many specialized

ToIntFunction

ToDoubleFunction

The Function functional interface

What do we have inside Function?

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
  default <V> Function<V, R> compose(...) { ... }
  default <V> Function<T, V> andThen(...) { ... }
static <T> Function<T, T> identity() { ... }
}
```

But... we can have a single abstract method! OK... there are default methods... we will explore them later

Other methods

we can use them as in FP

```
Function<Integer, Integer> w1 = x -> x * x;
Function<Integer, Integer> w2 = x -> x + x;
System.out.println(
        w1.andThen(w2).apply(2)
);
System.out.println(
        w1.compose(w2).apply(2)
);
System.out.println(
        w1.compose(w1).compose(w2).andThen(w2).apply(2)
);
```

Other methods

```
System.out.println(
        Function.identity().apply(2)
);

System.out.println(
        ((IntFunction)(x -> x * x)).apply(2)
);
System.out.println(
        ((Function<Integer, Integer>)(x -> x * x)).apply(2)
);
```

Type information

Sometimes, type information has to be provided!

```
(x -> x*x).apply(2)   // wrong!

((Function<Integer, Integer>)(x -> x * x)).apply(2)   // OK
```

Predicate examples

```
Predicate<Integer> greaterThanZero = x -> x > 0;
Predicate<Integer> smallerOrEqualThanZero =
greaterThanZero.negate();
Predicate<Integer> smallerThanFive = x -> x < 5;
Predicate<Integer> betweenZeroAndFive =
greaterThanZero.and(smallerThanFive);
Predicate<Integer> notBetweenZeroAndFive =
betweenZeroAndFive.negate();

System.out.println(
        notBetweenZeroAndFive.test(6)
);
```

Method references

```
Function<String, Integer> lg1 = x -> x.length();
Function<String, Integer> lg2 = String::length;

System.out.println(
        lg1.apply("Hello") + lg2.apply("Hi")
);
```

Method references

Can be applied to reference static and instance methods, and also to reference constructors

```
Function<String, Integer> len1 = s -> s.length();
Function<String, Integer> len2 = String::length;

BiPredicate<String, String> pred1 = (s1, s2) -> s1.equals(s2);
BiPredicate<String, String> pred2 = String::equals;

Supplier<ArrayList> c1 = () -> new ArrayList();
Supplier<ArrayList> c2 = ArrayList::new;
```

Other examples

```
static void doSomething(String s, Predicate<String> p,
                        Function<String, String> f) {
if (p.test(s))
   System.out.println(f.apply(s));
}
doSomething("Numeric", x -> x.contains("m"),
Function.<String>identity());
doSomething("Numeric", x -> x.contains("m"),
String::toLowerCase);
doSomething("Numeric", x -> x.contains("m"), x -> "yes");
doSomething("Numeric", x -> x.length() < 5, x -> "too small");
doSomething("", String::isEmpty, x -> "empty string");
```

Variable capture

this works:

```
int a = 1;
IntFunction w = x -> x + a + 1;
System.out.println(w.apply(3));
```

this does not:

```
int a = 1;
IntFunction w = x -> x + a + 1;
a++;
System.out.println(w.apply(3));
```

Example: a comparator

```
List<String> arr = Arrays.asList("Mariapia", "Teresa", "Stefano");
Collections.sort(arr, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
});
```

```
Collections.sort(arr, (o1, o2) -> o1.length() - o2.length());
Collections.sort(arr, String::compareToIgnoreCase);
```

```
System.out.println( arr.stream().collect(Collectors.joining(", ")) );
```


ESTECO
EXPLORE NEW PERSPECTIVES
Thank you
for your attention!
www.esteco.com