



Programming in Java

Part V – lambda functions



Carlos Kavka

Head of Research and Development

Agenda



Lambda functions

Functional interfaces

Method references

Variable capture

Lambda functions

represents a
functional interface

implements
behavior
parametrization

lambda
functions

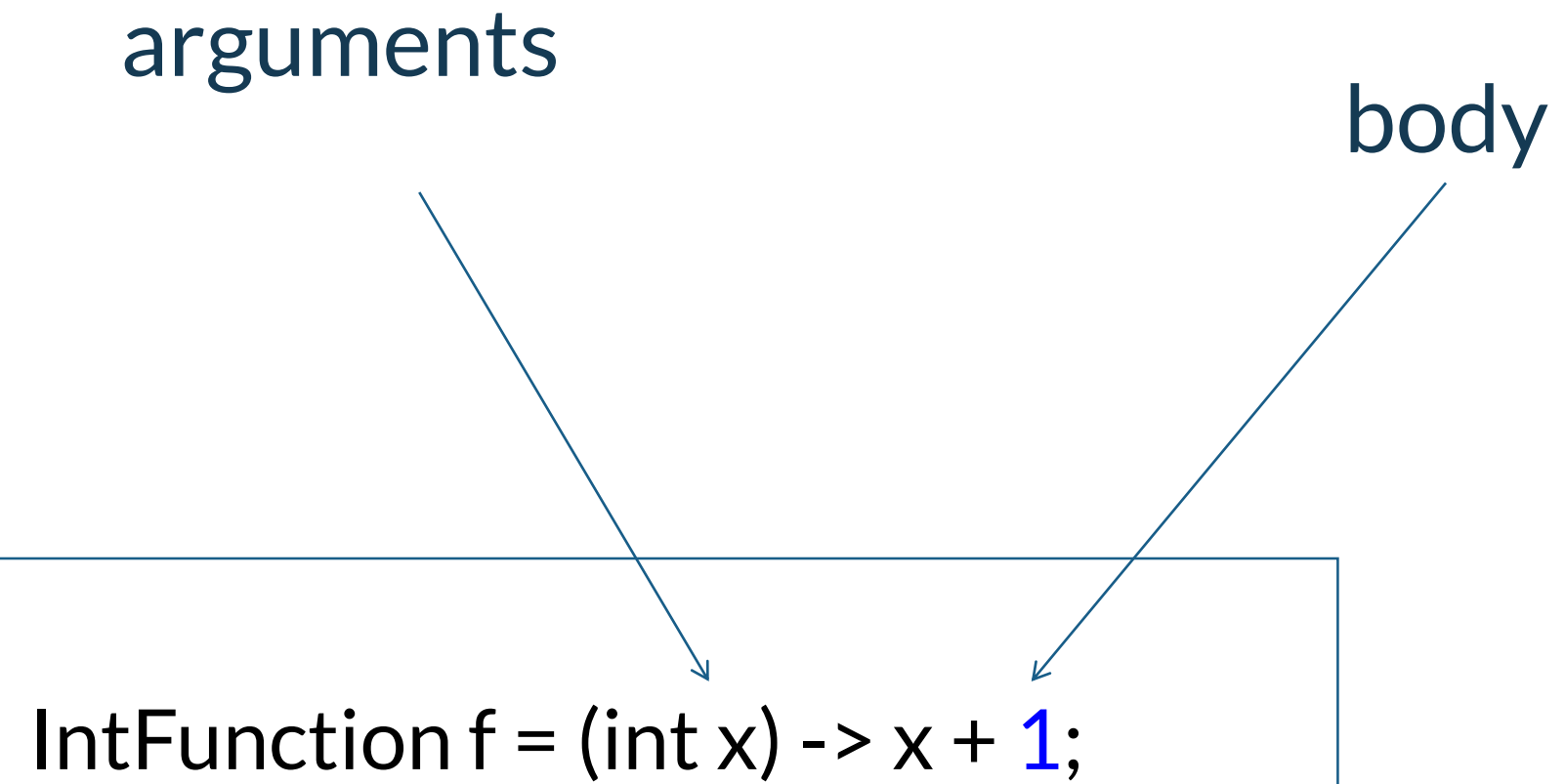
provides lazy
evaluation



A first example

arguments

body



```
IntFunction f = (int x) -> x + 1;
```

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



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?

```
package com.esteco;
...
@FunctionalInterface
interface StringFunction<R> {
    R apply(String value);
};
...
com.esteco.StringFunction<Integer> o = (String x) -> x.length();
System.out.println(o.apply("Hello"));
```

Yes!



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?

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

Yes!

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



Parameters

Can we use **more than one** parameter?

```
interface IntIntFunction<R> {  
    R apply(Integer x, Integer y);  
}
```

```
com.esteco.IntIntFunction q = (x, y) -> x + y;  
System.out.println(  
    q.apply(2, 3)  
);
```

Yes, of course



Examples

Let's do it also for doubles

```
interface DoubleDoubleFunction<R> {  
    R apply(Double x, Double y);  
}  
  
com.esteco.DoubleDoubleFunction<Double> 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<Integer> q = (x, y) -> x + y;
```

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

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



Anonymous classes

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

Lambdas can help when using **anonymous** classes

can be written as:

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



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

anonymous classes create a new object

but there are
some
differences!

for example, variable
capture is different

etc.



Functional interfaces

Interfaces with exactly
one abstract method

```
@FunctionalInterface
```

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

```
@FunctionalInterface
```

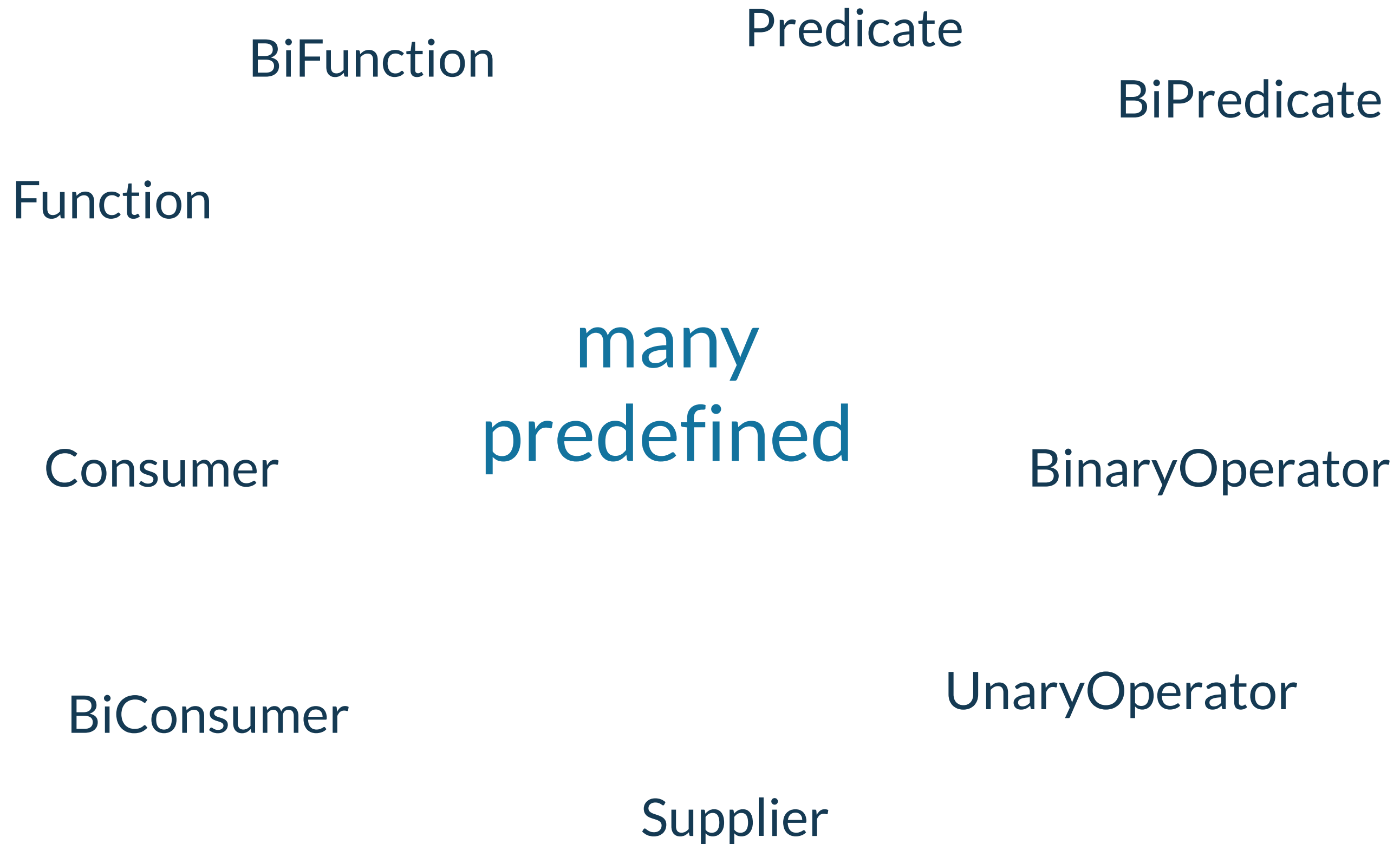
```
interface IntIntFunction<R> {  
    R apply(Integer x, Integer y);  
}
```

```
@FunctionalInterface
```

```
interface DoubleDoubleFunction<R> {  
    R apply(Double x, Double y);  
}
```



Functional interfaces



Functional interfaces

IntFunction

DoubleFunction

LongFunction

many
specialized

ToLongFunction

ToIntFunction

ToDoubleFunction



The Function functional interface

How is the **Function interface** defined?

```
@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() { ... }
}
```



Other methods

They can be used 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> smallerThanOrEqualToZero = 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> len1 = x -> x.length();
```

```
Function<String, Integer> len2 = String::length;
```

```
System.out.println(len1.apply("Hello") + len2.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));
```

Only “effectively final” variable can be captured



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(", ")) );
```





Thank you!

esteco.com

