psc_ TRANSITION TECHNOLOGIES

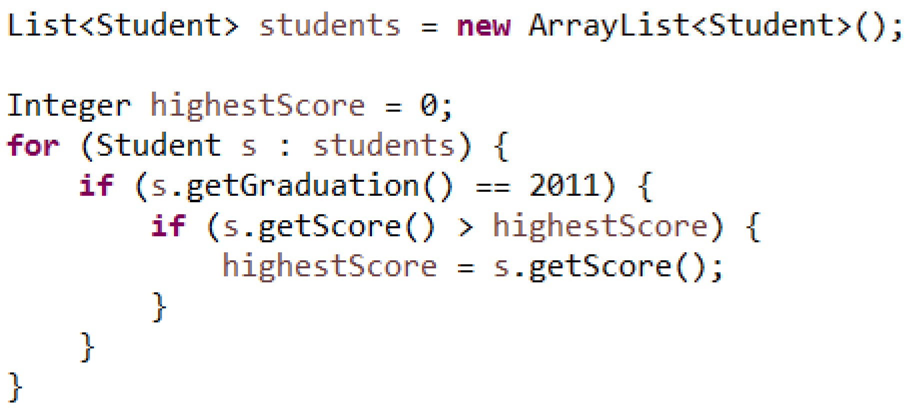
JDK8 Functional Programming Lambda Expressions

Michał Rudnik

Agenda

- Why does Java need Lambda expressions?
- Lambda expression syntax
- Functional interfaces and their definition
- Functional interfaces in the java.util.function package
- Method and constructor references
- Referencing external variables in Lambdas
- Useful new methods in JDK that support Lambdas

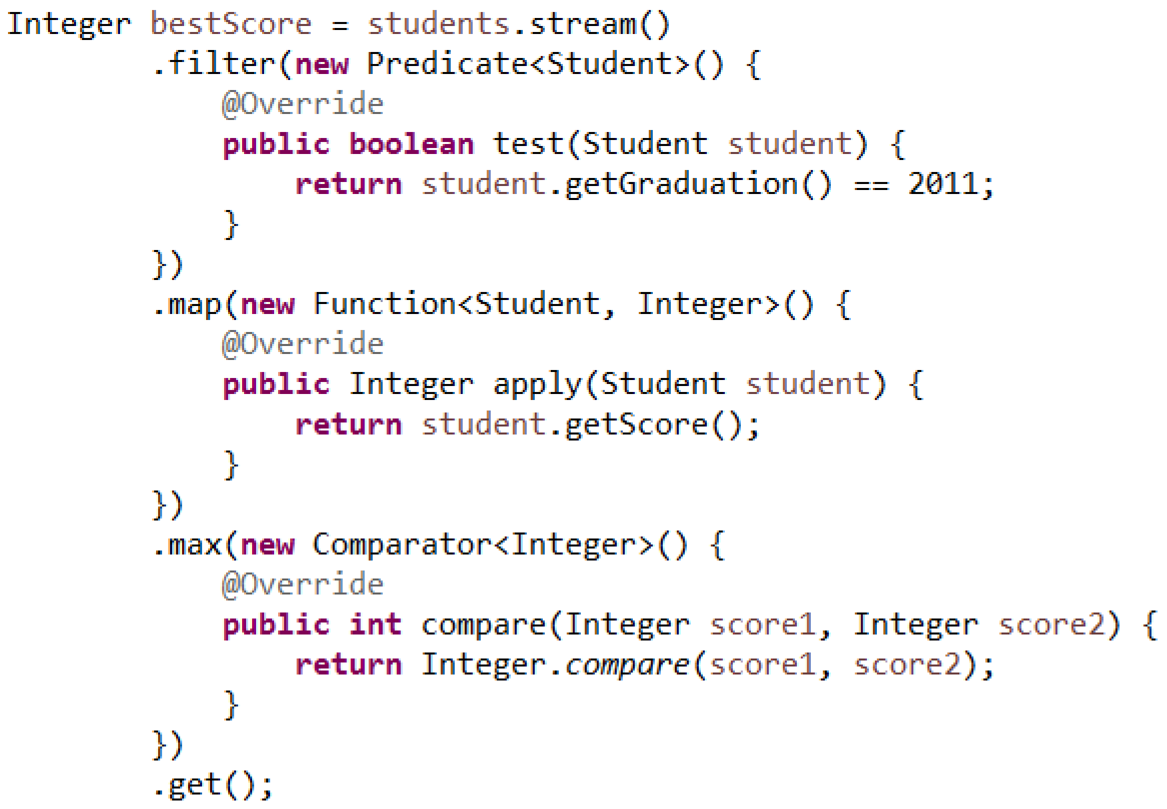
Problems with explicit iteration



```
List<Student> students = new ArrayList<Student>();

Integer highestScore = 0;
for (Student s : students) {
    if (s.getGraduation() == 2011) {
        if (s.getScore() > highestScore) {
            highestScore = s.getScore();
        }
    }
}
```

- Our code controls iteration
- Flow of the execution is predetermined
- Not thread safe
- Variable is mutable

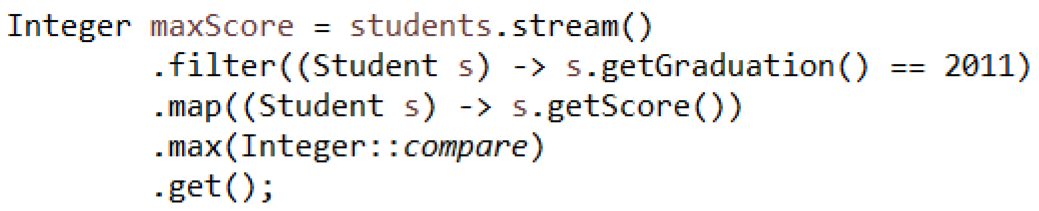
Functional approach



```java
Integer bestScore = students.stream()
        .filter(new Predicate<Student>() {
            @Override
            public boolean test(Student student) {
                return student.getGraduation() == 2011;
            }
        })
        .map(new Function<Student, Integer>() {
            @Override
            public Integer apply(Student student) {
                return student.getScore();
            }
        })
        .max(new Comparator<Integer>() {
            @Override
            public int compare(Integer score1, Integer score2) {
                return Integer.compare(score1, score2);
            }
        })
        .get();
```

- Iteration controlled by library
- Traversal may be done in parallel
- Traversal may be done lazily
- Thread safe
- Ugly ☹

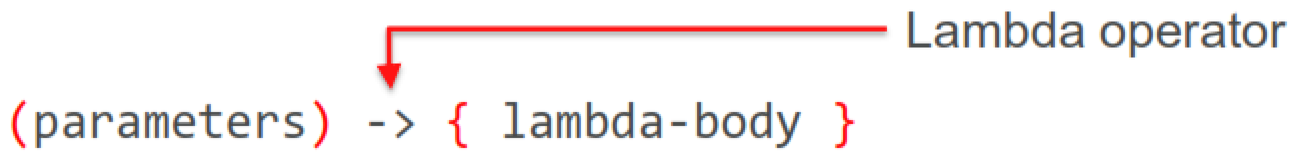
Introducing Lambda Expressions



```
Integer maxScore = students.stream()
        .filter((Student s) -> s.getGraduation() == 2011)
        .map((Student s) -> s.getScore())
        .max(Integer::compare)
        .get();
```

- More readable
- More abstract
- Less error-prone

Lambda Expressions = Anonymous Functions



```
(parameters) -> { lambda-body }
```

- Lambda Expressions are like methods, but they are not associated with a class
- Single line lambda body: optional braces, optional return statement
- Single parameter: optional brackets
- No parameters: required empty brackets

Lambda Expression syntax

- `() -> System.out.println("Hello Lambda")`
- `x -> x + 10`
- `(int x, int y) -> { return x + y; }`
- `(String x, String y) -> x.length() – y.length()`
-
  ```
  (String x) -> {
     listA.add(x);
     listB.remove(x);
     return listB.size();
  }
  ```

Type inference

- Example method definition

```
static T process(List<T> l, Comparator<T> c)
```

- Method call

```
List<String> list = getList();
process(list, (String x, String y) -> x.length() – y.length());
```

- Compiler deduces the type from the context

```
String r = process(list, (x, y) -> x.length() – y.length())
```

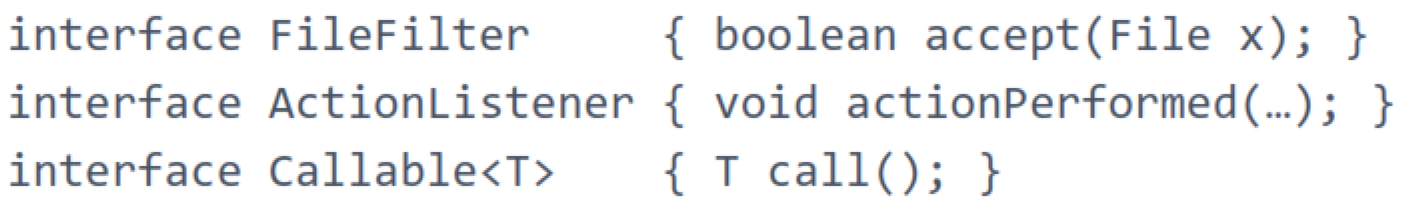
Functional interface definition

- An interface
- Has only one abstract method
  - JDK 8 allows static and default methods in interfaces
- @FunctionalInterface annotation (optional)

- Lambda expressions can be used **anywhere** the type is a **functional interface!**
- Lambda expression provides the **implementation of the single abstract method** of the functional interface!

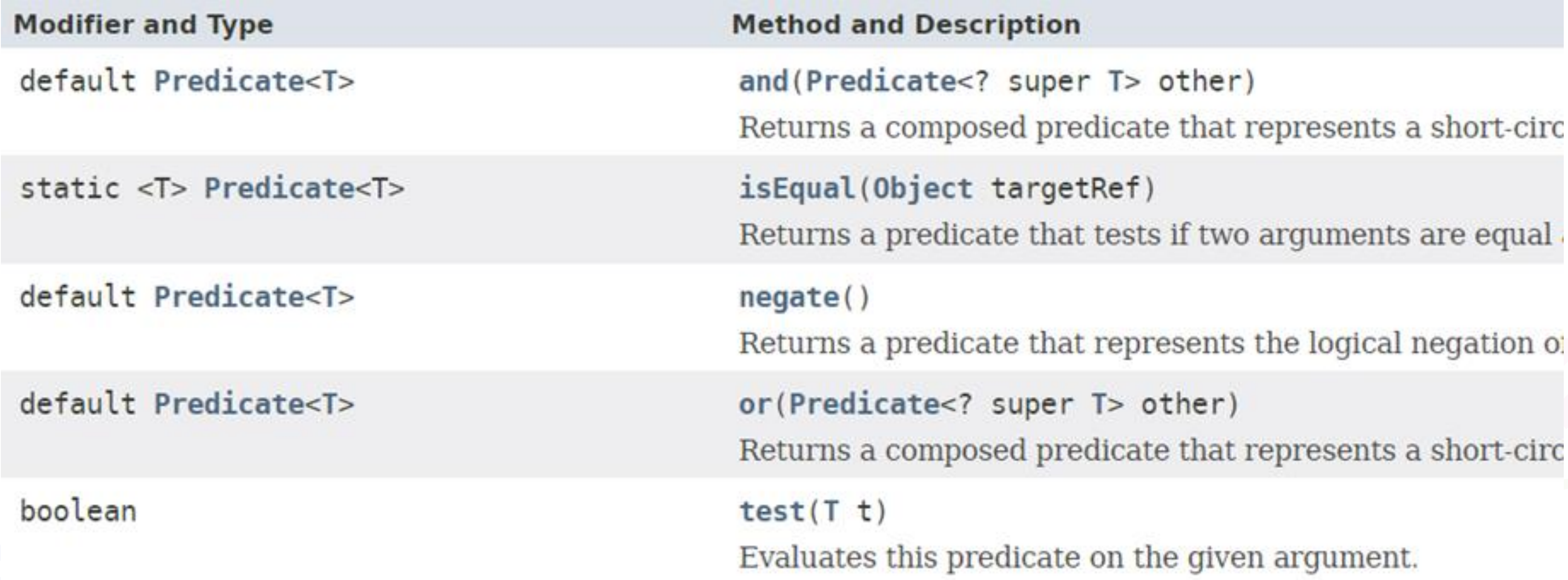
- Lambda expressions enable **passing behaviour as a parameter**

Functional interface examples



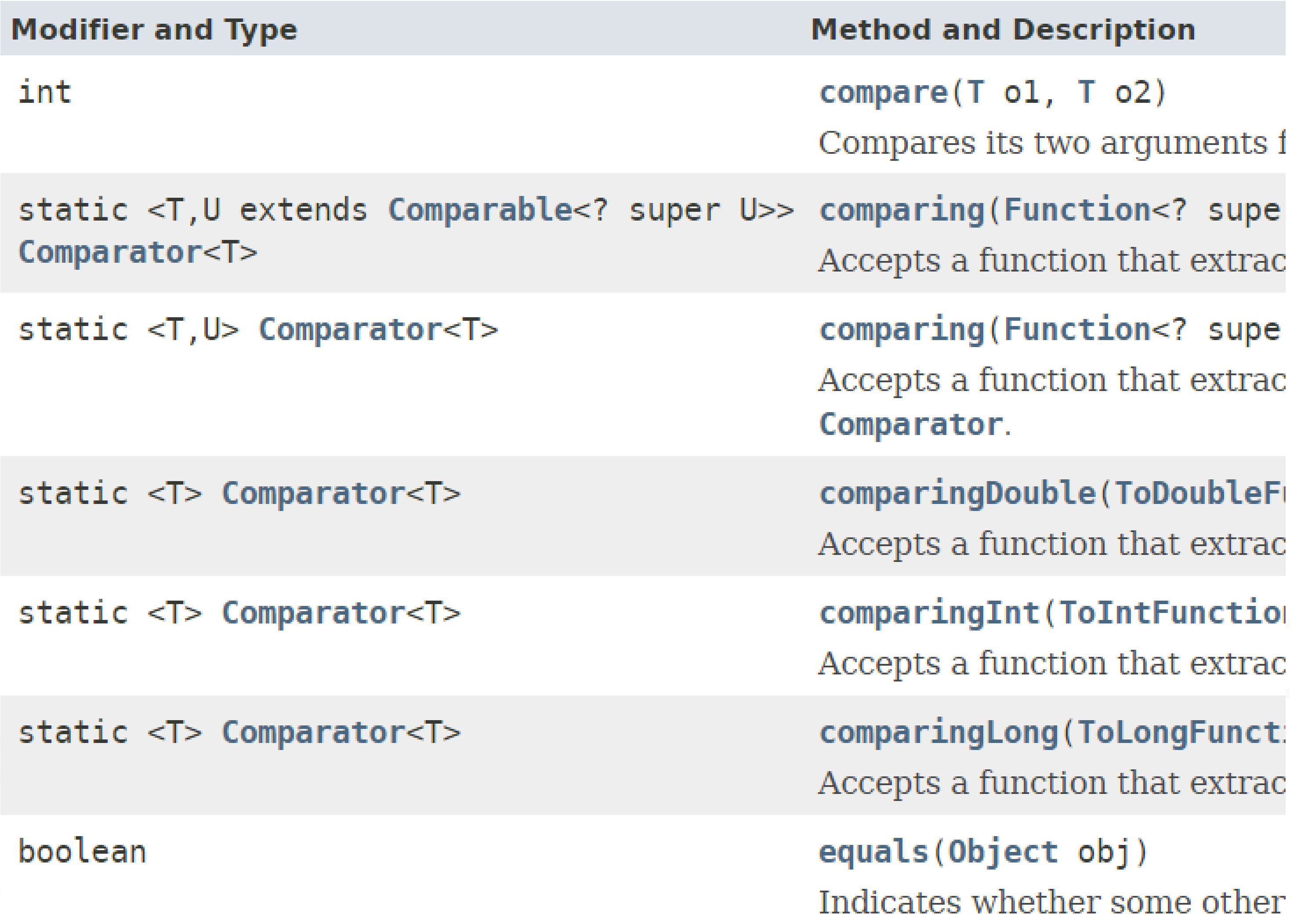
```
interface FileFilter     { boolean accept(File x); }
interface ActionListener { void actionPerformed(…); }
interface Callable<T>    { T call(); }
```

Is this a functional interface? JDK8 Predicate Interface



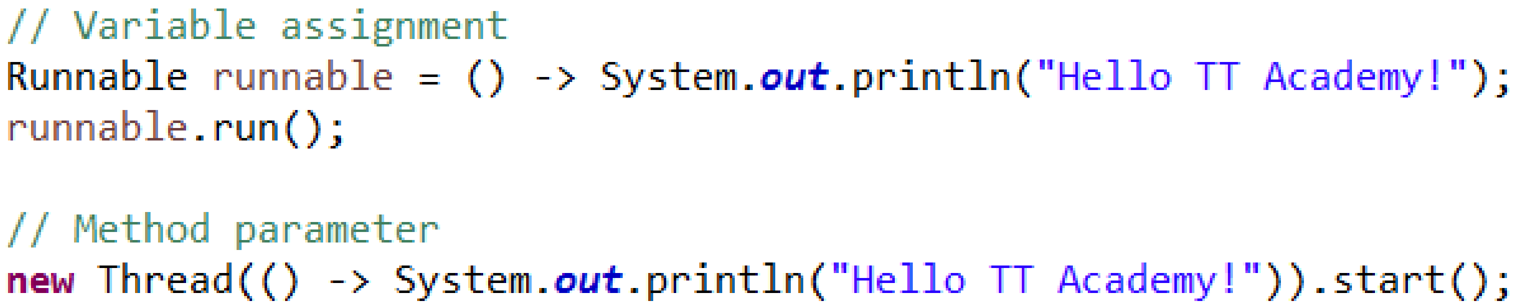
Modifier and Type	Method and Description
default Predicate<T>	and(Predicate<? super T> other) Returns a composed predicate that represents a short-circ
static <T> Predicate<T>	isEqual(Object targetRef) Returns a predicate that tests if two arguments are equal
default Predicate<T>	negate() Returns a predicate that represents the logical negation o
default Predicate<T>	or(Predicate<? super T> other) Returns a composed predicate that represents a short-circ
boolean	test(T t) Evaluates this predicate on the given argument.

Is this a functional interface? JDK8 Comparator Interface



Modifier and Type	Method and Description
`int`	`compare(T o1, T o2)` Compares its two arguments f
`static <T,U extends Comparable<? super U>> Comparator<T>`	`comparing(Function<? supe` Accepts a function that extrac
`static <T,U> Comparator<T>`	`comparing(Function<? supe` Accepts a function that extrac `Comparator`.
`static <T> Comparator<T>`	`comparingDouble(ToDoubleFu` Accepts a function that extrac
`static <T> Comparator<T>`	`comparingInt(ToIntFunctio` Accepts a function that extrac
`static <T> Comparator<T>`	`comparingLong(ToLongFunct` Accepts a function that extrac
`boolean`	`equals(Object obj)` Indicates whether some other

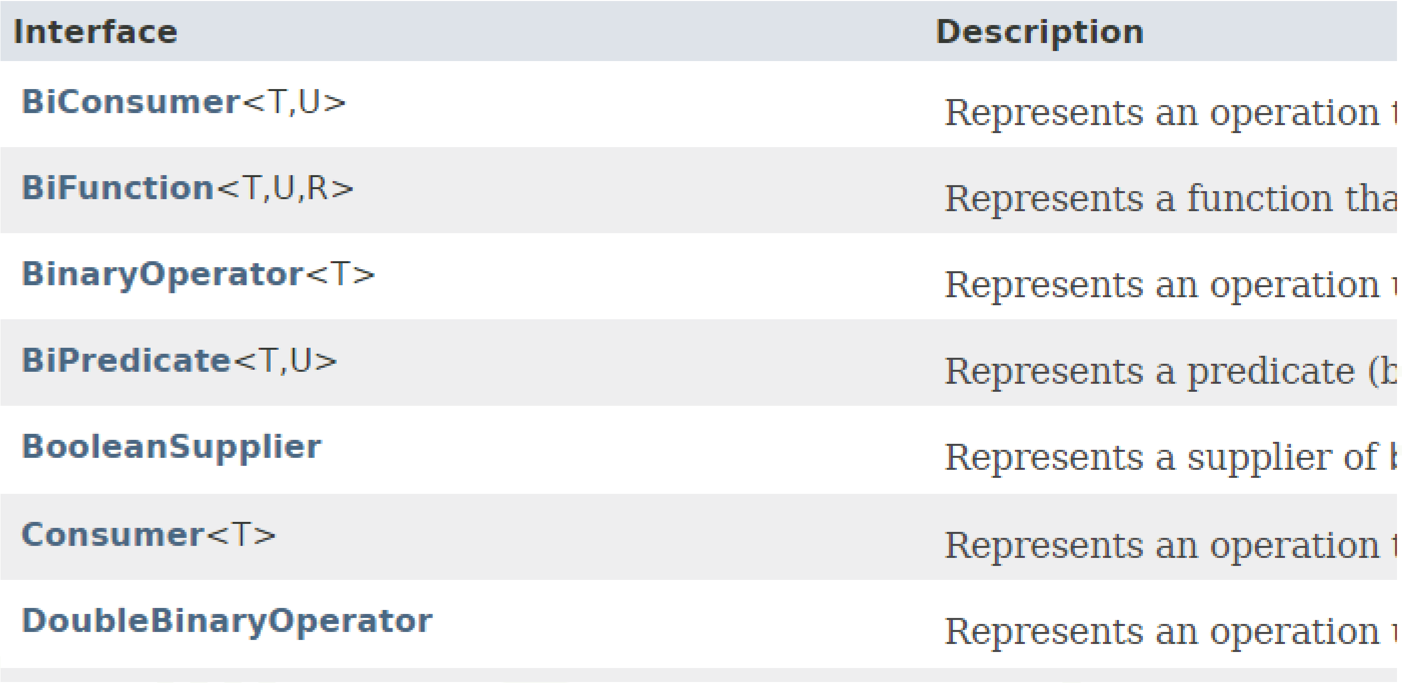
Example uses of Lambda Expression



```
// Variable assignment
Runnable runnable = () -> System.out.println("Hello TT Academy!");
runnable.run();

// Method parameter
new Thread(() -> System.out.println("Hello TT Academy!")).start();
```

java.util.function package

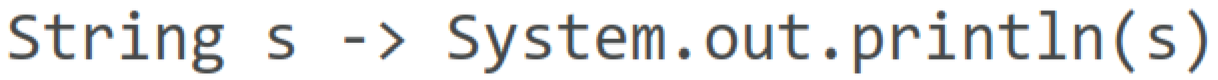
- Well defined set of general purpose functional interfaces
  - all have only one abstract method
  - lambda expressions can be used wherever these types are referenced
  - used extensively in the Java class libraries, especially with the Streams API



Interface	Description
BiConsumer<T,U>	Represents an operation t
BiFunction<T,U,R>	Represents a function tha
BinaryOperator<T>	Represents an operation u
BiPredicate<T,U>	Represents a predicate (b
BooleanSupplier	Represents a supplier of b
Consumer<T>	Represents an operation t
DoubleBinaryOperator	Represents an operation u

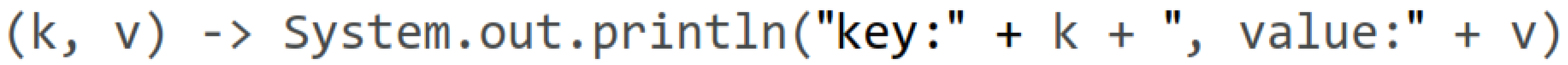
Consumer<T>

- Represents an operation that accepts a single input argument and returns no result



```
String s -> System.out.println(s)
```

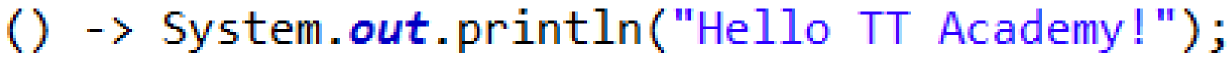
- BiConsumer<T,U> that accepts two arguments and returns no result



```
(k, v) -> System.out.println("key:" + k + ", value:" + v)
```

Supplier<T>

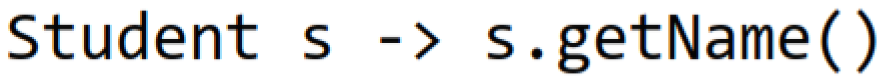
- The opposite of a Consumer – accepts no parameters and produces a result



```
() -> System.out.println("Hello TT Academy!");
```

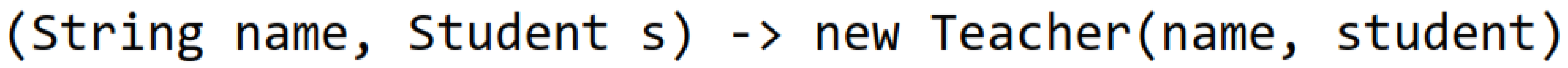
Function<T,R>

- Represents a function that accepts one argument and produces a result
  - type of the argument and result may be different



```
Student s -> s.getName()
```

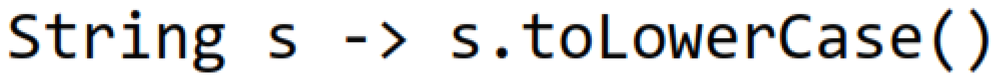
- BiFunction<T,U,R> that accepts two arguments and produces a result



```
(String name, Student s) -> new Teacher(name, student)
```

UnaryOperator<T>

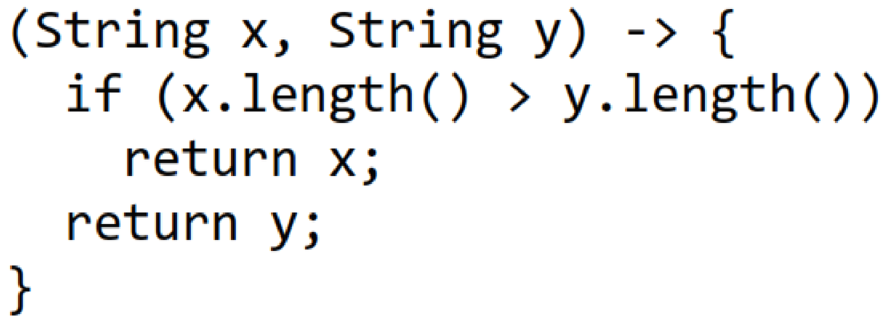
- Specialised form of `Function<T,R>`
- Single argument and result of the same type



```
String s -> s.toLowerCase()
```

BinaryOperator<T>

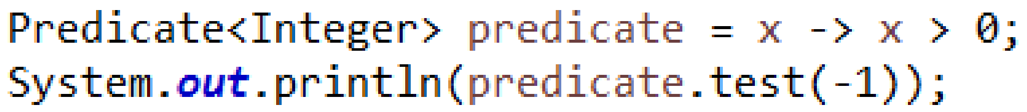
- Specialised form of BiFunction<T,U,R>
- Two arguments and a result of the same type



```
(String x, String y) -> {
  if (x.length() > y.length())
    return x;
  return y;
}
```

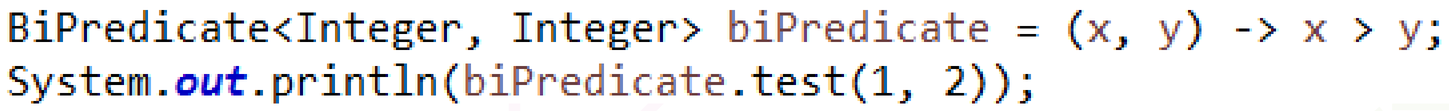
Predicate<T>

- Boolean valued function of one argument



```java
Predicate<Integer> predicate = x -> x > 0;
System.out.println(predicate.test(-1));
```

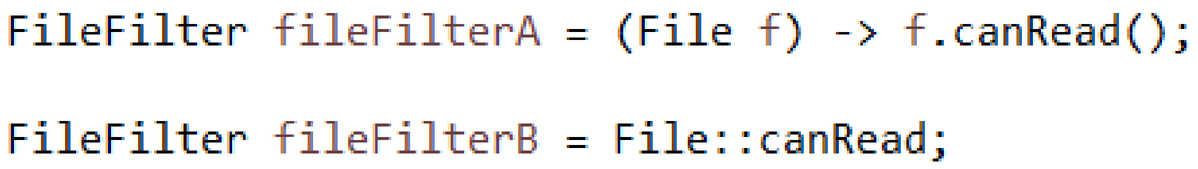
- BiPredicate<T,U> that takes two arguments



```java
BiPredicate<Integer, Integer> biPredicate = (x, y) -> x > y;
System.out.println(biPredicate.test(1, 2));
```

Method and Constructor References

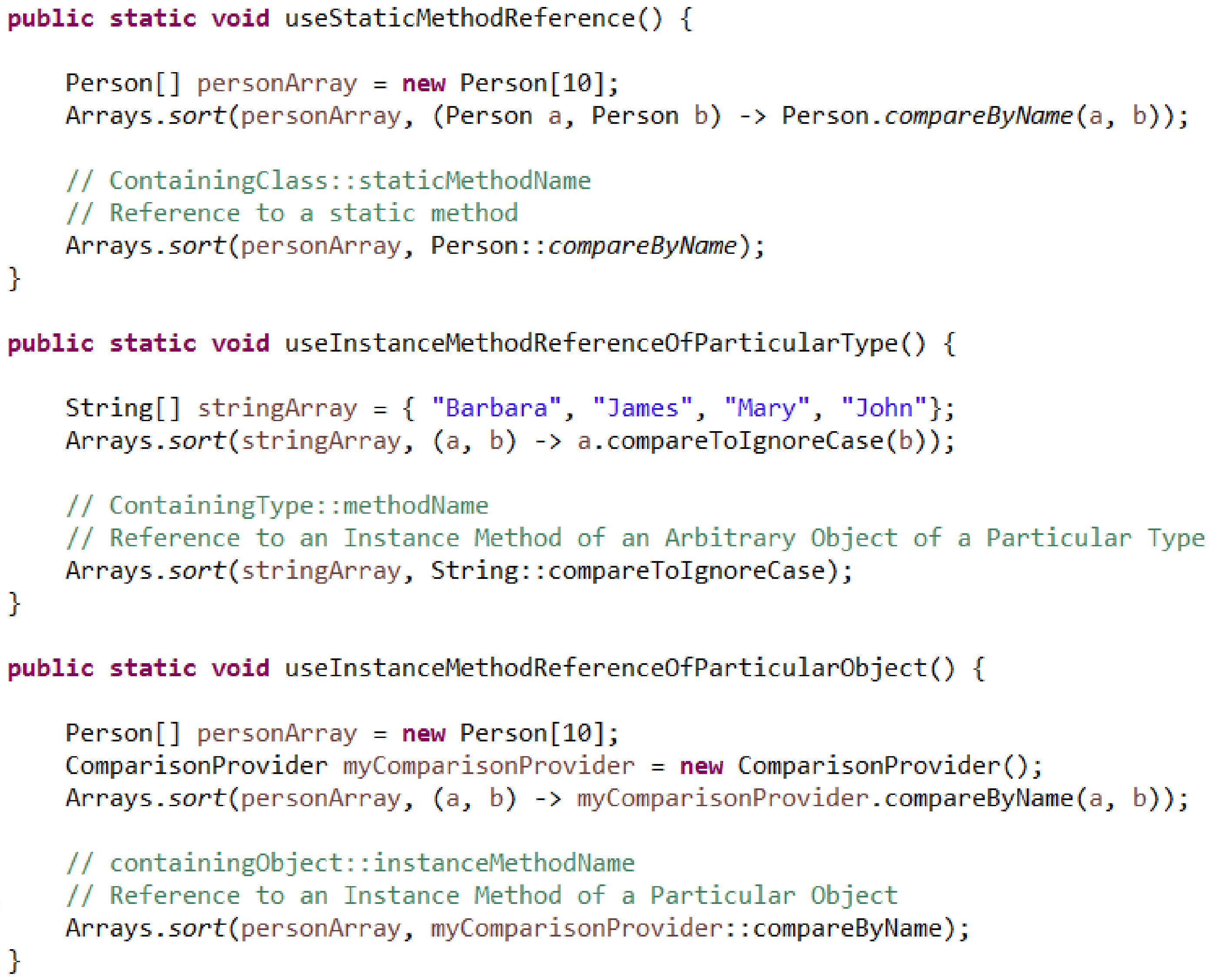
- Method references are shorthand form of defining lambda expressions



```
FileFilter fileFilterA = (File f) -> f.canRead();

FileFilter fileFilterB = File::canRead;
```

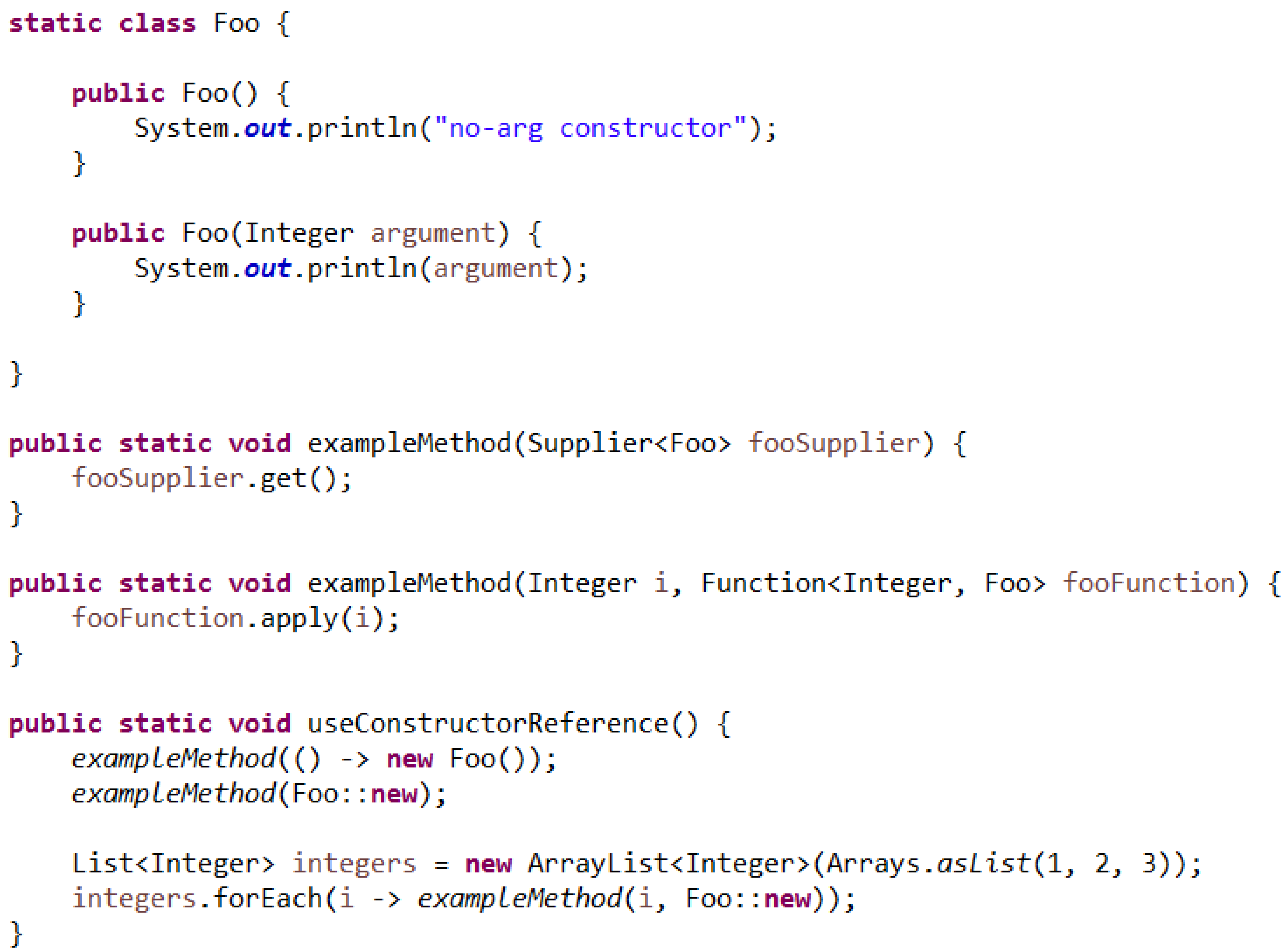
- General format: target_reference::method_name
- 4 kinds of references
  - static method
  - instance method of an arbitrary type
  - instance method of an existing object
  - reference to constructor

Method References



```
public static void useStaticMethodReference() {

    Person[] personArray = new Person[10];
    Arrays.sort(personArray, (Person a, Person b) -> Person.compareByName(a, b));

    // ContainingClass::staticMethodName
    // Reference to a static method
    Arrays.sort(personArray, Person::compareByName);
}

public static void useInstanceMethodReferenceOfParticularType() {

    String[] stringArray = { "Barbara", "James", "Mary", "John"};
    Arrays.sort(stringArray, (a, b) -> a.compareToIgnoreCase(b));

    // ContainingType::methodName
    // Reference to an Instance Method of an Arbitrary Object of a Particular Type
    Arrays.sort(stringArray, String::compareToIgnoreCase);
}

public static void useInstanceMethodReferenceOfParticularObject() {

    Person[] personArray = new Person[10];
    ComparisonProvider myComparisonProvider = new ComparisonProvider();
    Arrays.sort(personArray, (a, b) -> myComparisonProvider.compareByName(a, b));

    // containingObject::instanceMethodName
    // Reference to an Instance Method of a Particular Object
    Arrays.sort(personArray, myComparisonProvider::compareByName);
}
```

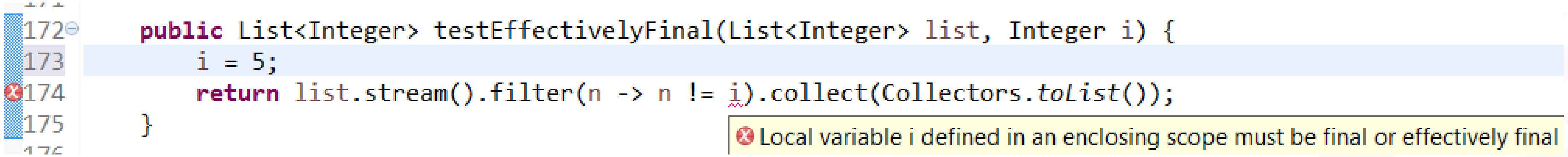
Constructor Reference



```java
static class Foo {

    public Foo() {
        System.out.println("no-arg constructor");
    }

    public Foo(Integer argument) {
        System.out.println(argument);
    }

}

public static void exampleMethod(Supplier<Foo> fooSupplier) {
    fooSupplier.get();
}

public static void exampleMethod(Integer i, Function<Integer, Foo> fooFunction) {
    fooFunction.apply(i);
}

public static void useConstructorReference() {
    exampleMethod(() -> new Foo());
    exampleMethod(Foo::new);

    List<Integer> integers = new ArrayList<Integer>(Arrays.asList(1, 2, 3));
    integers.forEach(i -> exampleMethod(i, Foo::new));
}
```

Referencing external variables in Lambda Expressions

- Lambda expressions can refer to effectively final local variables from the surrounding scope
- Effectively final: a variable that meets the requirements for final variables, even if not explicitly declared final (for example assigned only once)

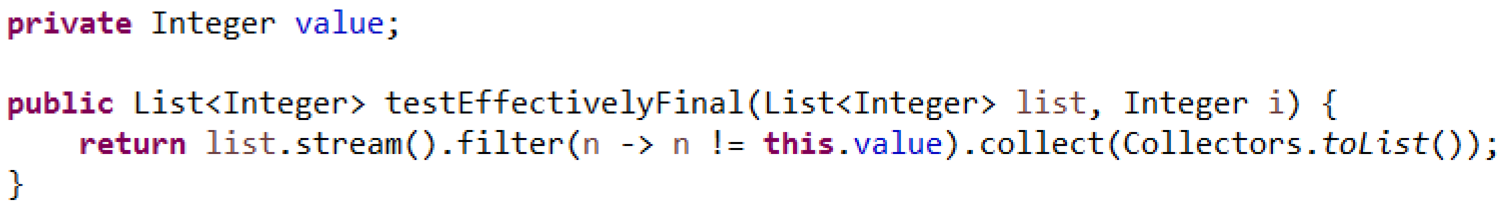


```
    public List<Integer> testEffectivelyFinal(List<Integer> list, Integer i) {
        i = 5;
        return list.stream().filter(n -> n != i).collect(Collectors.toList());
    }
```

Local variable i defined in an enclosing scope must be final or effectively final

What does this mean in a Lambda Expression

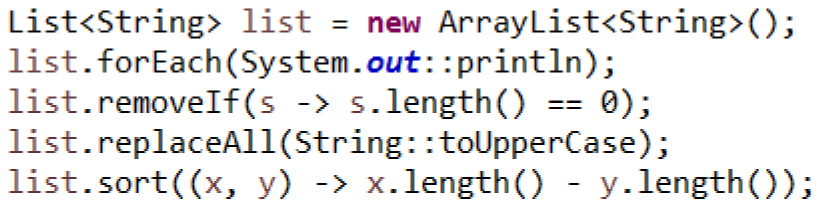
- this refers to the enclosing object, not the lambda itself
- Remember that the lambda is an anonymous function, therefore:
  - it is not associated with a class
  - there can be no this for the lambda



```
private Integer value;

public List<Integer> testEffectivelyFinal(List<Integer> list, Integer i) {
    return list.stream().filter(n -> n != this.value).collect(Collectors.toList());
}
```

Useful new methods in JDK8 that can use lambdas

- `Iterable.forEach(Consumer c)`
- `Collection.removeIf(Predicate p)`
- `List.replaceAll(UnaryOperator o)`
- `List.sort(Comparator c)`



```java
List<String> list = new ArrayList<String>();
list.forEach(System.out::println);
list.removeIf(s -> s.length() == 0);
list.replaceAll(String::toUpperCase);
list.sort((x, y) -> x.length() - y.length());
```

Summary

- Lambda expressions provide a simple way to pass behavior as a parameter or assign to a variable
- They can be used wherever a functional interface type is used
- Lambda provides the implementation of the single abstract method
- Method and constructor references can be used as shorthand
- Several useful new methods in JDK 8 that can use lambdas